

COMPUTE!'s Reference Guide to  
**COMMODORE**

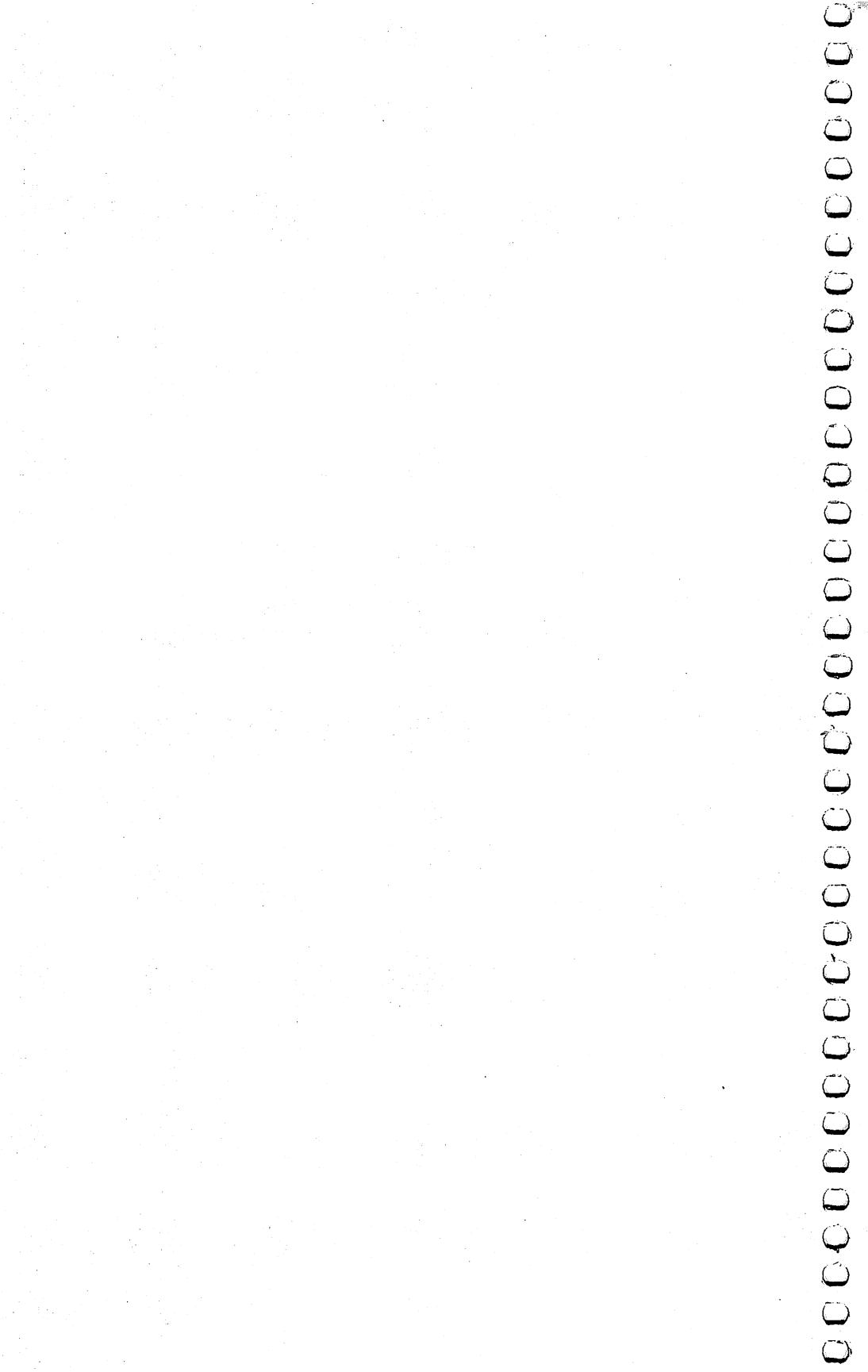
# **64** **GRAPHICS**

How to create graphics displays,  
pictures, and animation on the  
Commodore 64<sup>TM</sup>, plus five  
type-in-and-run graphics utilities.

---

---

By John Heilborn



COMPUTE!'s Reference Guide to  
**COMMODORE**

# **64** **GRAPHICS**

By John Heilborn

**COMPUTE!**™ Publications, Inc.   
One of the ABC Publishing Companies

Greensboro, North Carolina

Commodore 64 is a registered trademark of Commodore Electronics Limited.

Copyright 1983, COMPUTE! Publications, Inc. All rights reserved

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

ISBN 0-942386-29-9

10987654321

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is a subsidiary of American Broadcasting Companies, Inc., and is not associated with any manufacturer of personal computers. Commodore 64 is a trademark of Commodore Electronics Limited.

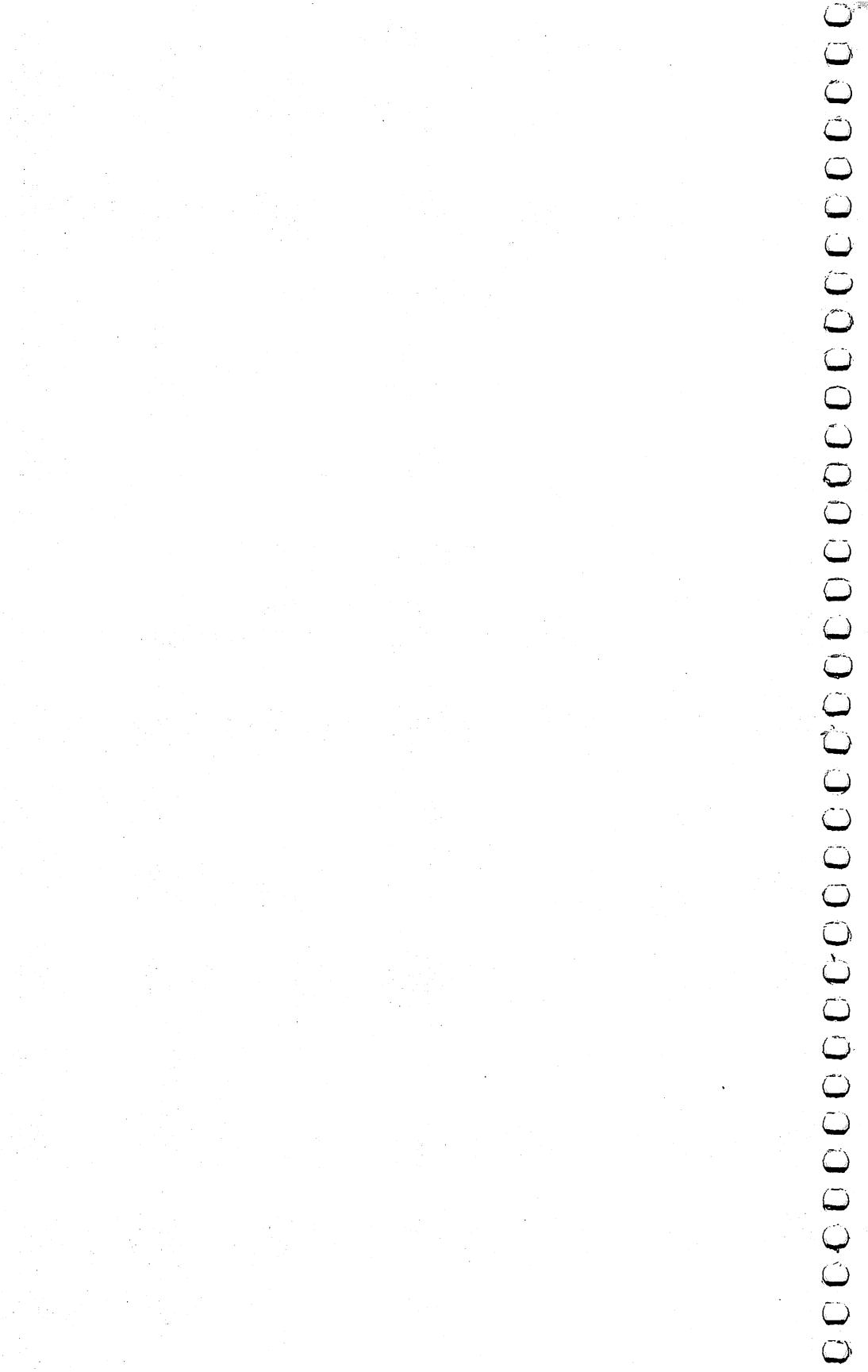
# Contents

## Foreword

<b>Chapter 1.</b> Designing Graphic Shapes .....	3
<b>Chapter 2.</b> Color .....	81
<b>Chapter 3.</b> Animation .....	151
<b>Chapter 4.</b> Advanced Graphics .....	177

## Appendices

<b>A:</b> A Beginner's Guide to Typing In Programs .	191
<b>B:</b> How to Type In Programs .....	193
<b>C:</b> Screen Location Table .....	195
<b>D:</b> Screen Color Memory Table .....	196
<b>E:</b> Screen Color Codes .....	197
<b>F:</b> ASCII Codes .....	198
<b>G:</b> Screen Codes .....	202
<b>H:</b> Commodore 64 Keycodes .....	204
<b>I:</b> Character Patterns and Screen Codes, Set 1 .	205
<b>J:</b> Character Patterns and Screen Codes, Set 2 .	211
<b>Index</b> .....	217



# Foreword

Ever since it was invented, the television set has had its own way in our homes. If a program wasn't broadcast, we couldn't receive it. If a story we wanted to see wasn't filmed, we couldn't see it. TV was a passive medium.

All that's changed, now that you have a Commodore 64 hooked up to your television. The television will display whatever you tell it to display. New owners usually begin by running commercially produced programs or games. But because the 64 is a fully programmable computer, you don't have to be limited by what software developers have produced.

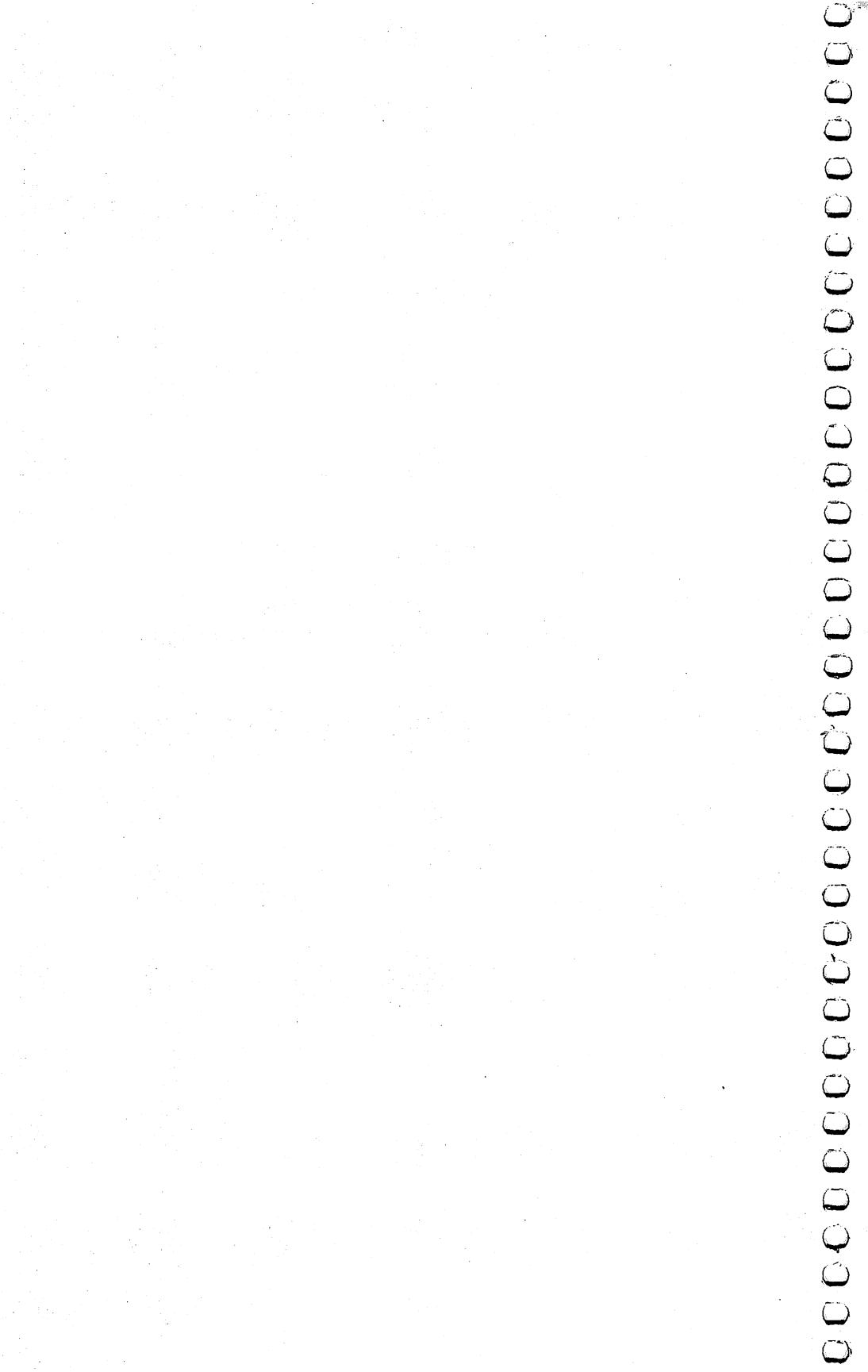
John Heilborn, well-known author of books about Commodore computers, clearly and accurately explains all you need to know to make your television screen show you anything you want to see.

- With character graphics, you can combine words and pictures however you want.
- If the built-in character set doesn't have the characters you need, you can design your own.
- Fine-line drawings are possible with bitmapping.
- Business and educational software can be created with different-colored windows using the extended color mode.
- You can animate figures with sprite graphics.

And to help make all of this as easy as possible, you can type in and run any or all of the complete editor programs:

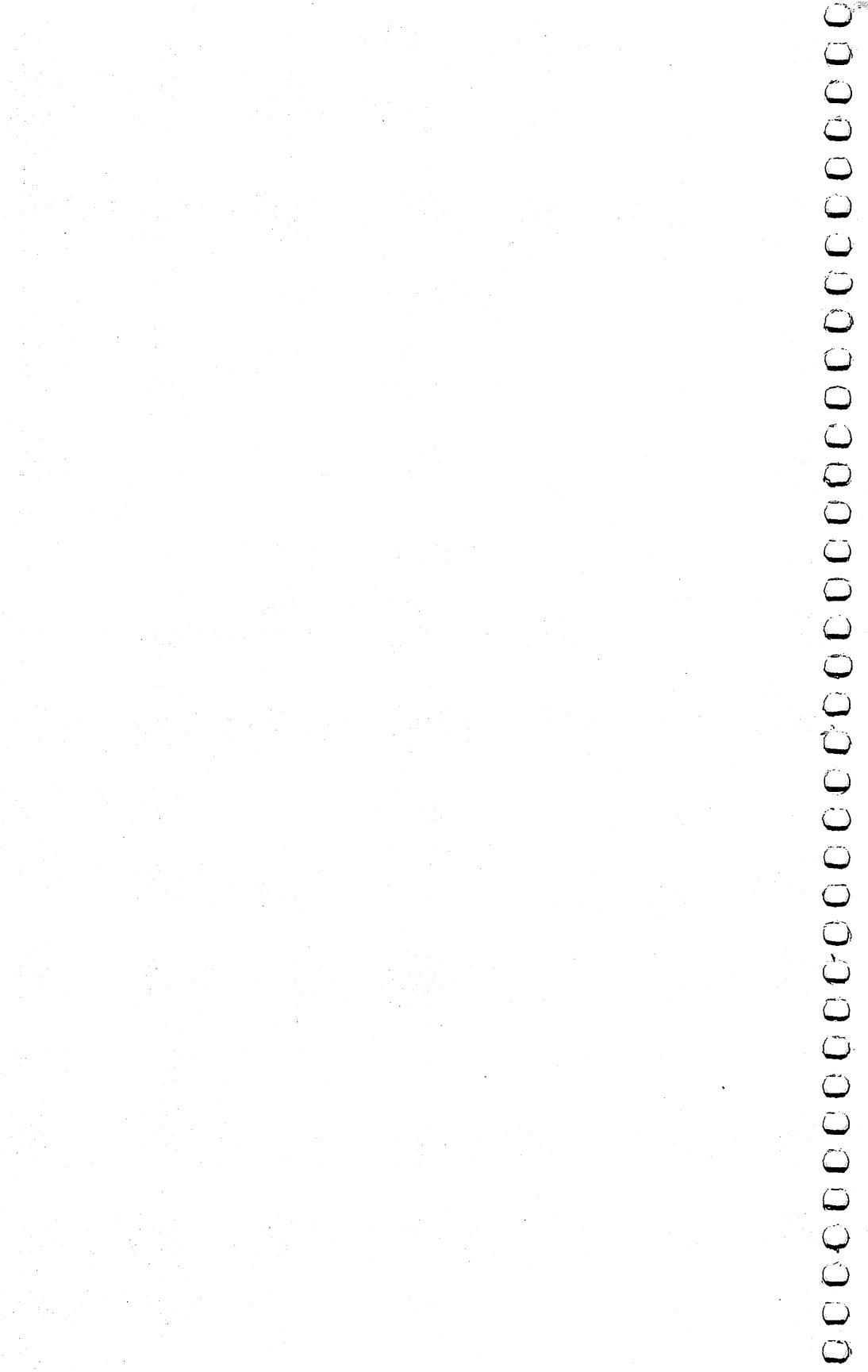
- Character editor
- Sprite editor
- Bitmapped screen editor
- Multicolor character editor
- Multicolor sprite editor

What's on TV tonight? With the help of *COMPUTE!'s Reference Guide to Commodore 64 Graphics*, you can create your own show.



# 1

# Designing Graphic Shapes



# Designing Graphic Shapes

## 1

From the very moment you first turn on the Commodore 64 computer, there are a wide variety of graphic symbols at your command. Some of them, such as the letters of the alphabet or the numbers zero through nine, have specific meanings to the computer. Others, however, are simply shapes with no particular meaning at all. These graphic shapes have been included in the 64 primarily for your use. By combining them effectively, you can create a wide variety of graphic displays with ease.

### **Putting Characters on the Screen**

The simplest way to put any character on the screen is to type it in from the keyboard. Figure 1-1 shows the layout of the 64 keyboard and all of the characters available by pressing individual keys.

If you look closely, you will find that most of the keys on the keyboard have at least two different symbols on them; many have three. These additional characters can be accessed by typing the keys in combination with the SHIFT or Commodore keys. (The Commodore key is the one in the lower-left corner of the keyboard, with the Commodore logo on it.)

### **PRINT Statements**

In addition to displaying characters typed at the keyboard, you can also program the computer to display any of the characters you enter by using the PRINT command. To PRINT characters in this fashion, you must enclose the characters to be printed in quotes. The computer will then display them on the screen. For example, enter the following line:

```
10 PRINT "THE 64 CAN PRINT LETTERS"
```

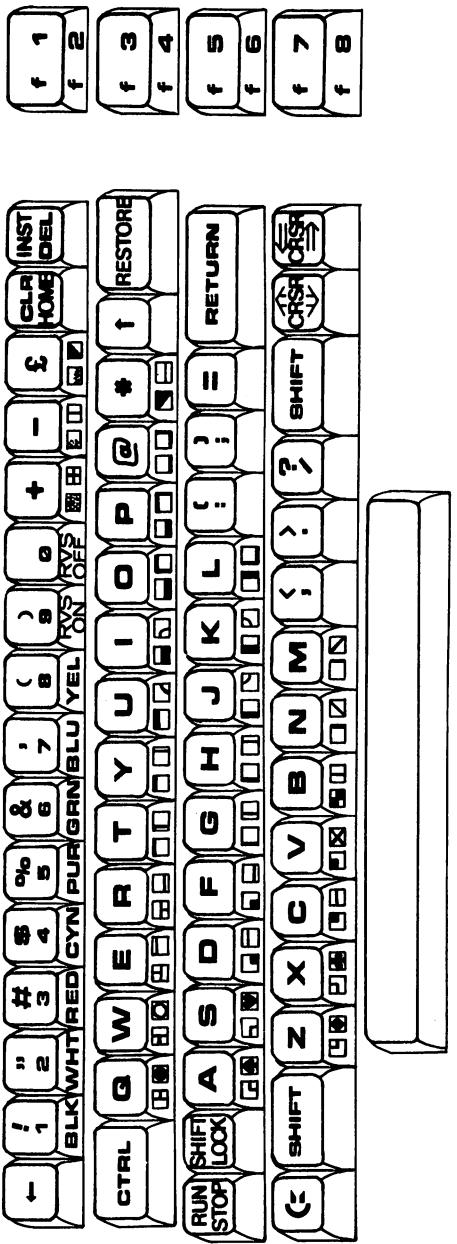
When you run this routine the computer will display the message:

**THE 64 CAN PRINT LETTERS**

If you put a different message between the quotes, the computer will display that instead.

# Designing Graphic Shapes

**Figure 1-1.** The Commodore 64 Keyboard



# Designing Graphic Shapes

The computer can also INPUT a message that you type in from the keyboard during program execution:

```
10 INPUT "INPUT YOUR MESSAGE"; M$  
20 PRINT "THIS IS YOUR MESSAGE:"  
30 PRINT M$
```

This routine asks you for an INPUT. It then takes that INPUT and prints it, just as the first routine did. The advantage to this routine, however, is that you can enter a different message each time you run the program without having to rewrite the program itself.

## Problems Using PRINT

Altogether, there are 512 preprogrammed characters stored in the 64. Each of them can be printed onto the screen in any of 1000 different locations. Here is a routine that INPUTs a character and then PRINTs it 1000 times, filling the screen:

```
10 INPUT "INPUT ONE CHARACTER"; C$  
20 FOR R = 1 TO 1000  
30 PRINT C$;  
40 NEXT
```

## The READY Problem

As you can see, there is a problem with this routine. After the computer prints the last character on the last line, it scrolls up a few lines and prints the READY message. This can ruin a carefully planned display. One way to avoid the READY message is to add a line at the end of the program to make it loop back on itself:

```
50 GOTO 50
```

With this line at the end of the program, it will never end and so will not display the READY message. To end the loop, press the RUN/STOP and RESTORE keys at the same time.

## The Scrolling Problem

While this loop does prevent the computer from printing the READY message, the screen still scrolls up when it reaches the last character on the bottom line. This is because each time the computer reaches the end of a line, the cursor jumps to the beginning of the next line. In this case, of course, the cursor will not be printed because we have prevented the routine from ending. Nevertheless, the computer makes room for it by scrolling up.

Since it is the last character that causes the screen to scroll, not printing that position should eliminate the problem. To con-

# Designing Graphic Shapes

firm this, try reducing the number in line 20 from 1000 to 999.

```
20 FOR R = 1 TO 999
```

Leaving out the last position does indeed work. The screen remains solidly in place.

Does this mean, then, that if we want to keep the display from scrolling, we can never use the very last position on the screen?

No, it just means that we need another way of putting the last character on the screen. One way to do this is to POKE the last character onto the screen instead of PRINTing it. To get an idea of how this is done, let's take a look at the way the computer handles the screen display.

## Screen Memory

On the 64, all of the characters that are to be displayed are stored in a section of memory called, appropriately enough, screen memory. The screen display is organized as 25 rows of 40 characters each. If you could see all of the individual character positions on the screen, they would appear very much as the individual boxes do on the grid in Figure 1-2.

Each of the little boxes represents a single location in screen memory, and every one of the memory locations contains the screen code for the character displayed in that position. When you first turn on the 64, screen memory resides between memory locations 1024 and 2023.

**Screen codes.** Every character has a different screen code. The codes for each of the characters in the standard character set are listed in Appendix I. To put a character on the screen, POKE its screen code into the memory location for the screen position at which you want it displayed.

**Alternate character set.** Appendix J lists the screen codes for the characters in the alternate character set. You can switch back and forth between the standard and alternate character sets by pressing the SHIFT and Commodore keys at the same time.

## POKEing Characters onto the Screen

The following routine will clear the screen and then fill it with 1000 asterisks using the POKE command:

```
10 PRINT "[CLR]"
20 FOR R = 1024 TO 2023
30 POKE R, 42
40 NEXT
```

# Designing Graphic Shapes

Figure 1-2. Character Graphics Screen Layout

	Columns																																										
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39			
Row	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39			
1	24	0																																									
2	16	1																																									
3	05	2																																									
4	14	3																																									
5	85	4																																									
6	22	5																																									
7	26	6																																									
8	30	7																																									
9	38	8																																									
10	25	9																																									
11	65	10																																									
12	95	11																																									
13	25	12																																									
14	Row	13																																									
15		14																																									
16		15																																									
17		16																																									
18		17																																									
19		18																																									
20		19																																									
21		20																																									
22		21																																									
23		22																																									
24		23																																									

# Designing Graphic Shapes

Using this method, every screen location can be filled and the screen will never scroll up. In fact, although the READY message is printed at the end of the program, it is printed at the top of the screen because that was the last cursor position. The cursor does not move when you POKE characters to the screen, and since it is the cursor that triggers the scroll, this method does not scroll the screen. Of course, if you still want to prevent the READY message from appearing, you can use the line

```
50 GOTO 50
```

again, and the computer will keep looping until you hit RUN/STOP and RESTORE.

If your screen does not fill with asterisks, you have a recent version of the 64. Add this line to the routine above:

```
25 POKE R+54272, 14
```

On newer versions of the 64 you need to POKE a number into *color memory* for each character you POKE into screen memory. These 64s automatically reset all of the color memory locations to the same color as the background each time the screen is cleared, so characters POKEd to the screen seem invisible unless color memory is POKEd with some other value. This has no effect on PRINTing to the screen.

## Programming Complex Characters

One thing you may have noticed as you ran the two preceding programs is that the first one (which used the PRINT command) ran quite a bit faster than the second one. That is because PRINT is quicker than POKE. To compare their relative speeds, enter and run the following program:

```
10 PRINT "[CLR]";  
20 FORR=1024 TO 2023  
30 POKE R,42: NEXT  
40 FORG=0TO500:NEXT  
50 PRINT "[CLR]{WHITE}";  
60 FORR=1TO999:PRINT "*"; :NEXT  
70 PRINT "[HOME]"
```

PRINT is actually almost twice as fast as POKE.

PRINT is also quite a bit more versatile in making graphic shapes because it allows you to combine a number of individual characters into a single complex character. The POKE command, on the other hand, puts only one character on the screen at a time.

# Designing Graphic Shapes

For instance, instead of PRINTing 1000 single asterisks in line 60 of the PRINT routine above, we could change the line to read:

```
60 FORR=1TO124:PRINT"**";:NEXT
```

By PRINTing two asterisks at a time, the screen will fill twice as fast as before. Of course, you can also change the routine to print three, four, or more asterisks at once. Each time you do this, you will increase the speed at which the screen fills.

## New Characters from Old

Another way we can take advantage of the 64's PRINT command is by making large pictures using several smaller characters. For example, we can write a program that draws rectangles based on the dimensions we specify. Here is one way this can be done:

```
10 INPUT "HEIGHT"; H
20 INPUT "WIDTH"; W
30 SQ$="O"
40 FOR R=1 TO W
50 SQ$ = SQ$+"{Y}":NEXT
60 SQ$=SQ$+"P"
70 FOR R=1 TO H
80 SQ$=SQ$+"{DOWN}{ 2 {LEFT}}{RIGHT}{M}":NEXT
90 SQ$=SQ$+"{DOWN}{LEFT}@"
100 FOR R=1 TO W
110 SQ$=SQ$+"{ 2 {LEFT}}{P}":NEXT
120 SQ$=SQ$+"{ 2 {LEFT}}L"
130 FORR=1 TO H
140 SQ$=SQ$+"{UP}{LEFT}{H}":NEXT
150 PRINTSQ$
160 FOR R=1 TO H
170 PRINT"{DOWN}";:NEXT
```

When you run this program, the computer will ask you to enter two dimensions, height and width. After you enter the dimensions, it will draw a rectangle to those specifications. In so doing, it will also assign the characters that make up the rectangle to the variable SQ\$. That way, each time you enter

**PRINT SQ\$**

the computer will PRINT your rectangle. It is as if SQ\$ were a single complex character, which can be PRINted as quickly as any other character.

Be careful, though. This program can create some very long strings (nonnumeric variables), and the longest allowable string is 255 characters. If you exceed this limit, you will get the message:

**?STRING TOO LONG ERROR**

# Designing Graphic Shapes

If this happens, try using a smaller number in one of the two dimensions.

## Cursor Addressing

You may have noticed in the listing above that in addition to the special line and corner characters used to draw the rectangles, the strings also include some cursor control characters. These were used to position the special characters used in the complex shapes. Cursor control characters are seen as standard characters by the 64 and are always treated as a part of the complex character. So they can be programmed in the same way as any other character.

As you saw in the previous example, this can be a very useful function. But in addition to helping create special characters, the cursor controls can also allow you to PRINT strings anywhere on the screen. This works because the next character PRINTed will always appear in the same position as the cursor.

Here is a program that positions the cursor on the screen based on a pair of X,Y coordinates. (The X coordinate is the cursor's left-right position on the screen. The Y coordinate specifies the up-down position.)

```
5 PRINT "{CLR}";
10 INPUT "X, Y COORDINATES{ 10 {SPACES}}"
    { 10 {LEFT}}";X,Y
20 C$ = "{HOME}"
30 FOR R=1 TO X
40 C$ = C$+"{RIGHT}"
50 NEXT
60 FOR R=1 TO Y
70 C$ = C$+"{DOWN}"
80 NEXT
90 PRINT C$;"{RVS} {OFF}";X;",";Y
100 PRINT "{HOME}";
110 GOTO 10
```

While this program PRINTs a small rectangle and its X,Y position on the screen, you could PRINT anything you chose in that position. Routines such as this one can be very useful in building complex displays.

## Moving Memory Around

Having the capability of combining characters this way opens the door to some very interesting possibilities. For example, the 64's character set resides in a section of memory called ROM (Read

# Designing Graphic Shapes

Only Memory) that is permanently programmed. By reading the character shapes from these locations in memory and using some complex characters, it is possible to make large copies of the pattern of every character in the 64.

## The 64's Crowded Memory

There are a few details, however, that must be dealt with in trying to read the character ROM. The most important of these is that the 64 has more than 64K of memory available. In addition to the 64K of RAM (Random Access Memory) that gives the 64 its name, it also has the operating system program (Kernel routines), the BASIC programming language, and the character set in ROM (Read Only Memory). Additional memory locations are required to control the operation of the various input/output (I/O) chips.

This gives the 64 a total of more than 87,000 memory locations. But the microprocessor in the 64 can address only 65,536 locations at any one time. This makes it impossible for the 64 to read every one of these memory locations at one time.

## Bank Switching

To compensate for this, the 64 has the capability of switching between various blocks of memory. For example, a section of ROM may reside at the same memory locations as a section of RAM. In a sense, they are placed one on top of the other. By switching banks, the microprocessor can look at the RAM or the ROM as it requires. The only constraint is that they cannot both be available at the same time. This allows the 64 to access the character set or any other block of memory it needs just by switching back and forth between blocks.

To show how this works, here's a program that PRINTs the number stored in one of the hidden memory locations, location 53248 (the first byte of the character ROM).

```
10 PRINT PEEK(53248)
20 POKE 56334, 0: POKE 1, 51
30 A=PEEK(53248)
40 POKE 1, 55: POKE 56334, 1
50 PRINT A
```

The program first displays the value of the location obtained before bank switching, and then again after bank switching. Since this process is a bit tricky, we'll go through it step by step.

1. Line 10 PEEKs location 53248 and displays the value stored there.

# Designing Graphic Shapes

2. Line 20 POKEs a zero into location 56334, which turns off the keyboard scanner. This is necessary because if it is running, the keyboard scanner will interfere with reading the data properly.

3. Line 20 also POKEs location 1 with the value 51. This switches the character ROM into position so it can be read by the microprocessor.

4. Line 30 assigns the value in location 53248 to the variable A. It does this instead of PRINTing the value immediately, because the I/O chips (which are also responsible for handling the video display) became unavailable to the microprocessor when we switched the character ROM into position for it to be read.

5. Line 40 restores locations 56334 and 1 to their former values, which switches the character ROM and the I/O chips back again. This line also turns on the keyboard scanner again.

6. Line 50 PRINTs the value of A.

## How Characters Are Formed

The data we have just read is the character pattern for the top of the @ symbol. The reason it doesn't appear to us that way is because it is coded as a series of binary bits. To see the pattern, we'll need to convert the number calculated in the routine above (the decimal number 60) into its binary equivalent.

## Interpreting Binary Data

There are a number of ways in which this can be accomplished. Here is a mathematical approach:

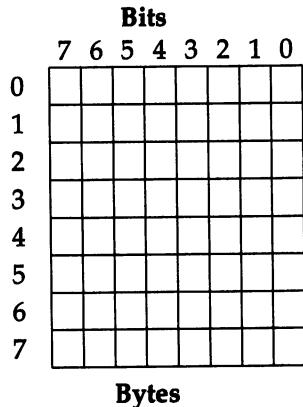
```
10 INPUT A
20 FOR R=0 TO 7
30 IF A/2=INT(A/2) THEN G$="0"+G$:GOTO50
40 G$="1"+G$
50 A=INT(A/2):NEXT
60 PRINTG$
```

By entering the number 60 into this conversion program, we find that its binary equivalent is 00111100.

Unfortunately, this is still not too enlightening until we look at the organization of each character. The grid in Figure 1-3 shows how the individual bits (0's and 1's) are organized in a single character as it is stored in memory. Each character contains eight bytes and each byte has eight bits. This makes a total of 64 bits for each character pattern. By storing 1's and 0's in these 64 locations, the shape of the character is formed.

# Designing Graphic Shapes

**Figure 1-3. The Character Matrix**



The @ character is stored in locations 53248 through 53255. Using the two programs above, we can determine the binary values of the numbers stored in those locations. They are:

Decimal Value	Binary Value	Memory Location
60	00111100	53248
102	01100110	53249
110	01101110	53250
110	01101110	53251
96	01100000	53252
98	01100010	53253
60	00111100	53254
0	00000000	53255

## Patterns of Light and Dark

By darkening the rectangles that contain 1's and leaving the rectangles that contain 0's blank, we can get a picture of the character as shown in Figure 1-4.

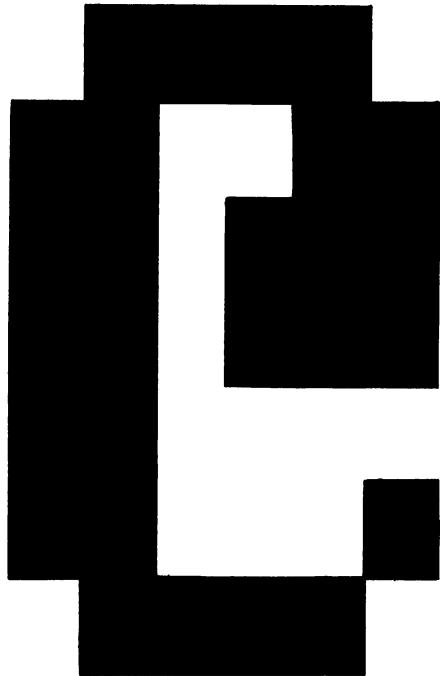
If we combine the two programs into a single routine and PRINT dark and light rectangles instead of 1's and 0's, we can produce a program that will display the patterns of every character in the 64 character set:

```
10 FOR R=53248 TO 57343  
20 POKE 56334, 0: POKE 1, 51  
30 A = PEEK(R)  
40 POKE 1, 55: POKE 56334, 1
```

# Designing Graphic Shapes

```
50 G$=""  
60 FOR F=0 TO 7  
70 IF A/2=INT(A/2) THEN G$=" "+G$:GOTO90  
80 G$="{RVS} {OFF}"+G$  
90 A=INT(A/2):NEXT  
100 PRINTG$  
110 NEXT
```

**Figure 1-4**



## Complex Characters

It is possible to create new characters by combining several of the standard characters into a new, complex character. Some further examples of these can be found in Programs 1-1 through 1-3.

### Program 1-1. Playing Cards

```
10 PRINT "[ 6 {SPACES}]U{ 3 *}I{ 4 {SPACES}}"  
20 PRINT "[ 6 {SPACES}]BX{ 2 {SPACES}}{Q}*I  
{ 2 {SPACES}}"
```

# Designing Graphic Shapes

```
30 PRINT" { 6 {SPACES}}B X BS{Q}*I"
40 PRINT" { 6 {SPACES}}B{ 2 {SPACES}}XBSBPB"
50 PRINT" { 6 {SPACES}}J*{R}*KSB{N}B"
60 PRINT" { 8 {SPACES}}J*{R}*K@B"
70 PRINT" { 10 {SPACES}}J{ 3 _*}K"
```

## Program 1-2. Telephone

```
10 PRINT" { 9 {RIGHT}}{ 13 {P}}"
20 PRINT" { 8 {RIGHT}}N{ 13 {SPACES}}M"
30 PRINT" { 7 {RIGHT}}N{ 3 {SPACES}}U*UI*UI*I
    { 3 {SPACES}}M"
40 PRINT" { 7 {RIGHT}}L{ 2 {P}}@ {A}{X}{Z}*{X}{Z}
    {S} L{ 2 {P}}@ "
50 PRINT" { 7 {RIGHT}}J{ 2 _}KN{X}{ 5 _}{Z}MJ
    { 2 _}K"
60 PRINT" { 10 {RIGHT}}N W J*K W M"
70 PRINT" { 9 {RIGHT}}{N}{ 2 _}{ 5 W}{ 2 _}{H}"
80 PRINT" { 9 {RIGHT}}{N}{ 11 {SPACES}}{H}"
90 PRINT" { 10 {RIGHT}}{Y}JK{ 5 {Y}}JK{Y} "
```

## Program 1-3. Truck

```
10 PRINT" { 7 {RIGHT}}O{ 14 {Y}}P"
20 PRINT" { 7 {RIGHT}}{H}{ 14 {T}}{N}{ 2 {RIGHT}}U
    { 4 _}I"
30 PRINT" { 7 {RIGHT}}{H}{ 14 {T}}{N}{ 2 {RIGHT}}B
    {G} OPB{@}"
40 PRINT" { 7 {RIGHT}}{H}{T}{ 2 {P}}{ 11 {T}}{N}
    { 2 {#}}B{G}-{ 2 {Y}}J*{W}"
50 PRINT" { 7 {RIGHT}}LNJKMT{ 10 {P}}@{ 2 {I}}BL@U
    { 2 _}IB"
60 PRINT" { 8 {RIGHT}}J{ 2 _}K{ 13 {RIGHT}}J{ 2 _}
    KJKJK"
70 PRINT" { 29 {RIGHT}}{ 2 {T}}"
```

While combining characters in this fashion can provide us with fairly detailed graphics, we are limited to combining those shapes that are provided in the standard character set. As a result, even with the variety of graphics available in the 64 character set, there are many shapes that we simply cannot produce.

## Custom Characters

Fortunately, the 64 provides us with the capability of programming our own character set. In so doing, we can create high-resolution characters in nearly any shape we choose. To get an

# Designing Graphic Shapes

idea of how this is done, we need to take a closer look at how the 64 produces characters.

## A Closer Look at Character Memory

**Looking up character patterns.** Earlier, we looked at the data in memory locations 53248 to 57343 and found that by interpreting that data we could look at the character patterns stored there. Remember too that each character has its own unique screen code (refer to Appendix G). The screen code actually indicates the position of those characters in character memory. For example, the letter A has a screen code of 1. This means that it is the second character stored in the character ROM (the screen codes start with 0). Each byte of character memory consists of eight bits, so it takes eight bytes to hold all 64 dots of each character's pattern. So if we want to look at the character pattern of a specific character, all we need to do is look up its character code, multiply that number by eight, and add the result to the beginning location of the character ROM, which is normally 53248. Where is the pattern of A? The screen code is 1. So the pattern begins  $1*8$  bytes into character memory. The ROM character set starts at 53248, so A's pattern starts at location 53256.

```
10 INPUT "ENTER CHARACTER CODE :" ;R
20 FOR M=0 TO 7
30 POKE 56334, 0: POKE 1, 51
40 A = PEEK (R*8+53248+M)
50 POKE 1, 55: POKE 56334, 1
60 G$=""
70 FOR F=0 TO 7
80 IF A/2=INT(A/2) THENG$=" "+G$:GOTO100
90 G$="[RVS] [OFF]"+G$
100 A=INT(A/2):NEXT
110 PRINTG$
120 NEXT
130 GOTO 10
```

This program looks up each character code you enter in the character table and displays its pattern.

Since these patterns are stored in ROM (Read Only Memory), we can't change them, but the 64 can get its character patterns from other locations. If you tell the 64 to get its character patterns from a block of RAM (Read And write Memory), you can program the character patterns yourself.

## Switching Memory Locations

The VIC-II chip (the video display chip that produces the 64's

# Designing Graphic Shapes

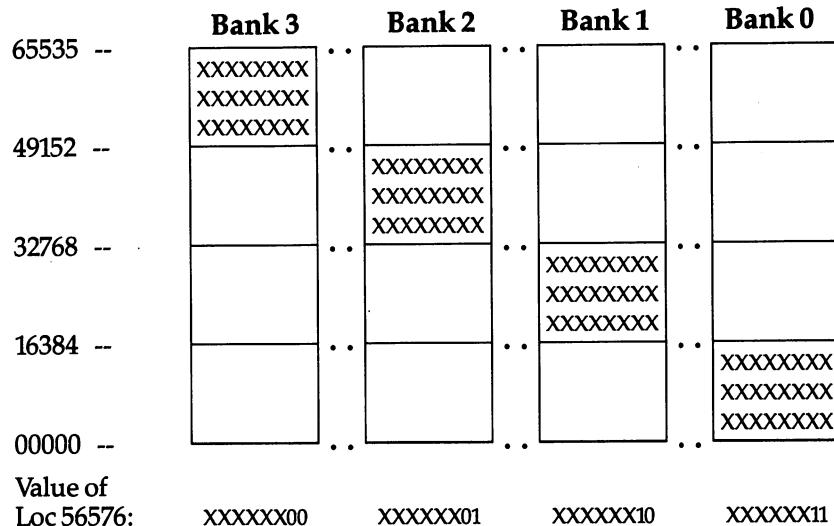
video signals) can access up to 16K (16384) memory locations at one time. Since the 64 has a total of 64K (65536) available memory locations, this means that the VIC-II chip can do all of its work using only one-fourth of the 64's total memory capacity. Therefore the VIC-II chip needs only 14 address lines to access the memory locations it uses. Since each address must consist of 16 bits, however, the other two address lines must come from some other place in the system.

## The Four Video Memory Banks

The 64 gets these other two bits from the CIA (Complex Interface Adapter) register in memory location 56576. The lower two bits in this register become the upper two bits of the VIC-II addresses. They control which of the four possible 16K blocks the VIC-II chip can access.

**Figure 1-5. Switching Video Memory Banks**

### Memory Locations



**Protecting the control bits.** As far as the VIC-II chip is concerned, the upper six bits of CIA register (memory location 56576) are of no importance, so we've marked them with X's in Figure 1-5. In fact, however, they are important to the operation of the

# Designing Graphic Shapes

computer, and when you change the number in the lower two bits of this register to switch banks, you must be careful to avoid changing the rest of the bits in the register.

As it turns out, there are several control registers in the 64 that behave this way. In order to effectively manipulate the values in these registers, we need a method of changing only those bits that need to be changed. The easiest method of accomplishing this is by using the logical operator AND.

**Bit masking** is a process that blocks off certain bits in a control register. We can perform this with the AND operator. When you AND two numbers together, all the bits in the result are 0's, except bits that were 1's in *both* numbers. Therefore, when you AND a known number with an unknown number, any 0 bits in the known number will be 0's in the result, regardless of the values in the unknown number. Any 1 bits in the known number will cause those bits in the result to be exactly what they were in the *unknown* number.

This is like what painters do when they put masking tape along the edge of the area to be painted. The *masked* area won't be painted. You can use AND to mask or protect any bits in an unknown number by putting 1's in the positions you want to protect.

For example, the decimal number 225 is equal to the binary number 11100001. If we wanted to change only the right-hand (lower) four bits of this number, we could AND our first number with the binary number 11110000. This would preserve the left (upper) four bits and set the lower four bits to zero. Table 1-1 shows the values to AND with the number to be modified.

After we have masked the bits we want to preserve, we can add the value of the bits we want to change to the preserved bits.

```
10 INPUT "ENTER BANK (0-3)";B  
20 A=PEEK(56576) AND 252  
30 POKE 56576, A+(3-B)
```

**Moving the screen editor.** After switching banks, the VIC-II chip will read the data from the selected bank. Unfortunately, this process does not move the screen editor, the built-in program that controls screen display. So when you try to enter anything on the screen after switching banks, the screen will display garbage.

Memory location 648 controls the location of the screen editor. It contains the upper eight bits of a 16-bit address. To change the location of the screen editor, you will need to deter-

# Designing Graphic Shapes

**Table 1-1. Binary Masks and Windows**

If you AND with these numbers, all bits marked with X are untouched, while all bits marked with - become zeros. If you OR with these numbers, all bits marked with - are untouched, while all bits marked with X become 1's.

Mask 76543210	Decimal	Window 76543210	Decimal
One-bit Masks and Windows			
-----X	1	XXXXXXX-	254
----X-	2	XXXXXX-X	253
---X--	4	XXXXX-XX	251
--X---	8	XXXX-XXX	247
-X----	16	XXX-XXXX	239
X-----	32	XX-XXXXXX	223
X-----	64	X-XXXXXXX	191
X-----	128	-XXXXXXX	127
Two-bit Masks and Windows			
----XX	3	XXXXXX--	252
---X-	6	XXXXX-X-	249
--XX--	12	XXXX--XX	243
--XX---	24	XXX--XXX	231
--XX----	48	XX--XXXX	207
-XX----	96	X--XXXXX	159
XX-----	192	--XXXXXX	63
Three-bit Masks and Windows			
----XXX	7	XXXXX---	248
---XXX-	14	XXXX--X	241
--XXX--	28	XXX--XX	227
--XXX---	56	XX--XXX	199
-XXX----	112	X--XXXX	143
XXX-----	224	---XXXXX	31
Four-bit Masks and Windows			
----XXXX	15	XXXX---	240
---XXXX-	30	XXX---X	225
--XXXX--	60	XX---XX	195
--XXXX---	120	X---XXX	135
XXXX---	240	----XXXX	15
Five-bit Masks and Windows			
----XXXXX	31	XXX---	224
---XXXXX-	62	XX---X	193
--XXXXX--	124	X---XX	131
XXXXX---	248	----XXX	7
Six-bit Masks and Windows			
--XXXXXX	63	XX----	192
-XXXXXX-	126	X-----X	129
XXXXXX--	252	-----XX	3
Seven-bit Masks and Windows			
--XXXXXXXX	127	X-----	128
XXXXXXXX-	254	-----X	1

Masks are used to isolate particular bits or groups of bits. To test whether bits 2 and 3 of a number are off, ignoring the rest of the byte, use the two-bit mask 12: IF (PEEK(ADDRESS)AND 12)>0 THEN 500.

Windows are used to erase particular bits or groups of bits while preserving the rest of the number. To erase bits 2 and 3, use the two-bit window 243: POKE ADDRESS, PEEK(ADDRESS)AND 243

# Designing Graphic Shapes

mine the value to POKE into location 648. Add this line to the routine above:

```
40 POKE648, ((PEEK(53272) AND240)/4)+((3-(PEEK(56576) AND3))*64)
```

**Finding the characters.** When you run this routine, you may notice that the computer operates normally in banks 0 and 2, but if you switch to banks 1 or 3, the characters turn into a set of random patterns. This is because in banks 0 and 2 the character ROM can be read by the VIC-II chip, but in banks 1 and 3 the ROM is unavailable. In those banks, there is RAM where the character ROM was.

## Putting Characters Where You Want Them

In order to get a character set in banks 1 or 3, you will need to copy the characters from the character ROM or create your own. For now, it will be simpler if we just copy the existing character set.

### Where do we get the characters and where do they go?

Initially, the VIC-II looks for the character set to start 4096 bytes from the start of the bank it is in. This means that we need to copy the characters into memory 4096 bytes into the bank we select. For example, bank 2 begins at memory location 32768; therefore, the character set should be copied into memory starting with memory location 36864.

**Copying the character ROM.** The built-in character ROM begins at location 53248. As was mentioned earlier, to read the contents of this ROM, you need to switch it into the microprocessor's memory space, read the ROM, and switch it back out again so the computer will be able to operate normally. The next program switches the VIC-II chip to bank 1 and copies the entire character ROM into that bank. The actual copying is simple — just PEEK each byte in ROM and POKE it into RAM in the same order.

```
10 B=1
20 A=PEEK(56576) AND 252
30 POKE 56576, A+(3-B)
40 POKE 648, 68
50 FOR R=53248 TO 57343
60 POKE 56334, 0: POKE 1, 51
70 A=PEEK(R)
80 POKE 1, 55: POKE 56334, 1
90 POKE R-32768,A:NEXT
```

## Programming Characters

With the character set transferred to RAM, you can begin creating

# Designing Graphic Shapes

your own custom characters. The character set resides between 20480 and 24575. By changing the values in these locations, you can alter the appearance of the displayed characters. For example, you can underline every character in the character set.

```
FOR R=20487 TO 20487+4095 STEP 8: POKE R,255:  
NEXT
```

**Manipulating the bits.** By changing the bits in character memory, you can create nearly any shape you want. To see how this can be done, let's make a character template in which we can design custom characters.

**Figure 1-6. Sample Custom Character Grid**

Memory Location	128	64	32	16	8	4	2	1	Decimal Value
20624									54
20625		■						■	65
20626									65
20627		■						■	65
20628			■				■		34
20629				■		■			20
20630					■				8
20631									0

Figure 1-6 is an 8-by-8 grid, with darkened squares corresponding to the lighted dots in the custom character. The numbers on the right are obtained by adding the bit values (shown at the top of the grid) of the darkened squares. These values are then POKEd into the memory locations on the left of the grid.

To determine the memory locations for any character, look up the screen code for that character in Appendices G, I, or J. Then multiply the screen code value by 8 and add it to the start of the custom character RAM (in this case, 20480).

# Designing Graphic Shapes

As an example, we can modify the letter R. Its screen code is 18. Our formula is:

$$18*8 + 20480 = 20624$$

If we POKE the custom character values into locations 20624 through 20631, then all the R's on the screen will become outlined hearts.

## A Custom Character Editor

By using this method, any character can be modified, but the process is slow, and the character set will reside right in the middle of the RAM space.

It would be preferable to move the screen, characters, and any other custom video functions to a place in memory that will not get in the way of BASIC. There is such a location up near the top of memory at locations 49152 through 53247.

To help you create your own characters, here is a custom character editor. It provides a number of character editing functions that can make the process of creating custom characters much simpler, such as a copy function, an invert function, and a means of SAVEing and LOADing whole sets of custom characters.

## Instructions

1. Type in the program and SAVE it on disk or cassette tape.
2. When you are finished, type RUN and be prepared to wait about 20 seconds while the character set loads into memory.

## The Screen Display

When the program has finished initializing itself, it will display the editor screen. The screen is divided into five distinct areas (see Figure 1-7).

**The character set display area.** This shows the entire 128 character set available for editing.

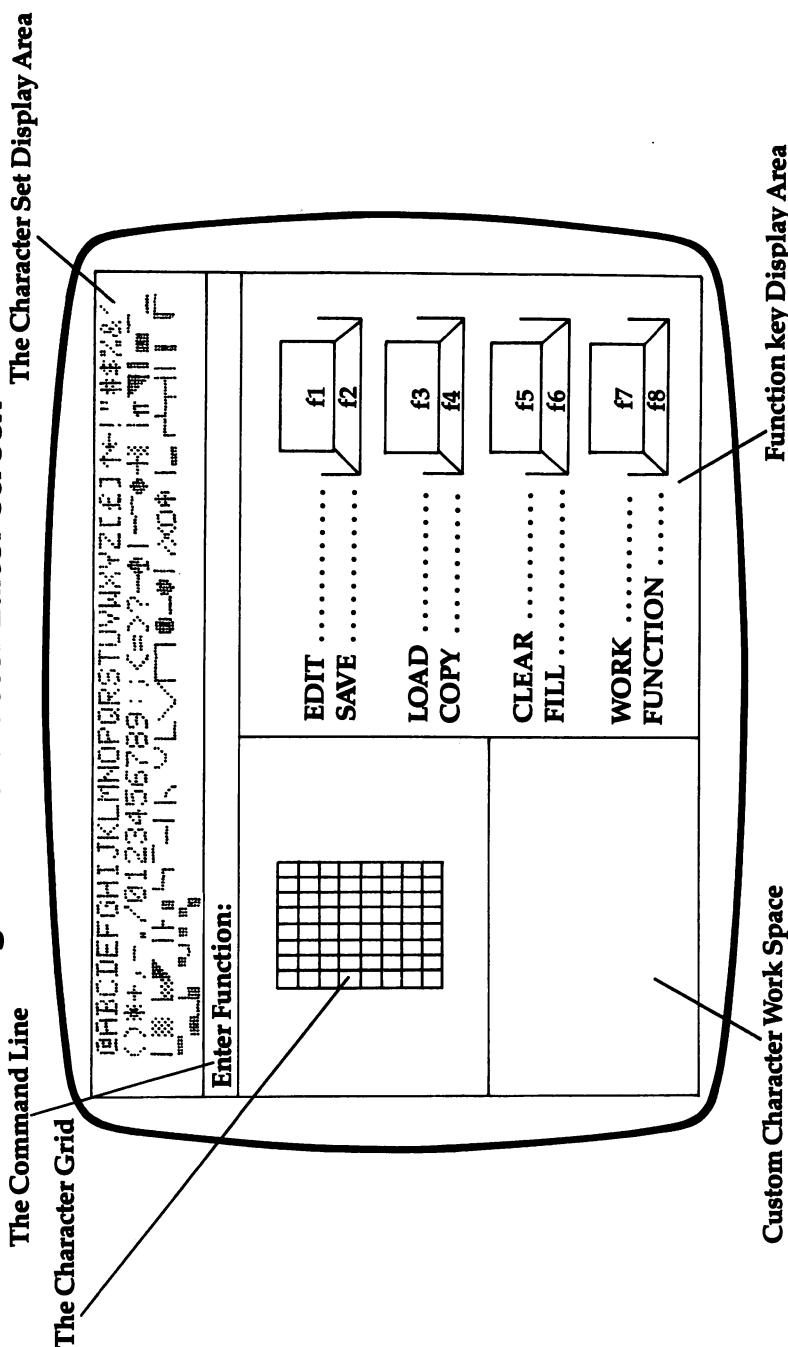
**The command line.** This area prompts the user whenever the system is awaiting a command. It is also used for entering data, such as filenames.

**The character grid.** All of the characters are displayed here. By moving the cursor around in the grid, character shapes are automatically formed and stored.

**The function key display area.** This area displays the four function keys and their current functions. By pressing f8 (SHIFT/f7), the functions will change, revealing eight more functions. So the eighth function in both sets is the function switch key. In all, there are 14 different editing functions.

# Designing Graphic Shapes

**Figure 1-7. Character Editor Screen**



# Designing Graphic Shapes

**The custom character work space.** This area is reserved for trying out the new characters in various combinations. It can be accessed by pressing f7 in the first function set.

## The Editing Functions

Here is a brief description of each of the functions:

**Edit.** After you press the EDIT key, a flashing cursor will appear in the character grid. You can move it anywhere within the grid by pressing the cursor up/down and cursor left/right keys exactly as you would normally use them for full screen editing.

Pressing SPACE (the space bar) will toggle the bit under the cursor. This means that a dark square will become light or a light square will become dark.

To exit the edit mode, just press any of the other function keys. You also can leave one character and begin editing another by simply pressing that character's key on the keyboard.

**Save.** Pressing the SAVE key generates the prompt:

**SAVE ON CASSETTE OR DISK? (C/D):**

Pressing C sets up the computer to save on the cassette unit.

Pressing D sets it up for the disk drive.

The next prompt displayed asks for the filename. The name is limited to six characters and a file type of .CHR. This will make all the character sets easily identifiable on a disk. Entering the sixth letter of the filename initiates the save process. When the save is complete, the ENTER FUNCTION prompt will return on the command line.

**Load.** The LOAD function uses the same process as the SAVE function. It allows for operation with disk or cassette. This operation requests the name of an existing character set and transfers it to memory. That set is then displayed at the top of the screen just as the original set was.

**Copy.** To copy a character, simply press the key that corresponds to the character you wish to replace. That character will then be displayed on the character grid. Then press the COPY key, f4. The command line will ask for the character to copy. Hit the key that corresponds to the character you wish to be copied into the grid, and the old character will be replaced with the new one.

**Clear.** This function turns off all the bits in a character, allowing you to create a character using a "fresh scratch pad."

**Fill.** This is the opposite of the CLEAR function. It turns on

# Designing Graphic Shapes

all the bits in a character. This is very handy for creating characters that are mostly filled-in.

**Work.** The work area is set aside to let you look at the new characters you create in various combinations. In this mode, the cursor control keys not only move the cursor, but also erase whatever they pass over, providing you with a bidirectional delete function.

**Function.** This changes the action of the function keys. It allows access to the other seven functions, for a total of 14 functions using only eight keys.

**Reverse.** This function toggles every bit in the character, turning off the bits that were on and turning on the bits that were off. In effect, it creates a reverse version of the original character.

**Invert.** Invert turns a character upside down.

**Flip.** Flip turns a character over from left to right.

**Scroll.** The scroll functions rotate all of the bits in the character up, down, left, or right by one row or column. Bits falling off the edge will wrap to the other side. As a result, eight scrolls in the same direction will result in the same character you started with.

## Program 1-4. The Custom Character Editor

```
10 REM *** CHANGE SCREEN POINTERS ***
20 REM
21 PRINT CHR$(8);
30 POKE 56578, PEEK(56578) OR 3
40 POKE 56576, (PEEK(56576) AND 252) OR 0
50 POKE 53272, (PEEK(53272) AND 240) OR 2
60 POKE 648, 196: FF=0
70 REM
80 REM *** TRANSFER CHARACTER SET ***
90 REM
100 POKE 56334, PEEK(56334) AND 254
110 POKE 1, PEEK(1) AND 251
120 FOR R=53248 TO 55296
130 POKE R-2048, PEEK(R): NEXT
140 POKE 1, PEEK(1) OR 4
150 POKE 56334, PEEK(56334) OR 1
160 REM
170 REM *** DEFINE CHARACTER GRID ***
175 LZ$(0)="O": LZ$(1)="{RVS} {OFF}"
180 REM
190 CG$ = "[HOME]{ 7 {DOWN}}{ 6 {RIGHT}}"
195 CF$ = "[HOME]{ 7 {DOWN}}{ 6 {RIGHT}}"
200 FOR R=0 TO 7
210 CG$=CG$+"{ 8 O}{G}{DOWN}{ 9 {LEFT}}"
```

# Designing Graphic Shapes

```
215 CF$=CF$+"{RVS}{ 8 {SPACES}}{OFF}{G}{DOWN}
   { 9 {LEFT}}": NEXT
220 CG$ = CG$ + "[ 8 {T}]"
225 CF$ = CF$ + "[ 8 {T}]"
230 REM
240 REM ** DEFINE CHARACTER DISPLAY **
250 REM
260 CD$ = "@ABCDEFGHIJKLMNPQRSTUVWXYZ"
270 CD$ = CD$ + "[f]↑← !" + CHR$(34)
280 CD$ = CD$ + "#$%&'()*,-./"
290 CD$ = CD$ + "0123456789:;=>?*ABCD"
300 CD$ = CD$ + "Z+{-}={↑}{*}{SPACE}{K}{I}{T}{@}
   {G}{+}{M}{f}{N}{Q}{D}{Z}{S}"
310 CD$ = CD$ + "EFGHIJKLMNOPQRSTUVWXYZ"
320 CD$ = CD$ + "{P}{A}{E}{R}{W}{H}{J}{L}{Y}{U}
   {O}@{F}{C}{X}{V}{B}"
330 REM
340 REM *** DEFINE FUNCTION KEYS ***
350 REM
351 FOR R=1TO30:S$=S$+"{RIGHT)":NEXT
360 FOR N=1TO8 STEP2
370 K$(N)="{ 2 {RIGHT}}{ 5 {I}}{DOWN}{ 5 {LEFT}}
   {RVS} F"+STR$(N)+" {DOWN}{ 7 {LEFT}}{OFF}{M}
   {RVS}{ 5 {I}}{OFF} {G}"
380 K$(N+1)="{M}{RVS} F"+STR$(N+1)+" {*}{OFF}{G}"
390 P$(N)="{ 2 {RIGHT}}{A}{ 3 *}{S}{DOWN}
   { 5 {LEFT}}BF"+STR$(N)+"B{DOWN}{ 7 {LEFT}}
   {OFF}{M} {Z}{ 3 *}{X}{OFF} {G}"
400 P$(N+1)="{M}N F"+STR$(N+1)+" M{OFF}{G}{DOWN}
   { 8 {LEFT}}{ 7 {T}}{RIGHT}{UP)": NEXT
405 REM
410 REM *** DEFINE KEY POSITIONS ***
415 REM
420 FOR R=1TO8:K$(R)="{ 7 {DOWN}}{ 31 {RIGHT}}"+K
   $(R):NEXT
430 FOR R=1TO7 STEP2: K$(R)="{HOME}"+SP$+K$(R):SP
   $=SP$+"{ 4 {DOWN}}":NEXT
440 SP$="{ 3 {DOWN}}":FOR R=2TO8 STEP2: K$(R)="
   {HOME}"+SP$+K$(R):SP$=SP$+"{ 4 {DOWN}}":NEXT
   :SP$=""
450 FOR R=1TO8:P$(R)="{ 7 {DOWN}}{ 31 {RIGHT}}"+P
   $(R):NEXT
460 FOR R=1TO7 STEP2: P$(R)="{HOME}"+SP$+P$(R):SP
   $=SP$+"{ 4 {DOWN}}":NEXT
470 SP$="{ 3 {DOWN}}":FOR R=2TO8 STEP2: P$(R)="
   {HOME}"+SP$+P$(R):SP$=SP$+"{ 4 {DOWN}}":NEXT
480 REM
490 REM ***** DEFINE MESSAGES *****
500 REM
```

# Designing Graphic Shapes

```
510 M$(1) = "[HOME]{ 8 {DOWN}}{ 22 {RIGHT}}EDIT....  
    { 2 {DOWN}}{ 8 {LEFT}}SAVE....{ 2 {DOWN}}  
    { 8 {LEFT}}"  
520 M$(1)=M$(1)+"LOAD....{ 2 {DOWN}}{ 8 {LEFT}}CO  
PY....{ 2 {DOWN}}{ 8 {LEFT}}CLEAR...  
    { 2 {DOWN}}{ 8 {LEFT}}"  
530 M$(1)=M$(1)+"FILL....{ 2 {DOWN}}{ 8 {LEFT}}WO  
RK....{ 2 {DOWN}}{ 8 {LEFT}}FUNCTION"  
540 M$(2) = "[HOME]{ 8 {DOWN}}{ 22 {RIGHT}}REVERSE.  
    { 2 {DOWN}}{ 8 {LEFT}}INVERT..{ 2 {DOWN}}  
    { 8 {LEFT}}"  
550 M$(2)=M$(2)+"FLIP....{ 2 {DOWN}}{ 8 {LEFT}}SC  
ROLL R{ 2 {DOWN}}{ 8 {LEFT}}SCROLL L  
    { 2 {DOWN}}{ 8 {LEFT}}"  
560 M$(2)=M$(2)+"SCROLL U{ 2 {DOWN}}{ 8 {LEFT}}SC  
ROLL D{ 2 {DOWN}}{ 8 {LEFT}}FUNCTION"  
570 REM  
580 REM ***** DEFINE RULER LINES *****  
590 REM  
600 LS = "[HOME]{ 4 {DOWN}}{ 40 *}{DOWN}"  
610 LS=LS+"{ 20 *}{R}{ 19 *}{RIGHT}"  
620 LS=LS+"{ 19 {RIGHT}}-{DOWN}{LEFT}-{DOWN}  
    {LEFT}-{DOWN}{LEFT}-{DOWN}{LEFT}-{DOWN}  
    {LEFT}-{DOWN}{LEFT}-{DOWN}{LEFT}-{DOWN}  
    {LEFT}-{DOWN}{LEFT}{{W}{DOWN}{LEFT}}-{DOWN}  
    {LEFT}-{DOWN}{LEFT}-{DOWN}{LEFT}-{DOWN}  
630 LS=LS+"{DOWN}{LEFT}-{DOWN}{LEFT}-{DOWN}{LEFT}-  
    {DOWN}{LEFT}-{ 8 {UP}}{ 21 {LEFT}}{ 20 *}{HOME}"  
900 REM  
910 REM ***** DISPLAY EDIT SCREEN *****  
920 REM  
930 PRINT"(CLR)";CD$:PRINTCG$:PRINTL$:PRINTM$(1)  
940 REM  
941 REM *** EDIT CHARACTERS ***  
942 REM  
945 A$="@"  
1000 REM  
1010 REM *** DISPLAY FUNCTION KEYS ***  
1020 REM  
1025 POKE 55753, 14  
1030 PRINT"[HOME]";:FOR R=1TO8: PRINTK$(R);: NEXT  
    : PRINT"[HOME]";  
1040 A(0)=1:A(1)=3:A(2)=5:A(3)=7:A(4)=2:A(5)=4:A(6)=6:A(7)=8  
1045 GOTO 1087  
1050 REM  
1060 REM *** GET KEYBOARD ENTRY ***  
1070 REM  
1080 PRINT"[HOME]{ 5 {DOWN}}ENTER FUNCTION:  
    { 23 {SPACES}}"
```

# Designing Graphic Shapes

```
1082 GETA$:IFA$=""THEN1080
1084 REM
1085 REM --- SKIP INVALID KEYS ---
1086 REM
1087 VA=ASC(A$)
1088 IF (VA<32 OR VA>223) THEN 1080
1089 IF (VA>95 AND VA<133) THEN 1080
1090 IF (VA>140 AND VA<161) THEN 1080
1091 IF (VA>140 OR VA<132) THEN1205
1100 REM
1110 REM --- GET FUNCTION KEYS ---
1120 REM
1130 PRINTP$(A(ASC(A$)-133))
1140 FORR=0TO99:NEXT
1150 PRINTK$(A(ASC(A$)-133));"{HOME}";
1160 GOTO 1390
1180 REM
1190 REM - CONVERT ASC TO SCREEN CODE -
1200 REM
1205 LT=VA
1210 IF VA>31 AND VA<64 THEN SC=VA: GOTO 1240
1220 IF VA>63 AND VA<193 THEN SC=VA-64: GOTO 1240
1230 IF VA>191 AND VA<224 THEN SC=VA-128: GOTO 1240
1240 REM
1250 REM -- DISPLAY EDITED CHARACTER --
1260 REM
1262 LT$=STR$(LT):LT$=RIGHT$(LT$,LEN(LT$)-1)
1265 PRINT"[HOME]{ 5 {DOWN}}ENTRY MODE: CHR$(",LT
$;"){ 17 {SPACES}}"
1270 POKE50633, SC
1273 PRINT"[HOME]{ 7 {DOWN}}{ 6 {RIGHT}}";
1275 PRINT"[HOME]{ 5 {DOWN}}ENTRY MODE: CHR$(",LT
$;"){ 17 {SPACES}}"
1276 POKE50633, SC
1277 PRINT"[HOME]{ 7 {DOWN}}{ 6 {RIGHT}}";
1278 FOR R=0 TO 7
1279 RR=PEEK(R+(51200+8*SC))
1280 PRINTLZ$(ABS((RRAND128)=128));LZ$(ABS((RRAND
64)=64));LZ$(ABS((RRAND32)=32));
1281 PRINTLZ$(ABS((RRAND16)=16));LZ$(ABS((RRAND8)
=8));LZ$(ABS((RRAND4)=4));
1282 PRINTLZ$(ABS((RRAND2)=2));LZ$(ABS((RRAND1)=1
));"{DOWN}{ 8 {LEFT}}";:NEXT:GOTO 1080
1350 GOTO 1080
1360 REM
1370 REM -- SPECIAL FUNCTION ROUTINES--
1380 REM
1390 IF FF=1 THEN 1500
1400 ON A(ASC(A$)-133) GOTO 2005,1433,1443,1453,1
462,1472,1483,1493
```

# Designing Graphic Shapes

```
1410 GOTO 1170
1430 REM
1431 REM --- SAVE A CHARACTER SET ---
1432 REM
1433 PRINT" {HOME}{ 5 {DOWN}}SAVE ON CASSETTE OR D
ISK? (C/D):{ 6 {SPACES}}"
1434 GET Q$ : IF Q$ = "THEN1434"
1435 IF Q$ <> "C" AND Q$ <> "D" THEN 1087
1436 GOTO 1760
1440 REM
1441 REM --- LOAD A CHARACTER SET ---
1442 REM
1443 PRINT" {HOME}{ 5 {DOWN}}LOAD FROM CASSETTE OR
DISK? (C/D):{ 4 {SPACES}}"
1444 GET Q$ : IF Q$ = "THEN1444"
1445 IF Q$ <> "C" AND Q$ <> "D" THEN 1087
1446 GOTO 1910
1450 REM
1451 REM --- COPY A CHARACTER ---
1452 REM
1453 PRINT" {HOME}{ 5 {DOWN}}ENTER CHARACTER TO CO
PY:{ 14 {SPACES}}"
1454 GET CA$ : IF CA$ = "" THEN 1454
1455 GOTO 1605
1456 IF CA$ = "" THEN 1454
1459 GOTO 1275
1460 REM
1461 REM ----- CLEAR A CHARACTER -----
1462 PRINT" {HOME}{ 5 {DOWN}}CLEAR CHAR: CHR$( ";"LT
$; ){ 17 {SPACES}}"
1463 H=51200+8*SC
1464 FOR R=H TO H+7: POKE R,0: NEXT
1465 PRINTCG$:GOTO 1080
1470 REM
1471 REM ----- FILL A CHARACTER -----
1472 PRINT" {HOME}{ 5 {DOWN}}FILL{ 2 {SPACES}}CHAR
: CHR$( ";"LT$; ){ 17 {SPACES}}"
1473 H=51200+8*SC
1474 FOR R=H TO H+7: POKE R,255: NEXT
1475 PRINTCF$:GOTO 1080
1480 REM
1481 REM ---- GOTO WORK SPACE ---
1482 REM
1483 PRINT" {HOME}{ 5 {DOWN}}ENABLE WORK SPACE
{ 22 {SPACES}}"
1484 GOTO 2705
1490 REM
1491 REM --- SWITCH FUNCTION SET ---
1492 REM
1493 FF=1: PRINTM$(2):GOTO 1080
```

# Designing Graphic Shapes

```
1499 RETURN
1500 ON A(ASC(A$)-133) GOTO 1523,1532,1542,1551,1
      561,1572,1582,1593
1510 GOTO 1170
1520 REM
1521 REM --- REVERSE CHARACTER BITS---
1522 REM
1523 PRINT" {HOME}{ 5 {DOWN}}REVERSE CHARACTER:
      { 20 {SPACES}}"
1524 H=51200+8*SC
1525 FOR R=H TO H+7: POKE R,255-PEEK(R): NEXT
1529 GOTO 1275
1530 REM
1531 REM --- INVERT CHARACTER BITS -----
1532 PRINT" {HOME}{ 5 {DOWN}}INVERTING CHARACTER:
      { 18 {SPACES}}"
1533 H=51200+8*SC
1534 FOR R=H TO H+7: T(R-H)=PEEK(R): NEXT
1535 FOR R=H TO H+7: POKE R,T(7-(R-H)): NEXT:GOTO
      1275
1540 REM
1541 REM ----- FLIP CHARACTER BITS -----
1542 PRINT" {HOME}{ 5 {DOWN}}FLIPPING CHARACTER:
      { 19 {SPACES}}"
1543 FOR U=51200+8*SC TO (51200+8*SC)+7
1544 Z=PEEK(U)
1545 R=128*(ABS((ZAND1)=1))+64*(ABS((ZAND2)=2))+3
      2*(ABS((ZAND4)=4))
1546 R=R+16*(ABS((ZAND8)=8))+8*(ABS((ZAND16)=16))
      +4*(ABS((ZAND32)=32))
1547 R=R+2*(ABS((ZAND64)=64))+1*(ABS((ZAND128)=12
      8))
1548 POKE U,R
1549 NEXT: GOTO1275
1550 REM ----- SCROLL RIGHT -----
1551 PRINT" {HOME}{ 5 {DOWN}}SCROLLING RIGHT:
      { 22 {SPACES}}"
1552 FOR U=51200+8*SC TO (51200+8*SC)+7
1553 R=(PEEK(U)/2)
1554 IF PEEK(U)/2<>INT(PEEK(U)/2) THEN R=R+128
1555 POKE U,R
1556 NEXT: GOTO1275
1560 REM ----- SCROLL LEFT -----
1561 PRINT" {HOME}{ 5 {DOWN}}SCROLLING LEFT:
      { 23 {SPACES}}"
1562 FOR U=51200+8*SC TO (51200+8*SC)+7
1563 R=(PEEK(U)*2)
1564 IF PEEK(U)=>128 THEN R=R+1
1565 IF R>255 THEN R=R-256
1566 POKE U,R
```

# Designing Graphic Shapes

```
1567 NEXT: GOTO1275
1570 REM
1571 REM ----- SCROLL UP -----
1572 PRINT" {HOME}{ 5 {DOWN}}SCROLLING UP:
{ 25 {SPACES}}"
1573 FOR R=0TO7
1574 Y(R)=PEEK(R+51200+8*SC):NEXT
1575 FOR R=0TO6
1576 POKE R+51200+8*SC, Y(R+1): NEXT
1577 POKE 51207+8*SC, Y(0)
1579 GOTO 1275
1580 REM
1581 REM ----- SCROLL DOWN -----
1582 PRINT" {HOME}{ 5 {DOWN}}SCROLLING DOWN:
{ 23 {SPACES}}"
1583 FOR R=0TO7
1584 Y(R)=PEEK(R+51200+8*SC):NEXT
1585 FOR R=1TO7
1586 POKE R+51200+8*SC, Y(R-1): NEXT
1587 POKE 51200+8*SC, Y(7)
1589 GOTO 1275
1590 REM
1591 REM --- SWITCH FUNCTION SET ---
1592 REM
1593 FF=0: PRINTM$(1):GOTO 1080
1599 RETURN
1600 REM
1601 REM *** COPY CHARACTER ROUTINE ***
1602 REM
1605 DA=ASC(CA$)
1610 IF (DA<32 OR DA>223) THEN 1456
1620 IF (DA>95 AND DA<133) THEN 1456
1630 IF (DA<141 AND DA>132) THEN A$=CA$:GOTO 1080
1640 IF (DA>140 AND DA<161) THEN RETURN
1650 PRINT" {HOME}{ 5 {DOWN}}{ 25 {RIGHT}}";CA$
1660 IF DA>31 AND DA<64 THEN SD=DA: GOTO 1690
1670 IF DA>63 AND DA<193 THEN SD=DA-64: GOTO 1690
1680 IF DA>191 AND DA<224 THEN SD=DA-128: GOTO 16
90
1690 VJ=51200+8*SD:JJ=51200+8*SC
1700 FOR R=0 TO 7
1710 POKE JJ+R,PEEK(VJ+R): NEXT
1720 GOTO 1456
1730 REM
1740 REM *** SAVE CHAR SET ROUTINE ***
1750 REM
1760 PRINT" {HOME}{ 5 {DOWN}}SAVE: FILE NAME -----
-.CHR{ 8 {SPACES}}{HOME}{ 5 {DOWN}}
{ 16 {RIGHT}}";
1770 LL=0:NM$=""
```

# Designing Graphic Shapes

```
1775 FOR R=0TO30: PRINT"{}{OFF}{LEFT}";
1780 GET A$:IFA$=""THEN NEXT
1790 IFA$<>""THEN 1840
1800 FOR R=0TO30: PRINT"-{LEFT}";
1810 GET A$:IFA$=""THEN NEXT
1820 IFA$<>""THEN 1840
1830 GOTO 1775
1840 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$=CHR$(34)ORA$="{UP}"THEN1775
1845 IFA$="{DOWN}"ORA$="{RIGHT}"ORA$="{LEFT}"THEN 1775
1847 IF ASC(A$)>132 AND ASC(A$)<141 THEN 1080
1850 PRINTA$;
1855 NM$=NM$+A$:LL=LL+1:IFLL=6THEN1870
1860 GOTO 1775
1870 NM$=NM$+".CHR"
1875 IF Q$="C"THEN1890
1880 OPEN 1,8,4,NM$+",W"
1885 FOR R=51200 TO 52224
1887 PRINT#1,PEEK(R):NEXT
1889 CLOSE1:GOTO 1275
1890 OPEN 1,1,1,NM$:GOTO 1885
1900 REM
1901 REM *** SAVE CHAR SET ROUTINE ***
1902 REM
1910 PRINT"{}{5 {DOWN}}LOAD: FILE NAME -----
 -.{CHR{8 {SPACES}}}{HOME}{5 {DOWN}}
 {16 {RIGHT}}";
1915 LL=0:NM$=""
1920 FOR R=0TO30: PRINT"{}{OFF}{LEFT}";
1925 GET A$:IFA$=""THEN NEXT
1930 IFA$<>""THEN 1955
1935 FOR R=0TO30: PRINT"-{LEFT}";
1940 GET A$:IFA$=""THEN NEXT
1945 IFA$<>""THEN 1955
1950 GOTO 1920
1955 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$=CHR$(34)ORA$="{UP}"THEN1920
1960 IFA$="{DOWN}"ORA$="{RIGHT}"ORA$="{LEFT}"THEN 1920
1965 IF ASC(A$)>132 AND ASC(A$)<141 THEN 1080
1970 PRINTA$;
1975 NM$=NM$+A$:LL=LL+1:IFLL=6THEN1980
1976 GOTO 1920
1980 NM$=NM$+".CHR"
1985 IF Q$="C"THEN1999
1987 OPEN 1,8,4,NM$+",R"
1988 FOR R=51200 TO 52224
1989 INPUT#1,VL:POKE R,VL:NEXT
```

# Designing Graphic Shapes

```
1990 CLOSE1:GOTO 1275
1999 OPEN 1,1,1,NM$:GOTO 1885
2000 REM
2001 REM ** CHARACTER EDIT ROUTINE **
2002 REM
2005 PRINT"{{HOME}{ 5 {DOWN}}EDIT{ 2 {SPACES}}}MODE
: CHR$(";LT$;"){ 17 {SPACES}}"
2010 PRINT"{{HOME}{ 7 {DOWN}}{ 6 {RIGHT}}}";
2015 LO=50462: SV=PEEK(LO) :SM=(51200+(8*SC)): EX
=7 :GT=SC
2020 FOR R=0TO30: POKE LO,SV
2030 GET A$:IFA$=""THEN NEXT
2040 IFA$<>"THEN 2090
2050 FOR R=0TO30: POKE LO,102
2060 GET A$:IFA$=""THEN NEXT
2070 IFA$<>"THEN 2090
2080 GOTO 2020
2090 IFA$="{UP}" THEN 2200
2091 IFA$="{DOWN}" THEN 2100
2092 IFA$="{LEFT}" THEN 2300
2093 IFA$="{RIGHT}" THEN 2400
2094 IF A$=CHR$(32) AND SV=160 THEN 2500
2095 IF A$=CHR$(32) AND SV=79 THEN 2600
2099 POKE LO,SV: GOTO 1087
2100 POKE LO,102:FORR=0TO40:NEXT
2110 IF GT=SC+7 THEN 2020
2120 POKE LO,SV: LO=LO+40: GT=GT+1: SV=PEEK (LO):
:SM=SM+1: GOTO 2020
2200 POKE LO,102:FORR=0TO40:NEXT
2210 IF GT=SC THEN2020
2220 POKE LO,SV: LO=LO-40: GT=GT-1: SV=PEEK(LO):
:SM=SM-1: GOTO 2020
2300 POKE LO,102:FORR=0TO40:NEXT
2310 IF EX=7 THEN 2020
2320 POKE LO,SV: LO=LO-1: EX=EX+1: SV=PEEK(LO):
GOTO 2020
2400 POKE LO,102:FORR=0TO40:NEXT
2410 IF EX=0 THEN 2020
2420 POKE LO,SV: LO=LO+1: EX=EX-1: SV=PEEK(LO):
GOTO 2020
2500 POKE LO, 79: POKE SM,PEEK(SM)-(2↑EX):SV=79:
GOTO2020
2600 POKE LO, 160: POKE SM,PEEK(SM)+(2↑EX):SV=160
:GOTO2020
2700 REM
2701 REM *** WORK SPACE ROUTINE ***
2702 REM
2705 HZ=0:LZ=0
2710 PRINT"{{HOME}{ 17 {DOWN}}};
```

# Designing Graphic Shapes

```
2720 FOR R=0TO30: PRINT" {RVS} {OFF}{LEFT}";  
2730 GET A$:IFA$=""THEN NEXT  
2740 IFA$<>"THEN 2790  
2750 FOR R=0TO30: PRINT" {LEFT}";  
2760 GET A$:IFA$=""THEN NEXT  
2770 IFA$<>"THEN 2790  
2780 GOTO 2720  
2790 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$  
=CHR$(34)THEN2720  
2810 IF ASC(A$)>132 AND ASC(A$)<141 THENPRINT" ";  
:GOTO 1087  
2820 IFA$={"UP"} THEN 3200  
2830 IFA$={"DOWN"} THEN 3100  
2840 IFA$={"LEFT"} THEN 3300  
2850 IFA$={"RIGHT"} THEN 3400  
2860 IF HZ<18 AND LZ<6 THEN PRINTA$;:HZ=HZ+1:GOTO  
2720  
2870 IF HZ=18 AND LZ<6 THEN 2720  
2880 IF HZ=18 AND LZ=6 THEN 2720  
3100 IF LZ=6 THEN2720  
3110 PRINT" {LEFT}{DOWN}";:LZ=LZ+1:GOTO 2720  
3200 IF LZ=0 THEN2720  
3210 PRINT" {LEFT}{UP}";:LZ=LZ-1:GOTO 2720  
3300 IF HZ=0 THEN2720  
3310 PRINT" {2 {LEFT}}";:HZ=HZ-1:GOTO 2720  
3400 IF HZ=18 THEN2720  
3410 PRINT" {LEFT}{RIGHT}";:HZ=HZ+1:GOTO 2720
```

## Sprites

In addition to custom characters, the 64 can also display eight special characters called sprites. Sprites are nearly eight times the size of standard characters, and they have a number of special features that make them far more versatile than custom characters.

### Sprite Features

**X,Y addressing.** Sprites can be positioned anywhere on the screen by entering the desired position using X and Y coordinates.

**Selectable display priority.** Sprites can be displayed so they appear to be either in front or in back of other sprites, the background, or the border.

**Expanded modes.** Sprites can be expanded by a factor of two in the X-direction (horizontally), the Y-direction (vertically), or both.

**Collision detection.** It is possible to determine whether a sprite collision has occurred and which sprites were involved by PEEKING a single memory location.

# Designing Graphic Shapes

## Sprite Control Registers

The sprites' positions, shapes, and colors are determined by the values in the sprite control registers. Table 1-2 shows the function and location of each of the sprite control registers.

**Table 1-2**

Memory Location	Control Function	Description
53248	Sprite #0: X location	Horizontal position of sprite #0
53249	Sprite #0: Y location	Vertical position of sprite #0
53250	Sprite #1: X location	Horizontal position of sprite #1
53251	Sprite #1: Y location	Vertical position of sprite #1
53252	Sprite #2: X location	Horizontal position of sprite #2
53253	Sprite #2: Y location	Vertical position of sprite #2
53254	Sprite #3: X location	Horizontal position of sprite #3
53255	Sprite #3: Y location	Vertical position of sprite #3
53256	Sprite #4: X location	Horizontal position of sprite #4
53257	Sprite #4: Y location	Vertical position of sprite #4
53258	Sprite #5: X location	Horizontal position of sprite #5
53259	Sprite #5: Y location	Vertical position of sprite #5
53260	Sprite #6: X location	Horizontal position of sprite #6
53261	Sprite #6: Y location	Vertical position of sprite #6
53262	Sprite #7: X location	Horizontal position of sprite #7
53263	Sprite #7: Y location	Vertical position of sprite #7
53264	Most significant bit (MSB) of sprite X location	The X value location is a 9-bit number. This allows the sprites to be anywhere on the screen. Each sprite uses one bit in this register.

# Designing Graphic Shapes

<b>Memory Location</b>	<b>Control Function</b>	<b>Description</b>
	Bit $n$	Sprite $n$ (1 = sprite $n$ is in rightmost portion of screen)
53265	VIC-II chip control register #1	
	Bits 0-2	Scroll in Y (vertical) direction.
	Bit 3	24/25 row display toggle (1 = 25)
	Bit 4	Screen blanking (0 = blank)
	Bit 5	Bitmap mode (1 = on)
	Bit 6	Extended color mode (1 = on)
	Bit 7	Raster compare MSB
53266	Raster compare register	Used by machine language routines to specify a screen position. This is a 9-bit number. The most significant bit (MSB) is bit 7 of the control register (53265).
53267	Light pen X-position	This register holds a number that indicates the <i>horizontal</i> position of the light pen.
53268	Light pen Y-position	This register holds a number that indicates the <i>vertical</i> position of the light pen.
53269	Enable sprites	This register has an enable bit for each sprite. Turning that bit <i>on</i> will turn on the sprite.
	Bit $n$	Enable sprite # $n$ (1 = on)
53270	VIC-II chip control register #2	
	Bits 0-2	Scroll in X (horizontal) direction.
	Bit 3	38/40 column display toggle (1 = 40 columns)
	Bit 4	Multicolor mode (1 = multicolor)
53271	Expand sprites (Y-direction)	This register has an expand bit for each sprite. Turning that bit <i>on</i> will double the size of the sprite in the vertical direction.
	Bit $n$	Expand sprite # $n$ (1 = on)
53272	Video display memory control register	Controls the location of the

# Designing Graphic Shapes

<b>Memory Location</b>	<b>Control Function</b>	<b>Description</b>
	Bits 1-3 Bit 3	screen and character memory Character memory location In bitmapped mode, the 8K bitmap location. (0 = low block, 1 = high block)
53273	Bits 4-7 VIC-II interrupt control register Bit 0 Bit 1 Bit 2 Bit 3 Bit 7	Screen memory  Raster interrupt occurred Sprite/background collision Sprite/sprite collision Light pen interrupt Interrupt flag. This bit will be <i>on</i> (1) if any interrupt occurred.
53274	Interrupt enable register	This register permits or prohibits interrupts. (1 = interrupt enabled)
53275	Sprite/background priority	This register controls the sprites' appearance relative to 1 bits and 01, 10, and 11 bit-pairs in the background. If the sprite has priority, then it will appear to be in front of the background, covering up background objects. If the background has priority, then the sprite will be "behind" so background objects will show through and the sprite will be covered.
53276	Bit <i>n</i> Multicolor sprite mode Bit <i>n</i>	(1 = sprite <i>n</i> has priority)  Sprite # <i>n</i> : (1 = on)
53277	Expand sprites (X-direction)  Bit <i>n</i>	This register has a bit for each sprite. Turning that bit <i>on</i> will double the size of the sprite in the horizontal direction. Expand sprite # <i>n</i> (1 = on)
53278	Sprite/sprite collision detection register	This register indicates which sprites have been in a

# Designing Graphic Shapes

<b>Memory Location</b>	<b>Control Function</b>	<b>Description</b>
	Bit <i>n</i>	collision with another sprite. For example, if you detect a collision in the collision interrupt register (53273), you can check this register to determine which sprites were involved.
53279	Sprite/background collision detection register	Sprite # <i>n</i> collision (1 = involved in collision) This register is similar to the sprite/sprite collision register (53278), except that it indicates which sprites have collided with screen data. Note that in multicolor mode, collisions with 01 bit-pairs cannot be detected. Sprite # <i>n</i> collision (1 = involved in collision)
	Bit <i>n</i>	POKE low nybble color values ( <i>see Appendix E</i> ).
53280	Border color control	POKE low nybble color values ( <i>see Appendix E</i> ). This register controls the basic background color in all modes (except one-color bitmapped) for the entire display area. This is the color that shows through wherever a 0 or 00 pixel is called for. Cells and characters cannot be individually controlled for off (0) pixels in one-color modes or 00 bit-pairs in multicolor modes. In extended color mode, this is the background color for characters with a screen code from 0 to 63.
53281	Background color control register #0	
53282	Auxiliary color control register (register #1)	POKE low nybble color values ( <i>see Appendix E</i> ). In

# Designing Graphic Shapes

<b>Memory Location</b>	<b>Control Function</b>	<b>Description</b>
53283	Auxiliary color control register (register #2)	multicolor text mode, this register controls the color displayed at all 01 bit-pairs. In extended color mode, this is the background color for all characters with a screen code from 64 to 127.
53284	Background color control register #3	POKE low nybble color values ( <i>see Appendix E</i> ). In multicolor text mode, this register controls the color displayed at all 10 bit-pairs. In extended color mode, this is the background color for all characters with a screen code from 128 to 191.
53285	Sprite multicolor control register #0	POKE low nybble color values ( <i>see Appendix E</i> ). In extended color mode, this is the color for all characters with a screen code from 192 to 255.
53286	Sprite multicolor control register #1	POKE low nybble color values ( <i>see Appendix E</i> ). Controls the color displayed at all multicolor sprite 01 bit-pairs.
53287- 53294	Sprite color control registers	POKE low nybble color values ( <i>see Appendix E</i> ). Controls the color displayed at all <i>on</i> (1) pixels in one-color sprites and all 11 bit-pairs in multicolor sprites.

# Designing Graphic Shapes

## Displaying the Sprites

Displaying sprites is a three-step process. First, the sprite must be designed, giving it a shape. Second, the sprite must be *enabled*, telling the computer that it exists and where to find it. Third, it must be positioned on the screen.

## Sprite Shapes

When you first turn on the 64, the sprites look much like squares of Swiss cheese. The shape of each sprite is determined by a block of data we'll call a shape table, and until that table is programmed, each sprite will be a mass of disorganized dots.

**Sprite shape tables.** A sprite shape table is very similar to a character pattern. The biggest difference between them is that sprites are made up of 63 bytes and characters are made up of only eight. Figure 1-8 illustrates the layout of a sprite shape table.

Actually, although each sprite uses only 63 bytes (as illustrated in Figure 1-8), its shape table contains a full 64 bytes. The last byte can be used as free RAM. In Chapter 3, we'll discuss how this last byte can be used to simplify the process of making sprites move.

With the total working space for the VIC-II chip being 16384 bytes, the total number of possible locations for the sprite tables is 256 ( $16384/64 = 256$ ).

**Addressing the shape tables (a look at pointers).** The 64 screen memory is a block of memory 1024 bytes long. But the screen only displays 25 rows of 40 characters each. So it only uses 1000 bytes of the total 1024-byte block. Although some of these extra bytes are not used, the last eight bytes of screen memory are always used to point to the locations of the sprite shape tables. For example, when the 64 is first turned on, screen memory begins at location 1024. This means that the end of screen memory will be at location 2047. The last eight bytes, 2040 to 2047, are the pointers that contain the locations of the sprite tables. Location 2040 would contain the pointer for sprite 0, 2041 would contain the pointer for sprite 1, and so on, with 2047 containing the pointer for sprite 7.

**How do pointers work?** When you have told the VIC-II to display sprites, it begins to display the sprite pattern. Since sprite data could be anywhere in the graphics block, the VIC-II needs to find out where to look. So it checks the appropriate pointer. For example, if location 2040 contains the number 25, then the VIC-II will look for the start of sprite 0's shape table at memory location 1600 ( $25*64$ ). Whatever data is in the 63 bytes from 1600 to 1662

# Designing Graphic Shapes

**Figure 1-8. Sprite Memory Map**

BYTE 00	BYTE 01	BYTE 02
BYTE 03	BYTE 04	BYTE 05
BYTE 06	BYTE 07	BYTE 08
BYTE 09	BYTE 10	BYTE 11
BYTE 12	BYTE 13	BYTE 14
BYTE 15	BYTE 16	BYTE 17
BYTE 18	BYTE 19	BYTE 20
BYTE 21	BYTE 22	BYTE 23
BYTE 24	BYTE 25	BYTE 26
BYTE 27	BYTE 28	BYTE 29
BYTE 30	BYTE 31	BYTE 32
BYTE 33	BYTE 34	BYTE 35
BYTE 36	BYTE 37	BYTE 38
BYTE 39	BYTE 40	BYTE 41
BYTE 42	BYTE 43	BYTE 44
BYTE 45	BYTE 46	BYTE 47
BYTE 48	BYTE 49	BYTE 50
BYTE 51	BYTE 52	BYTE 53
BYTE 54	BYTE 55	BYTE 56
BYTE 57	BYTE 58	BYTE 59
BYTE 60	BYTE 61	BYTE 62

will appear on the screen as the shape of sprite 0. If location 2041 contains the number 26, then the VIC-II will look for the start of sprite 1's shape table at location 1664. (You'll notice that location 1663 is not used.)

While this process may seem a bit awkward at first, it actually allows you great flexibility and speed. What if you wanted to *change* a sprite's shape in the middle of a program? Without pointers, you would have to POKE the new pattern into the sprite shape table, byte by byte. It would be very slow, and you'd have to go through the same process every time you wanted to change the shape. With pointers, however, you can set up many shape tables for the same sprite. Then you can change the sprite's shape

# Designing Graphic Shapes

instantly by pointing to a different table. A single POKE is always faster than 63 separate POKEs.

For example, if you set up four shapes at 1600, 1664, 1728, and 1792, then to switch from one shape to the next, you would merely have to POKE 2040 with 25, 26, 27, or 28. The ability to change shapes instantly is a vital part of animation.

## Displaying the Sprites

When power is first applied to the 64, all of the sprites are turned off and positioned behind the screen border. Before you can display sprite 0, you need to *enable* it, or turn it on; you also need to move it onto the screen.

**Moving sprites onto the screen.** To put a sprite onto the screen, POKE its X and Y location registers with numbers between 30 and 230. This command positions sprite 0 on the screen at X,Y location 100,100:

**POKE 53248,100:POKE 53249,100**

**Enabling sprites.** Sprite 0 is now somewhere near the center of the screen, but we can't see it because it hasn't been enabled. To enable the sprite, we need to POKE the sprite enable register (53269) with a value that will set the corresponding bit to 1.

To enable a specific sprite, use the command:

**POKE 53269, PEEK(53269) OR 2↑ (sprite number)**

To enable sprite 0, for example, enter:

**POKE 53269, PEEK(53269) OR 2↑ 0**

To enable more than one sprite, add the values for each individual sprite and use their sum in the formula.

**Disabling a sprite.** To disable a sprite (turn it OFF), use the formula:

**POKE 53269, PEEK(53269) AND NOT 2↑ (sprite number)**

To disable more than one sprite, add the values for each individual sprite and use their sum in the formula.

Here's a routine that enables all eight sprites, positions them across the top of the screen, and assigns them all to the same shape table (35) located between memory locations 2240 and 2303. The shape table is then defined as a solid dark rectangle by POKEing each location in the table with 255.

```
10 FOR R=0 TO 15
20 POKE 53248+R,50: NEXT
30 POKE 53269, 255
```

# Designing Graphic Shapes

```
40 FOR R=0 TO 14 STEP 2  
50 POKE R+53248,30+R*15:NEXT  
60 FOR R=2040 TO 2047  
70 POKE R,35:NEXT  
80 FOR R=2240 TO 2303  
90 POKE R,255:NEXT
```

## Programming Sprite Shapes

**Regular patterns.** Now that we have some sprites displayed on the screen, you can change the values in their shape table and observe the changes in the sprites as we do. For example, you can use this simple loop to change all of the values in the sprites to a repeating pattern of vertical stripes.

```
10 FOR R=2240 TO 2303  
20 POKE R,204: NEXT
```

**Nonrepeating patterns.** Of course, you will usually wish to produce sprites using nonrepeating patterns. This will require some more careful planning. Start with a grid like the one in Figure 1-9.

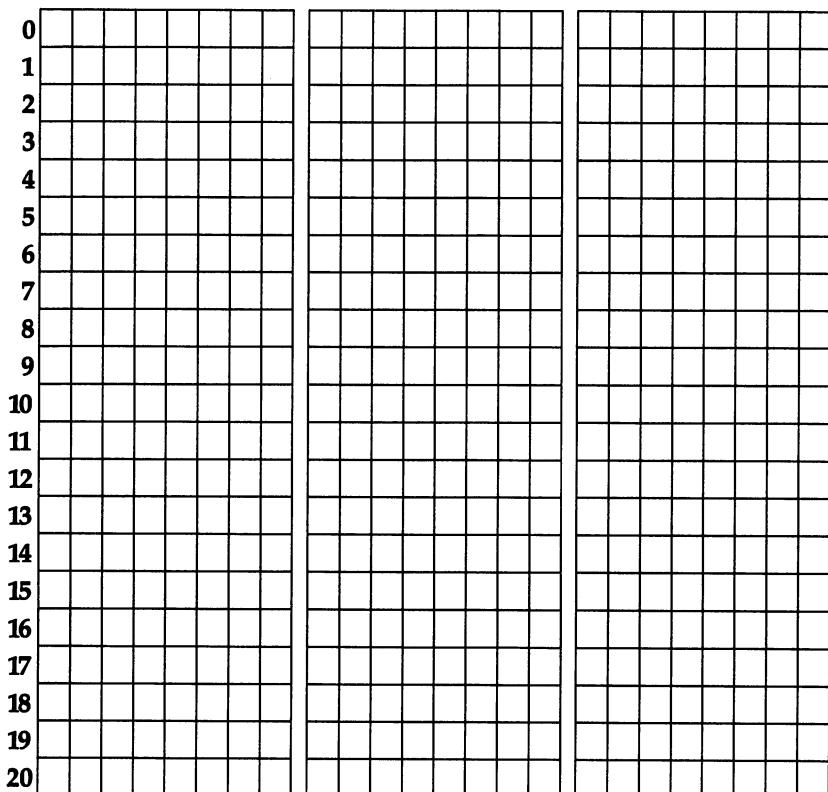
The grid is divided into three columns and 21 rows. The memory is arranged as shown in Figure 1-8, the sprite shape table. Begin designing the shape of your sprite by darkening the individual dots in the grid. As with custom characters, each group of eight bits corresponds to a single byte in memory. The top row of a sprite, from left to right, is contained in the first three bytes of the sprite shape table. The next row is in the next three bytes, and so on.

**Storing the sprite shape.** Once you have a pattern you like, you will need to convert the *on* and *off* bits into decimal numbers, put them in DATA statements, and POKE them into the sprite's shape table. You can, of course, arrange the DATA statements in the same order as the sprite shape table. However, this can be confusing when you want to change the sprite's shape. It's easier to find which DATA statement to change if you group the numbers by *column* instead of by *row*. That way, you know that the second number in the DATA statement controls the bits directly below those controlled by the first number. However, when you're POKEing a column into the shape table you must remember to add 3 to the address to prepare for the next POKE; or, if you're using a FOR-NEXT loop, use STEP 3.

**Calculating the byte values for the sprite shape.** Figure 1-10 shows the three top bytes in the first column of a sprite. In this

# Designing Graphic Shapes

**Figure 1-9. Sprite Grid**



**Figure 1-10. Calculating Sprite Byte Values**

	128	64	32	16	8	4	2	1	Byte Value
2240	■	□	□	■	□	□	■	□	$128 + 16 + 2 = 146$
2243	□	■	□	■	□	■	□	□	$64 + 16 + 4 = 84$
2246	□	□	■	■	□	□	□	□	$32 + 16 + 8 = 56$

# Designing Graphic Shapes

case, the shape table begins at location 2240. This corresponds to the location we used earlier. The darkened squares will become part of the sprite shape.

The numbers across the top of the byte grids are the bit values for each of the dots in the grid. Add up the values of the darkened dots to find the decimal number for that byte. In the first byte, which will be POKEd into location 2240, the darkened bits have the values 128, 16, and 2.  $128 + 16 + 2 = 146$ , so you will POKE 2240, 146. The next byte in the column will be POKEd into location 2243 (*not* 2241 — that will be the first byte in the *second* column).

## A Sprite Editor

As you can see, the process of programming a sprite can be a very long one. Also, once you have programmed a sprite, any changes must be made in the same time-consuming way.

To simplify this process, the following sprite editor automatically allocates some free memory space above the BASIC memory area (just as the character editor did) and allows you to edit sprites just as you edited the characters in the previous section.

## Instructions

1. Type in the program and SAVE it on disk or cassette tape.
2. When you are finished, type RUN and be prepared to wait about 20 seconds while the program initializes itself.

## The Screen Display

When the program has finished initializing, it will display the sprite editor screen. The screen is divided into four distinct areas (see Figure 1-11).

**The sprite display area.** This shows the eight sprites, numbered from left to right.

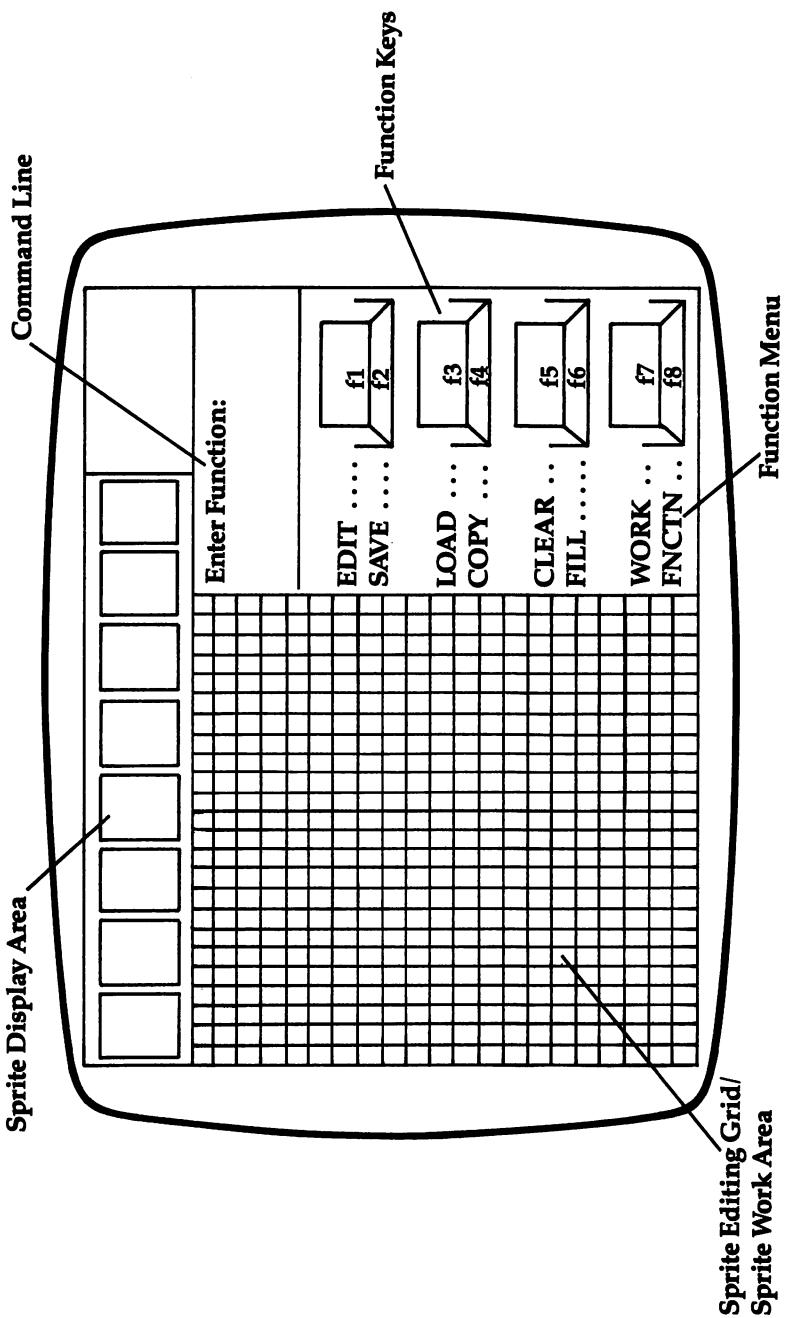
**The command line.** This area prompts the user whenever the system is awaiting a command. It is also used for entering data, such as filenames.

**The sprite editing grid and sprite work area.** While you are editing a sprite, it will appear in this grid. When you move the cursor around in the grid, the sprite shapes are automatically formed and stored.

When you are in the work space mode, this area will become blank, and the sprites may be moved here to experiment with them.

# Designing Graphic Shapes

**Figure 1-11. Sprite Editor Screen**



# Designing Graphic Shapes

**The function key display area.** This area displays the four function keys and their current functions. By pressing key f8 (SHIFT-f7), the functions will change, revealing eight more functions. (The eighth function in both sets is the function switch key.) In all, there are 14 different editing functions and seven work space functions for a total of 21 functions.

## The Editing Functions

**Edit.** After you press the EDIT key, a flashing cursor will appear in the sprite editing grid. You can move it anywhere within the grid by pressing the cursor up/down and cursor left/right keys exactly as you would normally use them for full screen editing.

Pressing SPACE will toggle the bit under the cursor. This means that a dark square will become light or a light square will become dark.

To exit the edit mode, just press any of the other function keys. Additionally, you can leave any sprite and begin editing another by simply pressing that character's key on the keyboard.

**Save.** Pressing the SAVE key generates the prompt:

**SAVE ON CASSETTE OR DISK? (C/D):**

Pressing C sets up the computer to SAVE on the cassette unit. Pressing D sets it up for the disk drive.

The next prompt asks for the filename. The name is limited to six characters and a filetype of .SPT. This will make all the sprite data sets easily identifiable on a disk. Entering the sixth letter of the filename initiates the SAVE process. When the SAVE is complete, the ENTER FUNCTION prompt will return on the command line.

**Load.** The LOAD function is nearly identical to the SAVE function in operation. It also allows for operation with either a disk or cassette unit. Of course, this operation requests the name of an existing character set and transfers it to memory. That set is then displayed at the top of the screen just as the original set was.

**Copy.** To copy a sprite, simply press the key that corresponds to the sprite you wish to replace. That sprite will then be displayed on the sprite editing grid. Then press the COPY key, f4. The command line will ask for the sprite to copy. Hit the number key that corresponds to the sprite (1-8) you wish to be copied into the grid and the old sprite will be replaced with the new one.

**Clear.** This function turns all of the bits in a sprite off allowing you to create a sprite using a "fresh scratch pad."

# Designing Graphic Shapes

**Fill.** This function is the opposite of the CLEAR function. It turns all of the bits in a sprite on. This is very handy for creating sprites that are mostly dark.

**Function** changes the action of the function keys. It allows access to the other seven editing functions, for a total of 14 editing functions using only eight keys

**Reverse.** This function toggles every bit in the character turning the bits that were on, off and the bits that were off, on. In effect, it creates a reverse version of the original sprite.

**Invert.** Invert turns a sprite upside down.

**Flip.** Turns a sprite over from left to right.

**Scroll.** The scroll functions rotate all of the bits in the sprite up, down, left, or right by one row or column. Bits falling off one edge will appear on the other side. Eight scrolls in one direction will restore the same sprite you started with.

## Work Mode

The work mode has seven functions of its own.

**Select.** This function only works in conjunction with other functions. For example, in the color mode it changes the color of the selected sprite.

**Color.** This function allows you to change the sprite colors. Pressing this key after selecting a sprite will allow you to change the sprite color by pressing the SEL key, f1. Pressing the SEL key repeatedly cycles through all 15 colors.

**Background.** This key changes the color of the work area. Once again, pressing the SEL key (f1) changes the colors. Press the SEL key repeatedly to cycle through all 15 colors.

**Add/Delete (AD/DL).** This allows you to put a sprite in or remove a sprite from the work area. Press the number of the sprite you wish to add to or delete from the work area and press the AD/DL key.

**Move.** This function allows you to move the sprite around within the work area. Once you have selected a sprite, pressing the MOVE key activates the cursor control keys. Pressing the cursor control keys will then move the selected sprite up, down, left, or right.

**Double Horizontal (2X HZ).** This key expands/shrinks the selected sprite by a factor of two in the horizontal direction. This is a toggle function. In other words, if the sprite is expanded, this key shrinks it; if it is normal size, this key expands it.

# Designing Graphic Shapes

**Double Vertical (2X VT).** This key expands/shrinks the selected sprite by a factor of two in the vertical direction. This is also a toggle function. If the sprite is expanded, this key shrinks it; if it is normal size, this key expands it.

**Function (FNCTN).** This key exits the work mode and returns to edit mode, erasing the work area and replacing it with the editing grid.

## **Program 1-5. The Sprite Editor**

```

10 REM *** CHANGE SCREEN POINTERS ***
20 REM
21 PRINT CHR$(8):;LZ$(0)="O":LZ$(1)="{RVS} {OFF}""
25 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,27,237,
     213,0,255,200,0,19,127,240,0,0,0
26 DATA 255,0,254,246,241,242,243,244,999
27 R3=PEEK(53272)
30 POKE 56578,PEEK(56578) OR 3
40 POKE 56576,(PEEK(56576) AND 252)OR 0
50 POKE 53272,(PEEK(53272) AND 240)OR 2
60 POKE648,196: FF=0
70 REM
71 REM *** SET - UP SPRITE REGISTERS ***
72 FOR R=53287 TO 53294: POKE R,14:NEXT
73 FOR R= 0 TO 7: POKE 51192+R, R: NEXT: POKE 532
     69, 255: POKE 53264,0
74 FOR R=49152 TO 49663 STEP 6:POKE R,170:POKER+1
     ,170:POKER+2,170:NEXT
75 FOR R=49155 TO 49663 STEP 6:POKE R,85:POKER+1,
     85:POKER+2,85:NEXT
76 FOR R=0 TO 14 STEP 2:POKE R+53248,25+R*15:POKE
     R+53249,52:NEXT
80 REM *** TRANSFER CHARACTER SET ***
90 REM
100 POKE 56334,PEEK(56334)AND254
110 POKE 1, PEEK(1)AND 251
120 FOR R=53248 TO 55296
130 POKE R-2048, PEEK(R): NEXT
140 POKE 1, PEEK(1)OR 4
150 POKE 56334,PEEK(56334)OR 1
160 REM
170 REM *** DEFINE CHARACTER GRID ***
171 DIM G(47),ZX(47),T(64),Y(64),CS(16)
180 REM
190 CG$="{ 24 O}{G}{DOWN}{ 25 {LEFT}}"
191 CF$="{}{RVS}{ 24 {SPACES}}{OFF}{G}{DOWN}
     { 25 {LEFT}}"
195 BK$="{}{ 24 {SPACES}}{DOWN}{ 24 {LEFT}}"
196 BE$="{}{ 24 {SPACES}}"

```

# Designing Graphic Shapes

```
230 REM *** ENABLE SPRITE DISPLAY ***
240 REM
250 POKE 53269,255
340 REM *** DEFINE FUNCTION KEYS ***
350 REM
351 FOR R=1TO30:S$=S$+"{RIGHT}":NEXT
360 FOR N=1TO8 STEP2
370 K$(N)={" 2 {RIGHT}}{ 5 {I}}{DOWN}{ 5 {LEFT}}
    {RVS} F"+STR$(N)+" {DOWN}{ 7 {LEFT}}{OFF}{M}
    {RVS}{ 5 {I}}{OFF} {G}"
380 K$(N+1)="{M}{RVS} F"+STR$(N+1)+" {*}{OFF}{G}
    {HOME}"
390 P$(N)={" 2 {RIGHT}}{A}{ 3 *} {S}{DOWN}
    { 5 {LEFT}}BF"+STR$(N)+"B{DOWN}{ 7 {LEFT}}
    {OFF}{M} {Z}{ 3 *} {X}{OFF} {G}"
400 P$(N+1)="{M}N F"+STR$(N+1)+" M{OFF}{G}{HOME}"
    : NEXT
405 REM
410 REM *** DEFINE KEY POSITIONS ***
415 REM
420 FOR R=1TO8:K$(R)={" 8 {DOWN}}{ 31 {RIGHT}}"+K
    $(R):NEXT
430 FOR R=1TO7 STEP2: K$(R)="{HOME}"+SP$+K$(R):SP
    $=SP$+"{ 4 {DOWN}}":NEXT
440 SP$=" { 3 {DOWN}}":FOR R=2TO8 STEP2: K$(R)="
    {HOME}"+SP$+K$(R):SP$=SP$+"{ 4 {DOWN}}":NEXT
    :SP$="
450 FOR R=1TO8:P$(R)={" 8 {DOWN}}{ 31 {RIGHT}}"+P
    $(R):NEXT
460 FOR R=1TO7 STEP2: P$(R)="{HOME}"+SP$+P$(R):SP
    $=SP$+"{ 4 {DOWN}}":NEXT
470 SP$=" { 3 {DOWN}}":FOR R=2TO8 STEP2: P$(R)="
    {HOME}"+SP$+P$(R):SP$=SP$+"{ 4 {DOWN}}":NEXT
480 REM
490 REM ***** DEFINE MESSAGES *****
500 REM
510 M$(1)="{HOME}{ 8 {DOWN}}{RIGHT}{DOWN}
    { 24 {RIGHT}}EDIT.{ 2 {DOWN}}{ 5 {LEFT}}SAVE
    .{ 2 {DOWN}}{ 5 {LEFT}}"
520 M$(1)=M$(1)+"LOAD.{ 2 {DOWN}}{ 5 {LEFT}}COPY
    { 2 {DOWN}}{ 5 {LEFT}}CLEAR{ 2 {DOWN}}
    { 5 {LEFT}}"
530 M$(1)=M$(1)+"FILL.{ 2 {DOWN}}{ 5 {LEFT}}WORK.
    { 2 {DOWN}}{ 5 {LEFT}}FNCTN"
540 M$(2)="{HOME}{ 9 {DOWN}}{ 25 {RIGHT}}REVRS
    { 2 {DOWN}}{ 5 {LEFT}}INVRT{ 2 {DOWN}}
    { 5 {LEFT}}"
550 M$(2)=M$(2)+"FLIP.{ 2 {DOWN}}{ 5 {LEFT}}SCL R
    { 2 {DOWN}}{ 5 {LEFT}}SCL L{ 2 {DOWN}}
    { 5 {LEFT}}"
```

# Designing Graphic Shapes

```
560 M$(2)=M$(2)+"SCL U{ 2 {DOWN}}{ 5 {LEFT}}SCL D
      { 2 {DOWN}}{ 5 {LEFT}}FNCTN"
570 REM
580 REM **** DEFINE RULER LINES ****
590 REM
600 L$="[{HOME}{ 3 {DOWN}}{ 30 *}{{E}{LEFT}{UP}-
      {LEFT}{UP}-{LEFT}{UP}-{ 3 {DOWN}}{ 9 *}}"
610 L$=L$+"{ 16 {LEFT}}{ 4 {DOWN}}{ 16 *_}"
900 REM
910 REM **** DISPLAY EDIT SCREEN ****
920 REM
930 PRINT"[CLR]";CD$:PRINTL$:PRINTM$(1)
932 PRINT"[HOME]{ 4 {DOWN}}";:FOR R=0TO19:PRINTCG
      $;:NEXT:PRINT"[ 24 O}{G}{HOME}";
1000 REM
1010 REM *** DISPLAY FUNCTION KEYS ***
1020 REM
1030 PRINT"[HOME]";:FOR R=1TO8: PRINTK$(R);: NEXT
      : PRINT"[HOME]";
1040 A(0)=1:A(1)=3:A(2)=5:A(3)=7:A(4)=2:A(5)=4:A(
      6)=6:A(7)=8
1050 REM
1060 REM *** GET KEYBOARD ENTRY ***
1070 REM
1080 PRINT"[HOME]{ 4 {DOWN}}{ 25 {RIGHT}}ENTER FU
      NCTION:{ 2 {DOWN}}{ 15 {LEFT}}";
1081 PRINT"[ 15 {SPACES}]"
1082 GETA$:IF A$=""THEN1080
1084 REM
1085 REM --- SKIP INVALID KEYS ---
1086 REM
1087 VA=ASC(A$)
1088 IF (VA>132 AND VA<141) THEN 1130
1089 IF (VA<49 OR VA>56) THEN 1080
1090 GOTO 1205
1100 REM
1110 REM --- GET FUNCTION KEYS ---
1120 REM
1130 PRINTP$(A(ASC(A$)-133))
1140 FORR=0TO99:NEXT
1150 PRINTK$(A(ASC(A$)-133));"[HOME]";
1160 GOTO 1390
1180 REM
1190 REM --- PRINT EDITED SPRITE ---
1200 REM
1205 POKE SP*2+53248,25+SP*30
1210 POKE 53264,2↑(VA-49):POKE53248+(VA-49)*2,40:
      SP=VA-49
1220 GOTO 1265
```

# Designing Graphic Shapes

```
1265 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}ENTRY MO
DE:{ 4 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1267 PRINT" SPRITE #";SP+1;" {LEFT}{ 6 {SPACES}}"
1268 PRINT" {HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINTC
G$;:NEXT:PRINT" { 24 O}{G}{HOME}";
1270 PRINT" {HOME}{ 4 {DOWN}}";
1272 LZ$(0)="O":LZ$(1)="{RVS} {OFF}"
1275 FOR R=0 TO 62
1277 IF (R<>0) AND (R/3=INT(R/3)) THEN GOSUB 1310
1278 RR=PEEK(R+(49152+64*SP))
1279 PRINTLZ$(ABS((RRAND128)=128));LZ$(ABS((RRAND
64)=64));LZ$(ABS((RRAND32)=32));
1280 PRINTLZ$(ABS((RRAND16)=16));LZ$(ABS((RRAND8)
=8));LZ$(ABS((RRAND4)=4));
1281 PRINTLZ$(ABS((RRAND2)=2));LZ$(ABS((RRAND1)=1
));
1283 NEXT: GOTO 1080
1310 PRINT" {DOWN}{ 24 {LEFT}}";:RETURN
1360 REM
1370 REM -- SPECIAL FUNCTION ROUTINES--
1380 REM
1390 IF FF=1 THEN 1500
1400 ON A(ASC(A$)-133) GOTO 2005,1432,1442,1452,1
461,1471,2700,1493
1410 GOTO 1170
1430 REM
1431 REM ----- SAVE SPRITES -----
1432 POKE SP*2+53248,25+SP*30:POKE 53264,0
1433 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SAVE ON
DISK OR{ 2 {DOWN}}{ 15 {LEFT}}";
1434 PRINT"CASSETTE? (C/D)"
1435 GET A$:IFA$=""THEN1435
1436 IF A$<>"C" AND A$<>"D" THEN 1087
1437 Q$=A$:GOTO 1760
1440 REM
1441 REM ----- LOAD SPRITES -----
1442 POKE SP*2+53248,25+SP*30:POKE 53264,0
1443 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}LOAD ON
DISK OR{ 2 {DOWN}}{ 15 {LEFT}}";
1444 PRINT"CASSETTE? (C/D)"
1445 GET A$:IFA$=""THEN1445
1446 IF A$<>"C" AND A$<>"D" THEN 1087
1447 Q$=A$:GOTO 1910
1450 REM
1451 REM ----- COPY A SPRITE -----
1452 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SPRITE T
O COPY? { 2 {DOWN}}{ 15 {LEFT}}";
1453 PRINT"ENTER (1-8){ 4 {SPACES}}"
1454 GET CA$:IF CA$="" THEN 1454
```

# Designing Graphic Shapes

```
1455 GOTO 1605
1456 IF CA$="" THEN 1454
1459 GOTO 1275
1460 REM ----- CLEAR A SPRITE -----
1461 PRINT"{}{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}CLEARING
    { 7 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1462 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1463 H=49152+64*SP
1464 FOR R=H TO H+63: POKE R,0: NEXT
1465 PRINT"{}{HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINTC
    G$;:NEXT:PRINT"{}{ 24 _}{G}{HOME}";
1467 GOTO 1080
1470 REM ----- FILL A SPRITE -----
1471 PRINT"{}{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}FILLING
    { 8 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1472 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1473 H=49152+64*SP
1474 FOR R=H TO H+63: POKE R,255: NEXT
1475 PRINT"{}{HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINTC
    F$;:NEXT:PRINT"{}{RVS}{ 24 {SPACES}}{OFF}{G}
    {HOME}"
1477 GOTO 1080
1490 REM
1491 REM --- SWITCH FUNCTION SET ---
1492 REM
1493 FF=1: PRINTM$(2); "{}{HOME}";:GOTO 1080
1499 RETURN
1500 ON A(ASC(A$)-133) GOTO 1522,1531,1541,1551,1
    561,1572,1582,1593
1510 GOTO 1170
1520 REM
1521 REM --- REVERSE SPRITE BITS---
1522 PRINT"{}{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}REVERSIN
    G:{}{ 5 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1523 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1524 H=49152+64*SP
1525 FOR R=H TO H+63: POKE R,255-PEEK(R): NEXT
1529 GOTO 1270
1530 REM --- INVERT SPRITE BITS---
1531 PRINT"{}{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}INVERTIN
    G:{}{ 5 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1532 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1533 H=49152+64*SP
1534 FOR R=H TO H+62: T(R-H)=PEEK(R): NEXT
1535 FOR R=H TO H+60 STEP 3
1536 POKE R,T(60-(R-H))
1537 POKE R+1,T(61-(R-H))
1538 POKE R+2,T(62-(R-H)): NEXT:GOTO 1270
1540 REM --- FLIP SPRITE BITS---
```

# Designing Graphic Shapes

```
1541 PRINT"{{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}}FLIPPING
      :{ 6 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1542 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1543 H=49152+64*SP
1544 FOR R=H TO H+62: T(R-H)=PEEK(R): NEXT
1545 FOR R=H TO H+60STEP 3:POKE R+2,T(R-H):POKE R+
    1,T(R+1-H):POKE R,T(R+2-H):NEXT
1546 FOR U=H TO H+63:Z=PEEK(U):R=128*(ABS((ZAND1)
    =1))+64*(ABS((ZAND2)=2))
1547 R=R+32*(ABS((ZAND4)=4))+16*(ABS((ZAND8)=8))+8*(ABS((ZAND16)=16))
1548 R=R+4*(ABS((ZAND32)=32))+2*(ABS((ZAND64)=64))+1*(ABS((ZAND128)=128)):POKEU,R
1549 NEXT:GOTO1270
1550 REM ----- SCROLL RIGHT -----
1551 PRINT"{{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}}SCROLLIN
      G RIGHT{ 2 {DOWN}}{ 15 {LEFT}}";
1552 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1553 FOR U=49152+64*SP TO (49152+64*SP)+62 STEP 3
1554 R=(PEEK(U)/2):R1=(PEEK(U+1)/2):R2=(PEEK(U+2)
    /2)
1555 IF PEEK(U)/2<>INT(PEEK(U)/2) THEN R1=R1+128
1556 IF PEEK(U+1)/2<>INT(PEEK(U+1)/2) THEN R2=R2+
    128
1557 IF PEEK(U+2)/2<>INT(PEEK(U+2)/2) THEN R=R+12
    8
1558 POKE U,R:POKEU+1,R1:POKEU+2,R2
1559 NEXT: GOTO1270
1560 REM ----- SCROLL LEFT -----
1561 PRINT"{{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}}SCROLLIN
      G LEFT { 2 {DOWN}}{ 15 {LEFT}}";
1562 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1563 FORU=49152+64*SPTO(49152+64*SP)+62STEP3:R=
    PEEK(U)*2:R1=PEEK(U+1)*2:R2=PEEK(U+2)*2
1564 IF PEEK(U)=>128 THEN R2=R2+1
1565 IF R2>255 THEN R2=R2-256
1566 IF PEEK(U+1)=>128 THEN R=R+1
1567 IF R>255 THEN R=R-256
1568 IF PEEK(U+2)=>128 THEN R1=R1+1
1569 IF R1>255 THEN R1=R1-256
1570 POKE U,R:POKEU+1,R1:POKEU+2,R2:NEXT: GOTO 12
    70
1571 REM ----- SCROLL UP -----
1572 PRINT"{{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}}SCROLLIN
      G UP{ 3 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1573 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1574 U=49152+64*SP:Y=PEEK(U):Y1=PEEK(U+1):Y2=PEEK
    (U+2)
1575 FOR R=0 TO 59 STEP 3
```

# Designing Graphic Shapes

```
1576 POKE U+R,PEEK(U+R+3):POKE U+R+1,PEEK(U+R+4):  
    POKE U+R+2,PEEK(U+R+5):NEXT  
1577 POKEU+60,Y:POKEU+61,Y1:POKEU+62,Y2  
1578 GOTO 1270  
1580 REM  
1581 REM ----- SCROLL DOWN -----  
1582 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SCROLLIN  
G DOWN { 2 {DOWN}}{ 15 {LEFT}}";  
1583 PRINT"SPRITE # { 5 {SPACES}}X { 7 {LEFT}}";SP  
+1  
1584 U=49152+64*SP:Y=PEEK(U+60):Y1=PEEK(U+61):Y2=  
PEEK(U+62)  
1585 POKEU+62,PEEK(U+62-3)  
1586 FOR M=59 TO 2 STEP -3  
1587 POKE U+M,PEEK(U+M-3):POKEU+M+1,PEEK(U+M-2):  
    POKE U+M+2,PEEK(U+M-1):NEXT  
1588 POKEU,Y:POKEU+1,Y1:POKEU+2,Y2  
1589 GOTO 1270  
1590 REM  
1591 REM --- SWITCH FUNCTION SET ---  
1592 REM  
1593 FF=0: PRINTMS(1);"{HOME}";:GOTO 1080  
1599 RETURN  
1600 REM  
1601 REM *** COPY SPRITE ROUTINE **  
1602 REM  
1605 DA=ASC(CA$)  
1610 IF (DA>132 AND DA<141) THEN VA=DA:GOTO 1130  
1620 IF (DA<49 OR DA>56) THEN 1080  
1690 VJ=49152+64*SP:JJ=49152+64*(DA-49)  
1700 FOR R=0 TO 63  
1710 POKE VJ+R,PEEK(JJ+R): NEXT  
1720 GOTO 1270  
1730 REM  
1740 REM *** SAVE SPRITES ROUTINE ***  
1750 REM  
1760 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SAVE: (NA  
ME){ 4 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";  
1765 PRINT"-----.SPR{ 5 {SPACES}}{ 15 {LEFT}}";  
1770 LL=0:NM$=""  
1775 FOR R=0 TO 30: PRINT" {RVS}-{OFF}{LEFT}";  
1780 GET A$:IFA$=""THEN NEXT  
1790 IFA$<>""THEN 1840  
1800 FOR R=0TO30: PRINT"-{LEFT}";  
1810 GET A$:IFA$=""THEN NEXT  
1820 IFA$<>""THEN 1840  
1830 GOTO 1775  
1840 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$  
=CHR$(34)ORA$="{UP}"THEN1775
```

# Designing Graphic Shapes

```
1845 IFA$={"DOWN"} ORA$={"RIGHT"} ORA$={"LEFT"} THEN  
1775  
1847 IF ASC(A$)>132 AND ASC(A$)<141 THEN 1080  
1850 PRINTA$;  
1855 NM$=NM$+A$:LL=LL+1:IFLL=6THEN1870  
1860 GOTO 1775  
1870 NM$=NM$+".SPR"  
1872 FOR R=53248 TO 53294:ZX(R-53248)=PEEK(R):  
    NEXT  
1874 POKE SP*2+53248,25+SP*30: POKE 53264, 0  
1875 IF Q$="C"THEN1890  
1880 RESTORE:OPEN 1,8,4,"@:"+NM$+",W": R=0  
1881 READ F: IF F<>999 THEN POKE R+53248,F:R=R+1:  
    GOTO 1881  
1882 FOR R=49152 TO 49663  
1883 PRINT"[HOME]{ 4 {DOWN}}{ 36 {RIGHT}}";R-4915  
    2  
1884 PRINT#1,PEEK(R):NEXT  
1885 FOR R=53248 TO 53294: POKE R,ZX(R-53248):  
    NEXT  
1888 CLOSE1: PRINT"[HOME]{ 6 {DOWN}}{ 25 {RIGHT}}  
    { 14 {SPACES}}";:GOTO 1080  
1890 OPEN 1,1,1,NM$:GOTO 1881  
1900 REM  
1901 REM *** LOAD SPRITES ROUTINE ***  
1902 REM  
1910 PRINT"[HOME]{ 4 {DOWN}}{ 25 {RIGHT}}LOAD:(NA  
    ME){ 4 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";  
1913 RESTORE  
1915 PRINT"-----.SPR{ 5 {SPACES}}{ 15 {LEFT}}";  
1920 LL=0:NM$=""  
1925 FOR R=0 TO 30: PRINT"(RVS)-{OFF}{LEFT}";  
1930 GET A$:IFA$=""THEN NEXT  
1935 IFA$<>""THEN 1960  
1940 FOR R=0TO30: PRINT"-{LEFT}";  
1945 GET A$:IFA$=""THEN NEXT  
1950 IFA$<>""THEN 1960  
1955 GOTO 1925  
1960 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$  
    =CHR$(34)ORA$="{UP}"THEN1925  
1965 IFA$>{"DOWN"}ORA$>{"RIGHT"}ORA$>{"LEFT"}THEN  
    1925  
1970 IF ASC(A$)>132 AND ASC(A$)<141 THEN 1080  
1975 PRINTA$;  
1980 NM$=NM$+A$:LL=LL+1:IFLL=6THEN1990  
1985 GOTO 1925  
1990 NM$=NM$+".SPR"  
1991 FOR R=53248 TO 53294:ZX(R-53248)=PEEK(R):  
    NEXT
```

# Designing Graphic Shapes

```
1992 POKE SP*2+53248,25+SP*30: POKE 53264, 0
1993 IF Q$="C"THEN 2001
1994 OPEN 1,8,4,NM$+",R": R=0
1995 READ F: IF F<>999 THEN POKE R+53248,F:R=R+1:
GOTO 1995
1996 FOR R=49152 TO 49663
1997 PRINT"{}HOME{} 4 {}DOWN{}){} 36 {}RIGHT})";R-4915
2
1998 INPUT#1,A:POKE R,A:NEXT
1999 FOR R=53248 TO 53294: POKE R,ZX(R-53248):
NEXT
2000 CLOSE1: PRINT"{}HOME{} 6 {}DOWN{}){} 25 {}RIGHT}){} 14 {}SPACES})";:GOTO 1080
2001 OPEN 1,1,0,NM$:GOTO 1995
2002 REM
2003 REM ** SPRITE EDIT ROUTINE **
2004 REM
2005 PRINT"{}HOME{} 4 {}DOWN{}){} 25 {}RIGHT}){}EDIT MOD
E:{} 5 {}SPACES}){} 2 {}DOWN{}){} 15 {}LEFT})";
2010 IF PEEK(53264)=0 THEN PRINT"{}RVS}WHICH SPRIT
E?{}OFF}){} 2 {}SPACES}){} 7 {}LEFT})";:GOTO 1082
2015 LO=50336: SV=PEEK(LO) :SM=(49152+(SP*64)): E
X=7 :GT=SC: BY=0
2020 FOR R=0TO30: POKE LO,SV
2030 GET A$:IFA$=""THEN NEXT
2040 IFA$<>""THEN 2090
2050 FOR R=0TO30: POKE LO,102
2060 GET A$:IFA$=""THEN NEXT
2070 IFA$<>""THEN 2090
2080 GOTO 2020
2090 IFA$="{UP}" THEN 2200
2091 IFA$="{DOWN}" THEN 2100
2092 IFA$="{LEFT}" THEN 2300
2093 IFA$="{RIGHT}" THEN 2400
2094 IF A$=CHR$(32) AND SV=160 THEN 2500
2095 IF A$=CHR$(32) AND SV=79 THEN 2600
2099 POKE LO,SV: GOTO 1087
2100 POKE LO,102:FORR=0TO40:NEXT
2110 IF GT=>SC+60 THEN 2020
2120 POKE LO,SV: LO=LO+40: GT=GT+3: SV=PEEK (LO)
:SM=SM+3: GOTO 2020
2200 POKE LO,102:FORR=0TO40:NEXT
2210 IF GT=<SC+2 THEN2020
2220 POKE LO,SV: LO=LO-40: GT=GT-3: SV=PEEK (LO):
SM=SM-3: GOTO 2020
2300 POKE LO,102:FORR=0TO40:NEXT
2310 IF EX=7 AND BY=0 THEN 2020
2315 IFEX=7THENPOKELO,SV:LO=LO-1:EX=0:SM=SM-1:GT=
GT-1:SV=PEEK (LO):BY=BY-1:GOTO 2020
```

# Designing Graphic Shapes

```
2320 POKE LO,SV: LO=LO-1: EX=EX+1: SV=PEEK(LO):  
      GOTO 2020  
2400 POKE LO,102:FORR=0TO40:NEXT  
2410 IF EX=0 AND BY=2 THEN 2020  
2415 IFEX=0THENPOKELO,SV:LO=LO+1:EX=7:SM=SM+1:GT=  
      GT+1:SV=PEEK(LO):BY=BY+1:GOTO 2020  
2420 POKE LO,SV: LO=LO+1: EX=EX-1: SV=PEEK(LO):  
      GOTO 2020  
2500 POKE LO, 79: POKE SM,PEEK(SM)-(2↑EX):SV=79:  
      GOTO2020  
2600 POKE LO, 160: POKE SM,PEEK(SM)+(2↑EX):SV=160  
      :GOTO2020  
2700 REM  
2701 REM *** WORK SPACE ROUTINE ***  
2702 REM  
2705 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}ENABLE S  
PRITE{ 2 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";  
2707 PRINT"WORK AREA{ 6 {SPACES}}"  
2709 FOR R=0 TO 14 STEP 2:POKE R+53248,25+R*15:  
      POKE R+53249,52:NEXT:POKE53264,0  
2710 W$(1)="{HOME}{ 8 {DOWN}}{RIGHT}{DOWN}  
{ 24 {RIGHT}}SEL..{ 2 {DOWN}}{ 5 {LEFT}}COLO  
R{ 2 {DOWN}}{ 5 {LEFT}}"  
2711 C$(0)="{BLACK}":C$(1)="{WHITE}":C$(2)="{RED}"  
":C$(3)="{CYAN}":C$(4)="{PURPLE}":C$(5)="  
{GREEN}":C$(6)="{BLUE}"  
2712 C$(7)="{YELLOW}":C$(8)="{ORANGE}":C$(9)="  
{BROWN}":C$(10)="{LT/RED}":C$(11)="{LT/GREY}"  
":C$(12)="{MED/GREY}":C$(13)="{LT/GREEN}"  
2713 C$(14)="{LT/BLUE}":C$(15)="{DK/GREY}"  
2720 W$(1)=W$(1)+"BCKGD{ 2 {DOWN}}{ 5 {LEFT}}AD/D  
L{ 2 {DOWN}}{ 5 {LEFT}}MOVE { 2 {DOWN}}  
{ 5 {LEFT}}"  
2730 W$(1)=W$(1)+"2X HZ{ 2 {DOWN}}{ 5 {LEFT}}2X V  
T{ 2 {DOWN}}{ 5 {LEFT}}FNCTN"  
2740 PRINT" {HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINTB  
K$;:NEXT:PRINTBE$;W$(1)  
2745 REM  
2747 REM -- GET WORK SPACE COMMANDS --  
2749 REM  
2750 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}ENTER FU  
NCTION { 2 {DOWN}}{ 15 {LEFT}}";  
2751 PRINT" { 15 {SPACES}}"  
2759 GETW$:IFW$=""THEN2759  
2760 WA=ASC(W$)  
2770 IF (WA>132 AND WA<141) THEN 3000  
2780 IF (WA<48 OR WA>56) THEN 2750  
2785 WS=WA-49  
2787 IF AD=1 THEN 2800
```

# Designing Graphic Shapes

```
2788 IF MV=1 THEN 3450
2789 GOTO 2750
2791 REM
2792 REM --- GET SPRITES ROUTINE ---
2793 REM
2800 IF PEEK(53249+WS*2)=>85 THEN 2900
2810 POKE 53248+WS*2,24:POKE 53249+WS*2,85:GOTO 2
    759
2900 POKE 53248+(WS*2),25+(WS*2)*15:POKE 53249+(WS
    *2),52:GOTO 2759
2910 GETW$ : IF W$ = "THEN2910
2920 WA=ASC(W$)
2930 IF (WA>132 AND WA<141) THEN 3000
2940 IF (WA<48 OR WA>57) THEN 2750
2950 WS=WA-49
2960 IF AD=1 THEN 2800
3000 ON WA-132 GOTO 3150,3325,3420,3620,3222,3510
    ,3720,3800
3100 REM
3101 REM ---- SELECT ROUTINE -----
3102 REM
3110 IF I=15 THEN I=-1
3120 I=I+1:RETURN
3150 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SELECT D
    ISABLED{ 2 {DOWN}}{ 15 {LEFT}}";
3160 PRINT "USE A/D OR MOVE"
3170 FOR R=0 TO 1500: NEXT: GOTO 2750
3200 REM
3210 REM --- SPRITE COLOR ROUTINE ---
3220 REM
3222 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}CHANGE S
    PRITE{ 2 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3223 PRINT "COLOR (USE SEL)"
3230 GET W$: CL$=W$: IF W$="" THEN 3230
3231 WA=ASC(W$)
3232 IF (WA>133 AND WA<141) THEN 3000
3233 IF (WA>47 AND WA<57) THEN WS=WA-49:GOTO 3270
3240 IF W$<>"{F-1}" THEN 2760
3250 GOSUB 3110
3260 POKE 53287+WS,I: GOTO 3230
3270 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SELECTED
    { 7 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3275 PRINT "SPRITE # { 7 {SPACES}}{ 7 {LEFT}}"; WS+1
3280 GOTO 3230
3300 REM
3310 REM -- BACKGROUND COLOR ROUTINE --
3320 REM
3325 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}USE SEL
    TO{ 5 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
```

# Designing Graphic Shapes

```
3326 PRINT"CHANGE BACKGRND"
3330 GET CL$: IF CL$=""THEN 3330
3340 IF CL$<>"{F-1}"THEN W$=CL$:GOTO 2760
3350 GOSUB 3100
3360 PRINT"{HOME}";C$(I);
3370 PRINT"{HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINT"
{RVS}";BK$;:NEXT:PRINTBE$;"{HOME}{LT/BLUE}"
{OFF}";: GOTO 3330
3400 REM
3410 REM --- MOVE SPRITES ROUTINE ---
3420 PRINT"{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}MOVING
{ 9 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3421 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";WS+1
3425 GETW$:IF W$=""THEN 3425
3427 WA=ASC(W$)
3429 IF W$="{DOWN}"ORW$="{UP}"ORW$="{RIGHT}"ORW$=
"{LEFT}"THEN 3435
3430 IF (WA>133 AND WA<141) THEN 3000
3432 IF (WA>47 AND WA<57) THEN WS=WA-49:GOTO 3480
3435 IF W$="{DOWN}"THEN 3460
3436 IF W$="{UP}"THEN 3465
3437 IF W$="{RIGHT}"THEN 3450
3438 IF W$="{LEFT}"THEN 3455
3450 IF PEEK(53248+WS*2)=192 THEN 3425
3451 POKE 53248+WS*2,PEEK(53248+WS*2)+1:GOTO 3425
3455 IF PEEK(53248+WS*2)=25 THEN 3425
3456 POKE 53248+WS*2,PEEK(53248+WS*2)-1:GOTO 3425
3460 IF PEEK(53249+WS*2)=228 THEN 3425
3461 POKE 53249+WS*2,PEEK(53249+WS*2)+1:GOTO 3425
3465 IF PEEK(53249+WS*2)=79 THEN 3425
3466 POKE 53249+WS*2,PEEK(53249+WS*2)-1:GOTO 3425
3480 PRINT"{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}MOVING
{ 9 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3481 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";WS+1
: GOTO 3425
3500 REM
3510 REM --- ADD/DELETE SPRITES ---
3520 PRINT"{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}ADD OR D
ELETE{ 2 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3530 PRINT"SPRITES{ 8 {SPACES}}"
3540 AD=1:MV=0:GOTO 2759
3600 REM
3610 REM --- EXPAND SPRITES VERTICAL --
3620 REM
3622 PRINT"{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}2X VERTI
CAL{ 4 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3623 PRINT"EXPANSION{ 6 {SPACES}}"
3660 IF (PEEK(53271) AND (2↑WS))=0 THEN POKE 5327
1,PEEK(53271)+(2↑WS):GOTO 2750
```

# Designing Graphic Shapes

```
3661 POKE 53271, PEEK(53271)-(2↑WS):GOTO 2750
3700 REM
3710 REM --- EXPAND SPRITES HORIZONTAL-
3720 REM
3722 PRINT"{{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}}2X HORIZ
ONTAL{ 2 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3723 PRINT"EXPANSION{ 6 {SPACES}}"
3760 IF (PEEK(53277) AND (2↑WS))=0 THEN POKE 5327
7,PEEK(53277)+(2↑WS):GOTO 2750
3761 POKE 53277, PEEK(53277)-(2↑WS):GOTO 2750
3800 REM
3810 REM --- RETURN TO MAIN ROUTINE ---
3815 PRINT"{{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}}EXITING
SPRITE { 2 {DOWN}}{ 15 {LEFT}}";
3817 PRINT"WORK AREA{ 6 {SPACES}}"
3820 PRINT"{{HOME}{ 4 {DOWN}}};:FOR R=0TO19:PRINTC
G$;:NEXT:PRINT"{{ 24 O}{G}{HOME}}";
3825 FOR R=0 TO 14 STEP 2:POKE R+53248,25+R*15:
POKE R+53249,52:NEXT:POKE53264,0
3830 PRINTM$(1);:FOR R=53287 TO53294: POKE R,14:
NEXT:POKE53277,0:POKE53271,0
3840 GOTO 1080
```

## High-Resolution Screen Graphics

The third kind of graphics the 64 can produce is high-resolution screen graphics. In this mode, the entire screen is defined by a single block of memory. Each bit in each byte of the memory block is displayed on the screen as a single dot. By turning the bits on and off, we turn individual dots on the screen on and off.

**Defining the high-resolution screen.** Like the text display screen, the high-resolution display screen is 40 characters wide by 25 rows high. But in high-resolution mode (also called bitmapped mode), the individual character positions are not a single byte containing a character code. Instead, each position, or cell, consists of *eight* bytes, with each cell defined right there in the bitmap. Each cell is defined by *on* and *off* bits exactly as characters are defined.

Since the actual number of bits stays the same on the high-resolution screen, the number of cells on the bitmapped screen is the same as the total number of characters on the text screen: 1000. Each of the 1000 cells is a grid of eight bits by eight bits, for a total of 64 bits. That means you can individually control a total of 64,000 dots on the screen!

# Designing Graphic Shapes

While the number of bits available to you in high-resolution mode is the same as the number of bits in text mode, in high-resolution mode, you are not limited to selecting one of the shapes in the character set. Instead, you have complete control over each and every dot. In a sense, it is like having one gigantic custom character, 320 dots wide and 200 dots high.

**The layout of high-resolution memory.** Figure 1-12 shows the 12 upper left-hand bytes of the high-resolution screen. In this mode, the bytes of screen memory are laid out in the same pattern as they were in character mode.

**Figure 1-12. Byte Map of the Upper-Left Corner of the High-Res Screen**

BYTE 0	BYTE 8	BYTE 16	BYTE 24	....
BYTE 1	BYTE 9	BYTE 17	BYTE 25	....
BYTE 2	BYTE 10	BYTE 18	BYTE 26	....
BYTE 3	BYTE 11	BYTE 19	BYTE 27	....
BYTE 4	BYTE 12	BYTE 20	BYTE 28	....
BYTE 5	BYTE 13	BYTE 21	BYTE 29	....
BYTE 6	BYTE 14	BYTE 22	BYTE 30	....
BYTE 7	BYTE 15	BYTE 23	BYTE 31	....
BYTE 320	BYTE 328	BYTE 336	BYTE 344	....
BYTE 321	BYTE 329	BYTE 337	BYTE 345	....
BYTE 322	BYTE 330	BYTE 338	BYTE 346	....
BYTE 323	BYTE 331	BYTE 339	BYTE 347	....
BYTE 324	BYTE 332	BYTE 340	BYTE 348	....
BYTE 325	BYTE 333	BYTE 341	BYTE 349	....
BYTE 326	BYTE 334	BYTE 342	BYTE 350	....
BYTE 327	BYTE 335	BYTE 343	BYTE 351	....
BYTE 640	BYTE 648	BYTE 656	BYTE 664	....
BYTE 641	BYTE 649	BYTE 657	BYTE 665	....
BYTE 642	BYTE 650	BYTE 658	BYTE 666	....
BYTE 643	BYTE 651	BYTE 659	BYTE 667	....
BYTE 644	BYTE 652	BYTE 660	BYTE 668	....
BYTE 645	BYTE 653	BYTE 661	BYTE 669	....
BYTE 646	BYTE 654	BYTE 662	BYTE 670	....
BYTE 647	BYTE 655	BYTE 663	BYTE 671	....

# Designing Graphic Shapes

While this configuration makes text and some special graphics applications easier to program, it makes bitmapped graphics (the kind of graphics we use on the hi-res screen most of the time) more difficult to program.

**The perfect screen grid.** With an ideal screen grid, we could specify a position on the screen by using simple X and Y (horizontal and vertical) position commands, which would turn the specified bit either ON or OFF.

Figure 1-13 shows a grid that is divided into eight squares by eight squares. To specify a single square within the grid, we can name its X and Y coordinates. The square in the position third from the left and fifth from the top would be expressed as the X,Y coordinates 2,E.

Within each of the 8 by 8 cells of the screen, we can access individual dots in more or less this fashion. But since the screen is arranged with the individual 8 by 8 cells in rows, we first have to determine which 8 by 8 cell on the screen contains the bit we want to change. Only then can we locate the bit *within* the cell.

**Figure 1-13. An Ideal Screen Grid**

	0	1	2	3	4	5	6	7
A								
B								
C								
D								
E								
F								
G								
H								

# Designing Graphic Shapes

**Locating cells within the screen.** Recall that the screen is divided into 1000  $8 \times 8$  cells — 40 in each row horizontally and 25 in each column. Now let's try to find the  $8 \times 8$  cell that contains the dot that is 55 dots in from the left and 43 dots down from the top of the screen. Remember that with X,Y coordinates, all locations on the screen are numbered from the upper left-hand corner of the screen. The upper left-hand cell is 0,0. The one just to the right is 1,0. The one just below is 0,1. (The column number, or horizontal position, is always listed first, as the X coordinate.)

Since we are looking for an  $8 \times 8$  cell, all we need to do is divide both the horizontal and vertical numbers by 8 to find the cell we need. (Ignore the remainder if the division doesn't come out even.) Since 55 divided by 8 is 6 (and a fraction), the X coordinate of the cell is 6. And since 43 divided by 8 is 5 (and a fraction), the Y coordinate of the cell is 5. So bit 55,43 is in cell 6,5.

How do you put that in a program? This line performs the operation:

**CX =INT(BX/8):CY =INT(BY/8)**

If BX and BY are the X and Y coordinates of the *bit*, then at the end of this routine CX and CY will contain the X and Y coordinates of the *cell*. The INT function chops off any fractions left by the division.

**Finding the Cell in Memory.** It isn't enough, unfortunately, to find the coordinates of the cell. We also have to find the actual bytes of the cell in memory.

Each cell consists of eight bytes, laid out exactly like a character pattern in character memory. That is, each row of eight bits is contained in a single byte, and eight bytes make up the whole  $8 \times 8$  cell. Look again at Figure 1-12. The *starting address* of each cell (that is, the address of the top byte in each cell) is always exactly 8 bytes higher than the address of the cell to the left. The starting address is always exactly 320 bytes higher than the address of the cell above. If the starting address of the bitmap is contained in the variable SB (Start Bitmap), then the starting address of cell 0,0 is SB + 0; the starting address of cell 1,0 is SB + 8; the starting address of cell 0,1 is SB + 320; and the starting address of the bottom right-hand corner cell, with coordinates 39,24, is 7992 — row number 24 times 320 bytes per row (7680) plus column number 39 times eight bytes per cell (312).

So to calculate, not the X,Y coordinate, but the actual starting

# Designing Graphic Shapes

address of the cell containing bit 55,43, we would use this line:

**CA =8\*INT(BX/8) +320\*INT(BY/8)**

Starting with the bit coordinates BX and BY, we end up with the Cell Address CA, which you would add to SB (Start of Bitmap) to get the actual address within the computer.

Let's try that with our example of bit 55,43. The horizontal bit position, BX, is 55. Dividing that by 8 and chopping off the fraction using INT, we get a result of 6. But now we multiply again by 8 to get the starting address of cell 6 within the row, since the starting byte of each cell is eight bytes away from the starting byte of the cell to the left or right of it. The horizontal byte address, then, is 48:  $8 \cdot \text{INT}(55/8)$ .

The vertical bit position, BY, is 43. Divide that by 8 and chop off the fraction to get the vertical cell position 5. Now we multiply that by 320 to get the starting address of row 5, since the starting byte of each row is 320 bytes away from the starting byte of the row above or below it. The vertical byte address, then, is 1600:  $320 \cdot \text{INT}(43/8)$ .

Add the horizontal address (48) and the vertical address (1600) to get CA, the starting address of the cell (counting from the start of the bitmap): 1648. The actual starting address in computer memory of the cell that contains bit 55,43 is SB + 1648.

**Finding the byte within the cell.** We've now located the actual starting address of the cell. Now, however, we have to get to the right byte *within* the cell. Since the bytes are "stacked" vertically, the byte within the cell will always be the remainder that was left over when we divided the vertical (Y) coordinate by 8.

Unfortunately, BASIC always expresses remainders as fractions, not as leftover integers. So to find the remainder in BASIC you always have to go through this process: *remainder=number-divisor\*INT(number/divisor)*. How does that work with real numbers? Let's see it with 43 divided by 8. We already know that the integer result of  $43/8$  is 5. To get the remainder, we multiply 5 by the divisor, 8. This gives us 40. Then we subtract that from the original number, 43, to get the remainder, which is 3.

How does this work when combined with our cell address routine? The variable CB will contain the final byte address. Along the way, we'll use the variable R (Remainder) and CV (Cell Vertical position):

**CV =INT(BY/8):R =BY -8\*CV:CA =8\*INT(BX/8) +320\*CV:CB =CA +R**

# Designing Graphic Shapes

That's an awful lot to pack into one line, so let's go through the process carefully to make sure it's clear. First, we calculate CV, the cell's vertical position, by getting the integer of BY divided by 8. Then we get R, the remainder, by subtracting  $8 \times CV$  from BY. Now we calculate CA, the cell address, exactly as we did before, except that CV already equals the result of the operation INT(BY/8), so we use CV instead of performing the same operation twice. Finally, we add R, the remainder, to CA, the cell address, in order to get the address of the byte that contains the bit 55,43.

Since the remainder was 3, the address of the byte is SB + 1648 + 3, or SB + 1651.

**Changing the Bit.** The variable CB now holds the address of the byte (relative to SB). However, we still have to find the bit within the byte. Fortunately, we can use the remainder from dividing BX by 8 to get the bit position, just as we used the remainder from dividing BY by 8 to get the byte address within the cell.

Let's use the variable BT to contain the bit position, and the variable CH as a temporary horizontal value, the way we used CV for a temporary vertical value:

```
CV=INT(BY/8) : R=BY-8*CV : CH=INT(BX/8) : BT=BX-8*CH  
: CA=8*CH+320*CV : CB=CA+R
```

The underlined statements are the routine that finds BT, the bit position. Notice that CB is still the same — it is used to find the *address*, which is the same regardless of which bit within the byte is to be changed.

Now that we have the bit position, what do we do with it? Remember that the bit position is a number from 0 to 7, representing the bit within the byte in this order:

**Bit position: 0 1 2 3 4 5 6 7**

Usually, however, we number the bits in a byte in the opposite direction:

**Usual order: 7 6 5 4 3 2 1 0**

There is a good reason for this. If bit 7 (usual order) is *on*, it has a value of 128. It so happens that 128 can be expressed as a power of 2 — in fact, 2 to the power of 7 ( $2 \uparrow 7$ ) or  $2^2 \cdot 2^2 \cdot 2^2 \cdot 2^2$ . Bit 6 has a value of 64, or  $2 \uparrow 6$ , or  $2^2 \cdot 2^2 \cdot 2^2 \cdot 2^2$ . Our process of locating the bit position has given us numbers that are the *opposite* of the usual order. Fortunately, they can be converted easily to the usual order simply by subtracting them from 7. This conversion could be done

# Designing Graphic Shapes

this way:  $BT = 7 - BT$ . In effect, this flips BT into reverse order. Now the value we want to use can be expressed as  $2 \uparrow BT$ .

**Turning on the dot.** Now we're ready to put the dot on the screen. If the screen is blank, we can do it this easily:

```
POKE SB+CB, 2↑BT
```

However, if there are dots already on the screen that we *don't* want to change, we have to be more careful. We have to pick up the byte at address SB + CB and turn on the bit without changing the rest of the byte. In fact, it might be that the bit we're turning on is *already* on. If you are drawing lines, this will often happen — where the new line crosses an old line, your program will try to turn on a bit that was already on.

So we will turn on the bit using the logical command OR. Remember that OR turns on bits whether they were already on or not. So our turn-on-the-bit command will look like this:

```
POKE SB+CB, PEEK(SB+CB) OR 2↑BT
```

In effect, we pick up the byte, make sure the right bit is on, and put it back without making any other changes.

**Turning off the bit.** Turning off a bit is a similar process, except that instead of putting a 1 in the bit position, we want to put a 0 there. So this time, we'll use AND with the complement of  $2 \uparrow BT$ . The complement of  $2 \uparrow BT$  is the number that has a 1 in every position *except* the bit we want to erase. We find the complement by subtracting  $2 \uparrow BT$  from 255.

To turn off the bit without disturbing the rest of the byte, use this statement:

```
POKE SB+CB, PEEK(SB+CB) AND 255-2↑BT
```

**Bit value arrays.** A faster way to handle bit operations is to avoid the exponential operation ( $\uparrow$ ) and the subtractions from 255 and 7 during run time by setting up the correct values in arrays. We'll use the array NG(*n*) to hold the values to use in turning *off* bits, and the array PS(*n*) to hold the values to use in turning *on* bits:

```
10 DIM NG(7),PS(7):GOSUB 5000
5000 FOR I=0 TO 7:BT=7-I:PS(I)=2↑BT:NEXT
5010 FOR I=0 TO 7:NG(I)=255-PS(I):NEXT:RETURN
```

Once these two arrays are in place, you can use BT in its original form, without flipping it over. To turn on a bit, use this line:

```
POKE SB+CB,PEEK(SB+CB)OR PS(BT)
```

# Designing Graphic Shapes

To turn off a bit, use this line:

**POKE SB +CB, PEEK(SB +CB)AND NG(BT)**

To speed up the operation even more, replace the expression SB + CB with a single variable, like AD (ADdress):

**AD =SB +CB:POKE AD,PEEK(AD)OR PS(BT)**

Here's a program that puts all this together. When prompted, enter the X,Y coordinates for any particular dot on the screen (0-319 horizontally, 0-199 vertically). The program will show you the relative address of the byte and the values to POKE there to draw or erase the dot.

```
10 DIM NG(7),PS(7):GOSUB 5000
100 PRINT "X (HORIZONTAL)":INPUT BX
110 IF BX<0 THEN BX=0
120 IF BX>319 THEN BX=319
130 PRINT "Y (VERTICAL)":INPUT BY
140 IF BY<0 THEN BY=0
150 IF BY>199 THEN BY=199
160 GOSUB 200
170 PRINT "ADDRESS=SB+"CB
180 PRINT "DRAW: POKE AD,PEEK(AD)OR "PS(BT)
190 PRINT "ERASE: POKE AD,PEEK(AD)AND "NG(BT)":PRIN
T:GOTO 100
200 CV=INT(BY/8):R=BY-8*CV:CH=INT(BX/8):BT=BX-8*CH
:CA=8*CH+320*CV:CB=CA+R
210 RETURN
1000 END
5000 FOR I=0 TO 7:BT=7-I:PS(I)=2↑BT:NEXT
5010 FOR I=0 TO 7:NG(I)=255-PS(I):NEXT:RETURN
```

## Switching to High-Resolution Mode

Switching the 64 to hi-res mode can be accomplished with a single POKE:

**POKE 53265,PEEK(53265) OR 32**

If you enter this, you will find the screen display becomes a mixture of colors and shapes that make little or no sense.

**A closer look at high-resolution graphics.** Normally, the 64 gets its video (screen display) information from three places in memory. These are screen memory, which is at 1024-2023 unless you change it; the character ROM, which contains the shapes of all the characters; and color memory, which controls the color(s) of the characters and their background. When the 64 switches into high-resolution mode, however, the video information comes from different places.

# Designing Graphic Shapes

The video shapes (as was mentioned in the previous section) come from a new place, the 8000-byte block of memory that controls the bit patterns on the screen. This is the *bitmap*. The *colors* are taken from the former *screen memory* (usually locations 1024-2023). By changing the values stored in the screen memory, we can change the crazy quilt of colors into something more readable.

**FOR R = 1024 TO 2023:POKE R,230:NEXT**

After running this line, the screen becomes our old, familiar dark blue on light blue again (with a multicolor message — the 64 thinks it is saying READY). For more information on how color works in high-resolution mode, see Chapter 2. For now, let's take another look at what you see on the screen.

The first thing you may notice is that the character set is displayed in the lower half of the screen. Also, some of the dots in the upper part of the screen are flickering. What are we looking at?

**An X-ray view of memory.** What you are seeing is the first 8000 bytes of memory in the 64. The upper left-hand corner displays memory location 0 and the lower-right corner displays location 7999. The bytes between 4096 and 7999 display most of the character set. This is because, although the character ROM is really located between 53248 and 57343, the VIC-II chip would not be able to use it if it were outside the 16K bank of RAM it is addressing. To compensate for this, the 64 has been designed so the VIC-II chip thinks that the character ROM is located between 4096 and 8191 when the VIC-II is reading bank 0, the default block. So we are looking at the first 8000 bytes of memory as the VIC-II chip sees them and the bytes that look like the character set actually are an image of the character ROM. The jumbled dots at the top of the screen are part of the computer's operating system and the flickering dots are bits in the computer's memory that switch on and off as the computer operates. In a sense, we are watching the 64 think.

Look about an eighth of the way down the screen at a section of screen that looks like parallel vertical bars. These bars are made up of 1024 bytes of memory, each of which contains the number 230. We just POKEd that data there when we changed the screen colors to light blue on dark blue. This is normally the screen RAM. In high-resolution mode, this area is used as the color RAM.

# Designing Graphic Shapes

**Clearing the high-resolution screen.** Looking at the hidden mysteries of the 64 by displaying RAM and ROM with the video chip is interesting, but it doesn't allow us to create any high-resolution displays of our own. To do that, we must first clear the screen. Try hitting SHIFT and CLR/HOME as you normally would to clear the screen.

It doesn't work. The screen switches to black on red, but the jumble of screen data remains. This is because screen memory, which is cleared out by POKEing 32 (the CHR\$ value for a space) into every location, is not used for the screen data any more, but is used for color data instead. So when we hit SHIFT and CLR/HOME, only the screen *colors* are changed — the bitmap, as a whole, is left alone.

To clear the screen, we need to POKE 0's into the 8000 bytes used for the bitmap. Right now, however, the bitmap sits on top of part of the operating system. Filling it with 0's will shut down the machine. We have to move the bitmap somewhere else.

Since the space the VIC-II can look at is 16K long, and we need almost half of that space to map the screen, we have only one choice — the other half of the 16K block, between 8192 and 16384.

**Moving the high-resolution screen.** To move the hi-res screen, we need to switch bit 3 of register 53272 in the VIC-II register. If bit 3 is set to 0, the bitmap will be in the lower half of the video block. If bit 3 is set to 1, it will be in the upper half. Here is a routine that will switch the 64 to high-resolution mode, move the bitmap to the upper half of the 16K block, and clear it.

```
10 POKE 53265, PEEK(53265) OR 32
20 POKE 53272, PEEK(53272) OR 8
30 FOR R=8192 TO 16191
40 POKE R,0: NEXT
50 FOR R=1024 TO 2023
60 POKE R, 230: NEXT
```

**More trouble with the READY message.** When you run this routine, you will first see all of the jumbled dots cleared to simple colored blocks. Then the colored blocks will be replaced by the familiar blue background. Afterward, however, a group of colored blocks appears on the screen. Once again, these blocks are the READY message showing up in screen memory (now used to hold the colors). How do we get rid of it?

As we did earlier to avoid having text scroll up and off the

# Designing Graphic Shapes

screen, we can defeat the READY message by keeping the program from ending. Now the routine loops back on itself.

```
70 GOTO 70
```

## A Sketching Routine

By combining the high-resolution routine with a routine like the one we used before to plot the X and Y coordinates of a single dot, we can create a simple high-resolution sketching routine for the 64.

This routine allows you to draw pictures on the screen of the 64 by using the CURSOR UP/DOWN and CURSOR LEFT/RIGHT keys on the keyboard. When you run this routine, it will switch the screen from normal text mode to high-resolution mode and clear the screen area one line at a time. It takes about 20 seconds for this process. When the bitmap has been cleared, the computer will wait for you to press one of the cursor keys. It will then start drawing in the direction you choose.

## Program 1-6. A Sketching Routine

```
10 PRINT "[CLR][BLUE]";:X=160:Y=100
20 FOR R=1024 TO 2023:POKE R,230:NEXT
30 POKE 53272,PEEK(53272)OR8:POKE 53265,PEEK(53265)
   OR32
40 FOR R=8192 TO 16383:POKE R,0:NEXT
50 GET A$:IF A$=""THEN 50
60 IF A$="{UP}" THEN Y=Y-1:GOTO 110
70 IF A$="{DOWN}" THEN Y=Y+1:GOTO 110
80 IF A$="{RIGHT}" THEN X=X+1:GOTO 110
90 IF A$="{LEFT}" THEN X=X-1:GOTO 110
100 GOTO 50
110 P=(8192+(INT(Y/8))*320+(INT(X/8))*8+(YAND7))
120 POKE P,PEEK(P)ORINT(2↑(7-(((INT(X)/8)-INT(INT
   (X)/8))*8)))
130 GOTO 50
```

## How the Program Works

Line 10 clears the screen and sets the starting point in the center of the screen. X can be any value from 0 to 319 while Y runs from 0 to 199. The values 160 and 100 are therefore approximately the middle positions on the screen.

Line 20 sets the color of the background to dark blue and the lines to light blue. (For more detail on how this works, see Chapter 2.)

# Designing Graphic Shapes

Line 30 switches the 64 to high-resolution mode and moves the screen data to the upper half of the 16K block.

Line 40 clears the bitmap.

Lines 50 to 100 GET the cursor direction and update the X and/or Y values.

Lines 110 and 120 put the new dot on the screen and line 130 sends the program back to get another control key.

## Screen Editor

Here is a program that incorporates the features of the sketching routine with a number of built-in plotting functions. It allows you to erase lines from the screen just as you drew them, using the cursor control keys.

**Running the screen editor.** When you first run this program, it seems to pause for a moment while the 64 sets up the shape tables and screen pointers for the high-resolution screen. Then the screen displays what appears to be garbage. This is the un-cleared bitmap and screen (color) memory display. These will be cleared while you watch. Afterward, you will see a display of the four function keys on the lower right side of the screen.

**Keys on.** This function turns on the function key display. It is accessed by pressing f1.

**Keys off.** This key turns the function key display off. It can be accessed by pressing SHIFT-f1.

**Erase.** This is the reverse of the draw function. It makes the dots on the screen the same color as the background. Pressing f3 enables the erase function. When the erase mode is enabled, there is a tiny flashing dot on the screen indicating the position of the cursor.

**Draw.** This function is enabled when the program is first run. To return to the draw mode after enabling the erase function, press SHIFT-f3.

**Square.** This function allows you to automatically draw a square or rectangle of any dimension (within the size of the screen) by simply specifying two of its opposite corners. This is how it works:

Enter the SQUARE mode by pressing the f5 function key.

Move the cursor to one of the corners of the square or rectangle to be drawn and press the C (corner) key.

Move the cursor to the opposite corner of the square (or rectangle) and press the C key again. The object will now appear.

**Clear.** This function erases the entire screen. Because this program was written entirely in BASIC, it will take several

# Designing Graphic Shapes

seconds for the screen to clear. Press SHIFT-f5.

**Line.** This function can be used to draw a line between any two points on the screen. The procedure is:

Enter the LINE mode by pressing the f7 function key.

Move the cursor to one of the end points of the line to be drawn and press the C key.

Move the cursor to the other end of the line and press the C key again. This starts the draw procedure.

**Circle.** This function will draw a circle of any size within the limits of the screen. In fact, it can draw part of a circle that exceeds the screen size if the center is placed near one of the screen edges. Placing the center near the left or right edge of the screen will draw a circle that wraps around the screen to the opposite side of the screen. Placing the center near the top or bottom of the screen will cause some of the circle to disappear entirely.

There is a danger in this, however. The bits that do not appear on the screen will write over other parts of memory, and if part of your program or some important data is near the high-resolution screen, that data will be written over. It is not advisable to put a circle near the top or bottom of the screen if part of it will be drawn off the edge.

The procedure for drawing a circle is very similar to drawing a square or line.

Enter the circle mode by pressing SHIFT-f7.

Move the cursor to the center of the circle to be drawn and press the C key.

Move the cursor to the outside edge of the circle and press the C key again. This starts the "draw" procedure.

## Saving Custom High-Resolution Screens

Once you have drawn a high-resolution screen, it will remain unchanged until you deliberately write some new data into that block of memory. You can even stop the program without affecting the screen by pressing RUN/STOP-RESTORE. To save a screen, exit the editing program and type:

**RUN 1000**

You will then be prompted with:

**NAME OF SCREEN?**

Give the screen a name (less than 12 characters) and press [RETURN]. You will then be asked:

**SAVE ON DISK OR CASSETTE (D/C)?**

# Designing Graphic Shapes

Entering C will save the program on cassette and entering D will save it on disk, assuming the device is already connected to your computer. Entering any other character will send you back to the prompt.

All screen files are saved with a file type of .SCN. This means that the letters .SCN will be added to the name you choose for your file. This will make screen data files easy to find on your tapes or diskettes. It will also allow for automatic retrieval of all screen files (see Chapter 4).

## Loading Custom Screens

This section of the program is nearly identical to the save section, except it reads data from disk or tape. To load a high-resolution screen using the editor, press RUN/STOP-RESTORE. Then type:

**RUN 2000**

You will be prompted with:

**NAME OF SCREEN?**

Enter the screen name (less than 12 characters) and press RETURN. You will then be asked:

**LOAD FROM DISK OR CASSETTE (D/C)?**

Entering C will LOAD the program from cassette and entering D will LOAD it from disk. Entering any other character will send you back to the prompt. If the file can't be found, you'll get an error message, so it's a good idea to write down each filename when you save it.

After the screen has been loaded, it will re-initialize and you may edit it just as if it had been entered by hand for the first time.

Study the program to see how circles, squares, and lines are plotted.

## Program 1-7. Hi-Res Screen Editor

```
0 REM *** SPRITE KEYSHAPE DATA ***
1 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
   15,255,240,12,62,240,13,252
2 DATA 240,12,126,240,141,254,241,141,252,113,143
   ,255,241,128,0,1,140,62,113
3 DATA 157,253,185,188,127,125,253,254,255,253,25
   2,63,255,255,255,0,0,0,0,0
4 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,15,255,240
   ,12,60,240,13,251,112,12,126
5 DATA 240,141,251,113,141,252,241,143,255,241,12
   8,0,1,140,60,241,157,250,249
```

# Designing Graphic Shapes

# Designing Graphic Shapes

```
96 REM *** ENABLE HI-RES SCREEN ***
97 REM
100 PRINT "[CLR]"; : X=160: Y=100
110 POKE 53272, 29: POKE 53265, 59: POKE 56576, 198
120 FOR R=17408 TO 18407: POKE R, 230: NEXT
130 FOR R=24576 TO 32767: POKE R, 0: NEXT
135 REM
136 REM *** KEYBOARD ENTRY ROUTINE ***
137 REM
140 GET A$: IF A$="" THEN 140
141 IF A$="[UP]" THEN 1030
142 IF A$="[DOWN]" THEN 1130
143 IF A$="[LEFT]" THEN 1230
144 IF A$="[RIGHT]" THEN 1330
145 IF A$="S" THEN 1430
146 IF A$="L" THEN 1530
150 IF ASC(A$)>140 OR ASC(A$)<133 THEN 140
151 F=ASC(A$)-132
152 ON F GOTO 230, 430, 630, 830, 330, 530, 730, 930, 103
    0, 1130, 1230, 1330
154 F=ASC(A$)-132
200 REM
210 REM *** KEYS ON ***
220 REM
230 POKE 53269, 255
240 GOTO 140
300 REM
310 REM *** KEYS OFF ***
320 REM
330 POKE 53269, 0
340 GOTO 140
400 REM
410 REM *** SET CURSOR TO ERASE ***
420 REM
430 M=1
440 GOTO 140
500 REM
510 REM *** SET CURSOR TO DRAW ***
520 REM
530 M=0
540 GOTO 140
600 REM
610 REM *** SQUARE ***
620 REM
630 IF S=1 THEN 650
640 S=1: X1=X: Y1=Y: GOSUB 2000: GOTO 140
650 GOSUB 2000: S=0: X2=X: Y2=Y: Y=Y1
655 FOR X=X1 TO X2 STEP(X1>X2OR1): GOSUB 2000:
    NEXT
```

# Designing Graphic Shapes

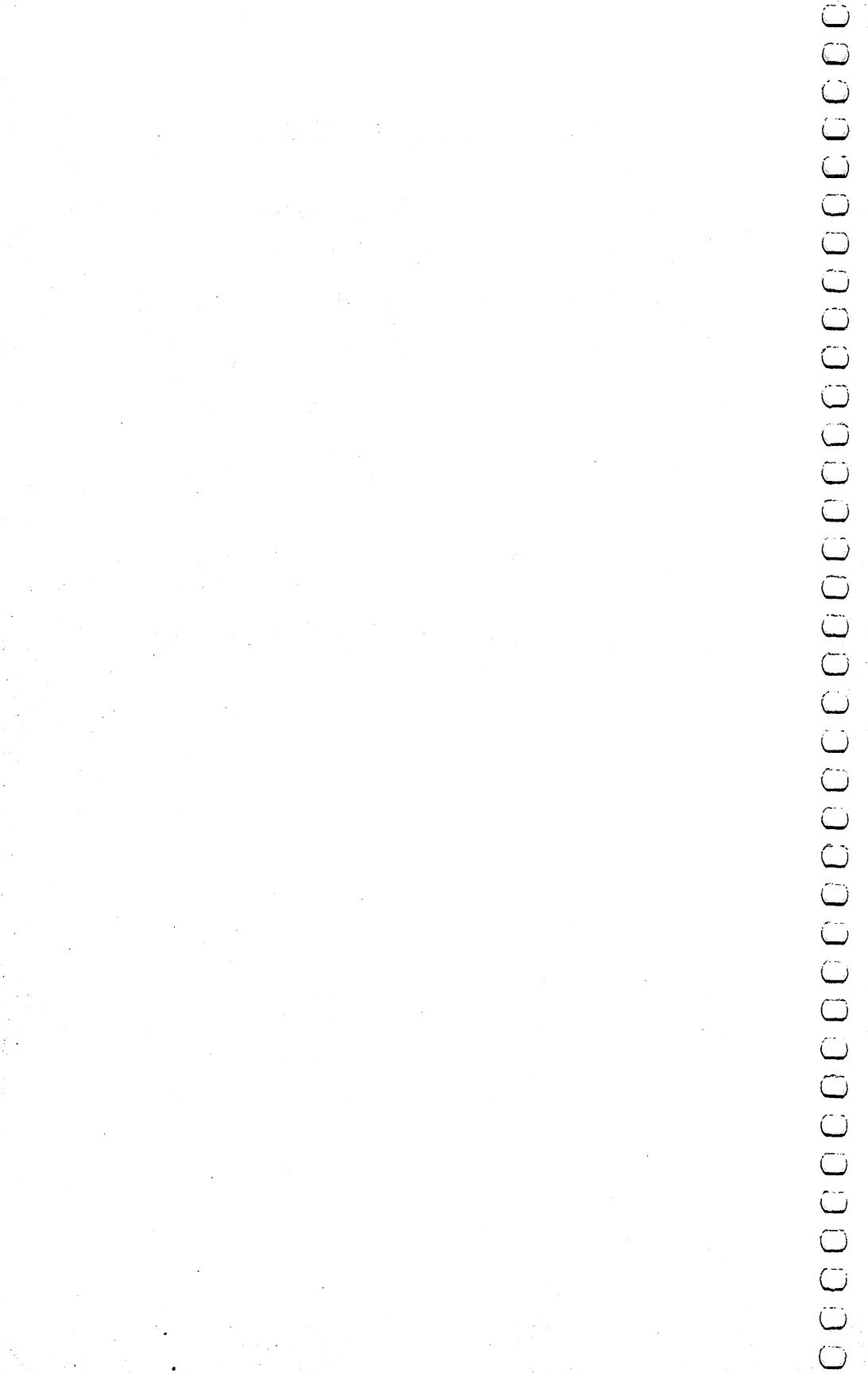
```
660 FOR Y=Y1 TO Y2 STEP(Y1>Y2OR1):GOSUB 2000:  
    NEXT  
665 Y=Y-(Y1>Y2OR1)  
670 FOR X=X2 TO X1 STEP(X2>X1OR1):GOSUB 2000:  
    NEXT  
675 Y=Y+(Y1>Y2OR1)  
680 FOR Y=Y2 TO Y1 STEP(Y2>Y1OR1):GOSUB 2000:  
    NEXT  
690 GOTO 140  
700 REM  
710 REM *** CLEAR ***  
720 REM  
730 GOTO 130  
800 REM  
810 REM *** LINE ***  
820 REM  
830 IF L=1 THEN 850  
840 L=1: X1=X: Y1=Y:GOSUB 2000:GOTO 140  
850 L=0: X2=X: Y2=Y: Y=Y1  
855 SP=(ABS(Y1-Y2))/(ABS(X1-X2))  
860 FOR X=X1TOX2STEP.3*(X1>X2OR1):Y=Y+(SP*.3*(Y1>Y  
    2OR1)):GOSUB2000:NEXT:GOTO140  
900 REM  
910 REM *** CIRCLE ***  
920 REM  
930 IF C=1 THEN 950  
940 C=1: X1=X: Y1=Y:GOSUB 2000:GOTO 140  
950 C=0: X2=X: Y2=Y  
955 R1=SQR(((X1-X2)↑2)+((Y1-Y2)↑2)):R2=R1*.72  
960 FORT=0TO2*{↑}STEP(1.5/R1):X=(R1*(COS(T)))+X1:  
    Y=(R2*(SIN(T)))+Y1:GOSUB2000:NEXT  
970 GOTO 140  
1000 REM  
1010 REM *** UP ***  
1020 REM  
1030 Y=Y-1  
1040 ON M+1 GOSUB 2000,2100  
1050 GOTO 140  
1100 REM  
1110 REM *** DOWN ***  
1120 REM  
1130 Y=Y+1  
1140 ON M+1 GOSUB 2000,2100  
1150 GOTO 140  
1200 REM  
1210 REM *** LEFT ***  
1220 REM  
1230 X=X-1  
1240 ON M+1 GOSUB 2000,2100
```

# Designing Graphic Shapes

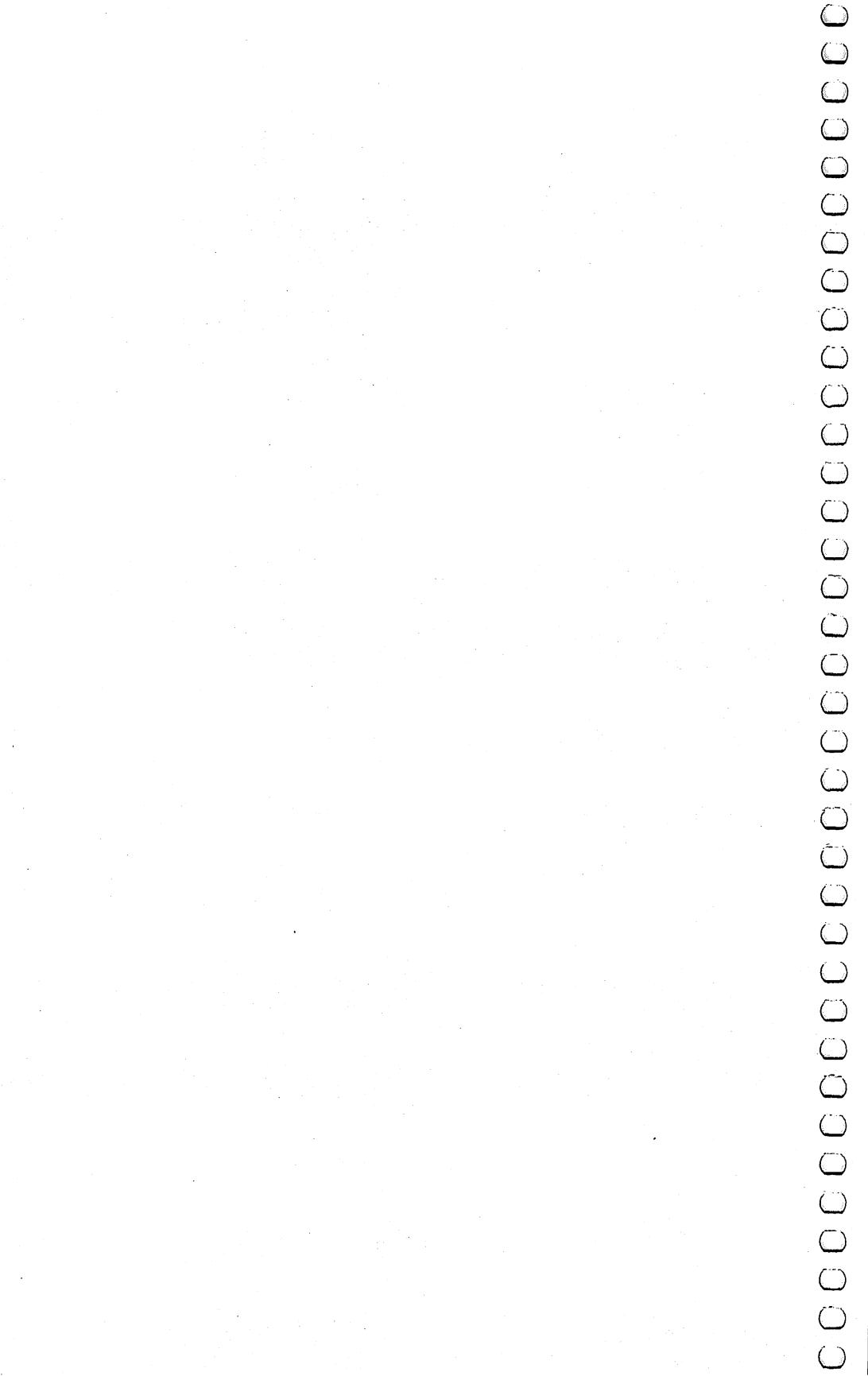
```
1250 GOTO 140
1300 REM
1310 REM *** RIGHT ***
1320 REM
1330 X=X+1
1340 ON M+1 GOSUB 2000,2100
1350 GOTO 140
1400 REM
1410 REM *** SAVE SCREEN ROUTINE ***
1420 REM
1430 POKE53272,21:POKE53265,27:POKE56576,147:POKE
53269,0
1435 INPUT"CLR){ 2 {DOWN}}NAME OF SCREEN";SN$
1440 INPUT" 2 {DOWN}}SAVE ON DISK OR CASSETTE (D
/C)";DC$
1450 IF DC$="D"THEN 1480
1460 IF DC$="C"THEN 1490
1470 INPUT"UP}PLEASE ENTER D OR C{ 12 {SPACES}}"
;DC$
1480 OPEN1,8,4,"@:"+SN$+.SCN,W":GOTO1495
1490 OPEN1,1,1,SN$+.SCN"
1495 FOR R=24576 TO 32767:PRINT#1,PEEK(R):NEXT:
CLOSE 1
1496 POKE53272,29:POKE53265,59:POKE56576,198:
POKE53269,255: GOTO 140
1500 REM
1510 REM *** LOAD SCREEN ROUTINE ***
1520 REM
1530 POKE53272,21:POKE53265,27:POKE56576,147:POKE
53269,0
1535 INPUT"CLR){ 2 {DOWN}}NAME OF SCREEN";SN$
1537 INPUT"LOAD FROM DISK OR TAPE (D/C)";DC$
1540 IF DC$="D"THEN 1570
1550 IF DC$="C"THEN 1580
1560 INPUT"UP}PLEASE ENTER D OR C{ 12 {SPACES}}"
;DC$
1565 IF DC$<>"D" OR DC$<>"C" THEN 1560
1566 GOTO 1540
1570 OPEN1,8,4,"@:"+SN$+.SCN,R":GOTO1590
1580 OPEN1,1,0,SN$+.SCN"
1590 FOR R=24576 TO 32767:INPUT#1,H:POKER,H:NEXT:
CLOSE 1
1596 POKE53272,29:POKE53265,59:POKE56576,198:
POKE53269,255: GOTO 140
2000 REM
2010 REM *** TURN ON DOT ROUTINE ***
2020 REM
2030 P=(24576+(INT(Y/8))*320+(INT(X/8))*8+(YAND7)
)
```

# Designing Graphic Shapes

```
2040 POKEP,PEEK(P)OR2↑(7-(XAND7)):RETURN  
2100 REM  
2110 REM *** TURN OFF DOT ROUTINE ***  
2120 REM  
2130 P=(24576+(INT(Y/8))*320+(INT(X/8))*8+(YAND7)  
     )  
2140 POKEP,PEEK(P)OR2↑(7-(XAND7))  
2150 P=(24576+(INT(Y/8))*320+(INT(X/8))*8+(YAND7)  
     )  
2160 POKEP,PEEK(P)AND(255-(2↑(7-(XAND7)))):RETURN
```



2  
Color



# Color

## 2

In the previous sections, we covered the three major methods of programming 64 graphics shapes: character graphics, sprite graphics, and screen graphics.

Since the methods of programming colors are intimately connected to the methods of creating shapes, let's review how shapes are made.

### Character Graphics

Character shapes are created in blocks eight bits high and eight bits wide. When the 64 is first turned on, it uses a section of pre-programmed memory, the character ROM, to define the shapes of these blocks. Each pattern is contained in a group of eight bytes, with the first byte defining the top row and the last byte defining the bottom row of the character pattern.

There are three methods of using these shapes; each of them uses the preprogrammed character shapes, but each accesses them in a different way.

**Using the keyboard.** Every time you press a key, the computer reads the keyboard and receives a special number corresponding to the key that was pressed. In most cases, the computer then translates that number into a character code and displays that character on the screen.

**The CHR\$ function.** Character codes can also be entered using PRINT CHR\$(*n*). For example, to PRINT the letter A, which has character code 65, you would type:

**PRINT CHR\$ (65)**

The character codes for all of the characters the 64 can produce using the built-in character ROM are listed in Appendix F. These codes also allow you to use a number of control functions from within a program that would not otherwise be available, such as CHR\$(13), which is the code for RETURN.

**POKEing into screen memory.** The screen is mapped in screen memory, where each of 1000 bytes contains the screen code for the character to be displayed at a particular spot. Screen codes are not the same as character codes; screen codes are based on the order of the character patterns in the character ROM. Appendices G, I, and J list all of the screen codes for the 64.

# Color

**Custom characters.** You can bypass character ROM and design your own patterns, as we saw in Chapter 1. These are dealt with in exactly the same way as standard characters. As far as the computer is concerned, the only difference between custom characters and standard characters is where they are stored in memory.

## Sprite Graphics

Sprites are also preprogrammed shapes, but instead of having a built-in shape as the characters do, sprites are always designed by you. Sprites are very much like custom characters. They can be programmed using similar methods, and they have the same one-to-one correspondence between memory bits and dots on the screen. The biggest differences between sprites and characters are their size and the way they are positioned on the screen. Characters are eight bits by eight bits, while sprites are 24 bits wide by 21 bits high. Characters may be located in any of 1000 locations on the screen, while sprites can be in any of 64,000 positions on the screen.

## High-Resolution Screen Graphics

The high-resolution screen allows you to have bit-by-bit control of the entire screen, not just of individual characters or moveable sprites. This gives you the ability to create some very complex and specialized displays.

Just as designing sprites is a lot like designing custom characters, designing a high-resolution screen is a lot like designing a sprite. Once again, the biggest difference is its size. Sprites are 24 by 21 (504 bits), while the high-resolution screen is 320 by 200 (64,000 bits). This means that a single high-resolution screen requires 8000 bytes of memory — one-half of the entire 16K memory block the VIC-II chip can access.

## Coloring Characters: An Introduction to Color

One of the most obvious features of the 64 keyboard is the row of eight color keys, accessed by pressing CONTROL and the number keys 1 through 8. With these keys you can change the color of the characters as they are printed on the screen. To see how this works, press the CTRL key and one of the color keys simultaneously. Then type some characters. All of the characters entered after the color control key will be displayed in the new color. The

eight colors you can get using these keys are:

- CTRL-1: black**
- CTRL-2: white**
- CTRL-3: red**
- CTRL-4: cyan (blue green)**
- CTRL-5: purple**
- CTRL-6: green**
- CTRL-7: blue**
- CTRL-8: yellow**

**Eight more colors.** In addition, there are eight more colors available using the Commodore key (lower-left corner of the keyboard) and the numbered keys. They are:

- Commodore-1: orange**
- Commodore-2: brown**
- Commodore-3: light red**
- Commodore-4: dark grey**
- Commodore-5: medium grey**
- Commodore-6: light green**
- Commodore-7: light blue**
- Commodore-8: light grey**

Pressing any of these keys with the Commodore key will change the color of subsequent characters as they are printed on the screen.

## Programming the Color Control Keys

In addition to changing the character colors from the keyboard, you can modify the character colors from within your programs. The first method is simply to enter the quote mode (by pressing the quote key SHIFT-2) and then press the combination of CTRL or Commodore keys and numbered keys corresponding to the color you want PRINTed. Using this method, you can PRINT multicolored lines.

```
10 A$="{}RVS{}{ 40 {SPACES}}"  
20 A$=A$+A$  
30 DATA "{'BLACK'}","{'WHITE'}","{'RED'}","{'CYAN'}","{'PURPLE'}","{'GREEN'}","{'DK/GREY'}","{'YELLOW'}","{'MED/GREY'}","{'LT/GREY'}","{'LT/GREEN'}"  
40 FORO=0TO10  
50 READ C$(O)  
60 NEXT  
70 FORR=0TO10  
80 PRINTC$(R);A$;  
90 NEXT  
100 PRINT"{'LT/BLUE'}";
```

# Color

**How it works.** In lines 10 and 20, the program creates a string variable called A\$ that consists of 80 reverse spaces. By itself, it would be displayed as a wide horizontal bar. To see this, after running the program, enter PRINT A\$, and assuming you have not entered anything else since running the routine, the bar will be displayed.

Line 30 contains the color data, and lines 40 to 50 read the data and assign it to the variables C\$(0) through C\$(10).

Lines 70 to 90 PRINT the colored bars, and line 100 resets the character color to light blue.

To see the effect different colors have on this routine, you can substitute some or all of the colors in the DATA statement on line 30.

**Using CHR\$ to Program the Character Colors.** Just as every control function and character on the 64 has a CHR\$ value, so do the color control keys. Table 2-1 lists the CHR\$ values for the sixteen built-in colors.

**Table 2-1. CHR\$ Values of the Color Keys**

CHR\$ (ASCII) Value	Key	Color
CHR\$(144)	CTRL-1	Black
CHR\$(5)	CTRL-2	White
CHR\$(28)	CTRL-3	Red
CHR\$(159)	CTRL-4	Cyan (Blue/Green)
CHR\$(156)	CTRL-5	Purple
CHR\$(30)	CTRL-6	Green
CHR\$(31)	CTRL-7	Blue
CHR\$(158)	CTRL-8	Yellow
CHR\$(129)	Commodore-1	Orange
CHR\$(149)	Commodore-2	Brown
CHR\$(150)	Commodore-3	Light Red
CHR\$(151)	Commodore-4	Dark Grey
CHR\$(152)	Commodore-5	Medium Grey
CHR\$(153)	Commodore-6	Light Green
CHR\$(154)	Commodore-7	Light Blue
CHR\$(155)	Commodore-8	Light Grey

CHR\$ values can be substituted for the list of strings in the program above whenever it would be easier to use numeric data than string data.

# Color

```
10 A$="{}RV${}{ 40 {SPACES}}"
20 A$=A$+A$
30 DATA 144,5,28,159,156,30,155,158,152,151,153
40 FORO=0TO10
50 READ A
55 C$(O)=CHR$(A)
60 NEXT
70 FORR=0TO10
80 PRINTC$(R);A$;
90 NEXT
100 PRINT"{}LT/BLUE{}";
```

## Color Memory

In the two preceding programs, you were able to enter a single color command and all of the characters that followed that command were displayed in the new color. This function is actually a bit more complex than it seems. The computer's operating system keeps track of what color the characters should be and updates a block of memory called *color memory* each time a new character is entered.

Color memory and screen memory are like two independent grids. Each grid contains 1000 bytes which define 25 rows of 40 characters apiece. These two grids have a byte-for-byte, character-for-character correspondence. For example, the *shape* of the character in the upper left-hand corner of the screen is decided by the screen code in the *first* byte of screen memory, and the *color* of that character is decided by the color code in the first byte of color memory. (There is always a character displayed in every space — blanks contain the screen code 32, for the SPACE character.)

Each byte of screen memory contains the screen code for one of the characters displayed. Each byte of color memory contains the color code for one of the characters, in the same order. These are entirely separate from the character patterns stored in character memory. The same character pattern can be displayed in any of the 16 colors. The screen code decides *which* character pattern will be displayed; the color code decides what color the dots of the pattern will be.

**The screen editor.** When you enter characters from the keyboard, the part of the operating system that you are using the most is the *screen editor*. This routine handles all the information pertaining to displaying characters on the screen. When you press a key on the keyboard, the screen editor keeps track of where that character should be displayed (the *cursor position*) and what color it should have. Each time the screen editor is told to

# Color

display a character, it POKEs the character's screen code into the current cursor position in screen memory and also POKEs the most recent color code into the current cursor position in color memory.

**Changing colors in an existing screen.** Since color memory is located in one section of RAM (Read/Write Memory) and screen memory is located in another, you can change the colors any time you want without affecting the characters.

Color memory is between locations 55296 and 56295. Here is a routine that will fill the upper half of the screen with asterisks.

```
10 PRINT "[CLR]";  
20 FOR R=0 TO 479  
30 PRINT "*";  
40 NEXT
```

This will give us some room to experiment with color memory. After running this routine, you will have filled the first 480 locations (0-479) of color memory with the value 14. To confirm this, you can check those locations with:

```
10 FOR R=55296 TO 55775  
20 PRINT "[HOME]{ 14 {DOWN}}"; "[LEFT]"; (PEEK(R) AND  
15); "[LEFT]{ 3 {SPACES}}";  
30 NEXT
```

You may have noticed that in this routine, the value in color memory is always ANDed with 15. We do this because color memory is a little different from the rest of memory. All of the color memory RAM locations contain only four bits, not the normal eight. But the computer *thinks* there are eight bits, and with most 64s, if you try to read them, the upper four bits will be calculated as random values and will provide you with a false number. ANDing them with 15 causes the upper four bits to be ignored by the computer, resulting in an accurate reading of the color code.

**Checking the screen.** Color code 14 is light blue. To see this program work with a different color, CLEAR the screen by pressing RUN/STOP and RESTORE. Then RUN the program again. This time, the number displayed will be 6 (dark blue).

**POKEing colors onto the screen.** Enter the asterisk program again and RUN it so you will have a screen that is half asterisks and half clear. Remember that color memory begins at memory location 55296, so let's change the color of the first location, the upper-left asterisk on the screen.

**POKE 55296, 1**

# Color

**Color codes.** By POKEing different color codes into one of the memory locations between 55296 and 55775, you can change the color of the displayed characters. You can use any of the 16 available colors. Table 2-2 shows the color memory codes and their corresponding colors.

**Table 2-2. Color Memory Codes**

Color Code	Color
0	BLACK
1	WHITE
2	RED
3	CYAN (BLUE/GREEN)
4	PURPLE
5	GREEN
6	BLUE
7	YELLOW
8	ORANGE
9	BROWN
10	LIGHT RED
11	DARK GREY
12	MEDIUM GREY
13	LIGHT GREEN
14	LIGHT BLUE
15	LIGHT GREY

## Managing Character Colors

Having a separate block of memory to control the color of the displayed characters provides you with a great deal of flexibility in creating colorful displays, but it also means that you must coordinate both memories whenever you are POKEing characters onto the screen display. If you do not, you may run into some very unpredictable display problems.

**The mysterious nonappearing characters.** Filling the screen with @'s using the POKE command should be no more complicated than POKEing the value for the @ symbol into every memory location in screen memory. Enter this program, and then, without clearing the screen, RUN it.

```
10 FOR R=1024 TO 2023
20 POKE R,0
30 NEXT
```

On some of the earlier 64s, some of the @'s will be displayed in light blue and the rest will be white. Newer 64s will only display those @'s that are in locations originally filled with displayed

# Color

characters. This is because the screen editor POKEd the appropriate color values only into those locations that had characters displayed in them when you had the program listed on the screen. The remaining locations contain either a 1 (on older machines), which causes the characters to be displayed in white, or a 6 (on newer machines), which causes the characters to be displayed in the background color, making them effectively invisible. So to display all of the characters that you POKE into screen memory properly, you will also need to POKE a contrasting color into each screen location's corresponding color location.

```
10 FOR R=1024 TO 2023  
20 POKE R,0  
25 POKE R+54272,14  
30 NEXT
```

**Designing color overlays.** While the details of updating color memory may seem like a bit of a bother, they also open the door to some interesting design possibilities. One of these, I call color overlays. These are colors that are added to a screen that has been previously filled with characters. The colors make the images appear.

```
10 FOR R= 1024 TO 2023  
20 POKE R+54272,14  
30 POKE R,81  
40 NEXT  
50 P=55296  
60 FOR C=0 TO 255  
70 FOR R=0 TO 999 STEP 41+C  
80 POKE P+R, C  
90 NEXT  
100 FOR R=0 TO 999 STEP 41+C  
110 POKE P+R, 14  
120 NEXT  
130 NEXT
```

The routine above fills the screen with 1000 stationary dots. The program then displays 255 different colored overlays in sequence. Although the images on the screen do not change, the overlays give the screen the appearance of a flashing sign.

**Hiding characters on the screen.** You can also use this method to "hide" characters on the screen by displaying them in the same color as the background. Changing the character to a contrasting color will effectively "uncover it."

**Mix and match.** This routine is a simple game that uses the color overlay technique. First, the computer displays a four-by-

four grid on the screen. It then generates a pattern using eight pairs of characters. The characters are displayed at random locations within the grid. Initially, the characters are displayed in the same color as the screen background, so they are invisible (the background color overlays the images).

The player then enters the position of any square, based on its coordinates. The squares in the grid are numbered 1 through 4 across the top and A through D along the side. To find the coordinates for any given square, look for the number at the top of its column and the letter at the left end of its row. For example, the upper-left square is A1 and the lower-right square is D4. Always enter the letter first and the number second. (See Figure 2-1.)

**Figure 2-1. An Overlay Grid**

A1

	1	2	3	4
A				
B				
C				
D				D4

After you enter the coordinates, the corresponding character on the grid will be lit. This is accomplished by POKEing a contrasting color into its color memory location, making the hidden character appear.

```
5 DIM A(15),B(15),C(15)
6 REM
7 REM ***** SET UP DISPLAY GRID *****
8 REM
10 PRINT "{CLR}{ 2 {DOWN}}{ 16 {SPACES}}1 2 3 4
20 PRINT "{ 15 {SPACES}}{A}*{R}*{R}*{R}*{S}
30 PRINT "{ 13 {SPACES}})A B B B B
40 PRINT "{ 15 {SPACES}}{Q}*+*+*+*{W}
50 PRINT "{ 13 {SPACES}})B B B B B
```

# Color

```
60 PRINT" { 15 {SPACES}}{Q}*+*+*+*{W}
70 PRINT" { 13 {SPACES}}C B B B B B
80 PRINT" { 15 {SPACES}}{Q}*+*+*+*{W}
90 PRINT" { 13 {SPACES}}D B B B B B
100 PRINT" { 15 {SPACES}}{Z}*{E}*{E}*{E}_*{X}
110 PRINT" {DOWN}{ 40 _}"
111 REM
112 REM ***** DEFINE GRID POSITIONS *****
113 REM
120 DATA 176,178,180,182,256,258,260,262,336,338,
      340,342,416,418,420,422
130 FOR R=0 TO 15
140 READ N
150 A(R)=N+1024
160 B(R)=N+55296
170 NEXT
180 FOR R=0 TO 15
190 POKE B(R), 6
200 NEXT
201 REM
202 REM ***** DEFINE SYMBOLS *****
203 REM
210 DATA 81,81,83,83,87,87,88,88,90,90,127,127,86
      ,86,91,91
220 FOR C=0 TO 15
230 L=INT(RND(0)*16)
240 IF C(L)=1 THEN 230
250 C(L)=1
260 READ N
270 POKE A(L),N
280 NEXT
281 REM
282 REM ***** GET LETTER COORDINATE *****
283 REM
290 PRINT" {HOME}{ 16 {DOWN}}{ 6 {SPACES}}ENTER PO
      SITION OF SQUARE{ 4 {SPACES}}{ 3 {LEFT}}";
295 GET LS:IF LS="" THEN 290
300 PRINT LS;:K=K*-1
305 IF LS="A" THEN 400
310 IF LS="B" THEN 500
320 IF LS="C" THEN 600
330 IF LS="D" THEN 700
340 GOTO 290
341 REM
342 REM ** GET NUMBER FOR 'A' ENTRY **
343 REM
400 GET NS:IF NS="" THEN 400
405 N=VAL(NS)
410 IF N<1 OR N>4 THEN 400
```

```
420 PRINTN$;
430 ON N GOTO 440,450,460,470
440 POKE 55472,14: GOTO 290
450 POKE 55474,14: GOTO 290
460 POKE 55476,14: GOTO 290
470 POKE 55478,14: GOTO 290
471 REM
472 REM ** GET NUMBER FOR 'B' ENTRY **
473 REM
500 GET N$:IFN$="" THEN 500
505 N=VAL(N$)
510 IF N<1 OR N>4 THEN 500
520 PRINTN$;
530 ON N GOTO 540,550,560,570
540 POKE 55552,14: GOTO 290
550 POKE 55554,14: GOTO 290
560 POKE 55556,14: GOTO 290
570 POKE 55558,14: GOTO 290
571 REM
572 REM ** GET NUMBER FOR 'C' ENTRY **
573 REM
600 GET N$:IFN$="" THEN 600
605 N=VAL(N$)
610 IF N<1 OR N>4 THEN 600
620 PRINTN$;
630 ON N GOTO 640,650,660,670
640 POKE 55632,14: GOTO 290
650 POKE 55634,14: GOTO 290
660 POKE 55636,14: GOTO 290
670 POKE 55638,14: GOTO 290
671 REM
672 REM ** GET NUMBER FOR 'D' ENTRY **
673 REM
700 GET N$:IFN$="" THEN 700
705 N=VAL(N$)
710 IF N<1 OR N>4 THEN 700
720 PRINTN$;
730 ON N GOTO 740,750,760,770
740 POKE 55712,14: GOTO 290
750 POKE 55714,14: GOTO 290
760 POKE 55716,14: GOTO 290
770 POKE 55718,14: GOTO 290
```

## Sprite Colors

When you first turn on the sprites, they are each assigned a different color.

```
10 PRINT"{}CLR{}{}WHITE{}{} 3 {}DOWN{}{}"
20 POKE 53281,0
```

# Color

```
30 POKE 53269, 255
40 DATA 30,60,60,60,90,60,120,60,150,60,180,60,21
     0,60,240,60
50 FOR R=53248 TO 53263
60 READ A
70 POKE R,A
80 NEXT
90 FOR R=2040 TO 2047
100 POKE R,50
110 NEXT
120 FOR R=3200 TO 3263
130 POKE R, 255
140 NEXT
```

These default colors are:

- Sprite #0 white
- Sprite #1 red
- Sprite #2 light green
- Sprite #3 purple
- Sprite #4 green
- Sprite #5 blue
- Sprite #6 yellow
- Sprite #7 medium grey

## Sprite Color Control Registers

The sprites can be displayed in any of the 16 colors available to characters. To change the color of a sprite, simply POKE a different number into that sprite's color control register. (See Table 2-3.)

**Table 2-3.**

Sprite	Register location
#0	53287
#1	53288
#2	53289
#3	53290
#4	53291
#5	53292
#6	53293
#7	53294

The bits in the sprite that are *on* will be displayed in the color specified by that sprite's color register. The *off* bits will be transparent and the screen display will show through.

# Color

## Using Priority for More Color

Although you can do a lot using the methods outlined above, sometimes you just need to have more color on the screen.

In Chapter 1, we learned that sprites have a number of special capabilities that characters do not. One of them is the ability to be displayed in front of or behind characters, the background, or other sprites. This provides us with a very simple and versatile method of creating high-resolution sprites using as many as sixteen colors at the same time.

Here is a typical high-resolution sprite that uses one color for its image.

```
10 POKE 2040, 50
20 PRINT "[CLR]{ 3 {DOWN}}"
30 DATA 255,255,255,128,0,1,128,0,1,128,0,1,128,0
   ,128,0,1,128,0,1,128,0,1
40 DATA 128,255,1,128,255,1,128,255,1,128,255,1,1
   28,255,1,128,255,1,128,0,1
50 DATA 128,0,1,128,0,1,128,0,1,128,0,1,128,0,1,2
   55,255,255
60 FOR R=3200 TO 3262
70 READ A
80 POKE R,A
90 NEXT
100 POKE 53287,14
110 POKE 53248,170
120 POKE 53249,135
130 POKE 53269,255
```

By positioning a second sprite behind the first, we can get a single high-resolution two-color sprite.

```
10 POKE 2040, 50
20 PRINT "[CLR]{ 3 {DOWN}}"
30 DATA 255,255,255,192,0,3,192,0,3,192,0,3,192,0
   ,3,192,0,3,192,0,3,192,0,3
40 DATA 192,255,3,192,255,3,192,255,3,192,255,3,1
   92,255,3,192,255,3,192,0,3
50 DATA 192,0,3,192,0,3,192,0,3,192,0,3,192,0,3,2
   55,255,255
60 FOR R=3200 TO 3262
70 READ A
80 POKE R,A
90 NEXT
100 POKE 53287,14
110 POKE 53248,170
120 POKE 53249,135
130 POKE 53269,255
140 DATA 0,0,0,63,255,252,48,0,61,48,0,61,48,0,61
   ,48,0,61,48,0,61,48,0,61,48
```

# Color

```
150 DATA 0,61,48,0,61,48,0,61,48,0,61,48,0,61,48,  
    0,61,48,0,61,48,0,61,48,0,61  
160 DATA 48,0,61,63,255,252  
180 POKE 2041, 51  
190 FOR R=3264 TO 3320  
200 READ A  
210 POKE R,A  
220 NEXT  
230 POKE 53288,2  
240 POKE 53250,170  
250 POKE 53251,135  
260 POKE 53269,3
```

And by placing another sprite behind the second, we can obtain yet another color.

```
5 REM ***** FIRST SPRITE *****  
6 REM  
10 POKE 2040, 50  
20 PRINT"[CLR]{ 3 {DOWN}}"  
30 DATA 255,255,255,192,0,3,192,0,3,192,0,3,192,0  
    ,3,192,0,3,192,0,3,192,0,3  
40 DATA 192,255,3,192,255,3,192,255,3,192,255,3,1  
    92,255,3,192,255,3,192,0,3  
50 DATA 192,0,3,192,0,3,192,0,3,192,0,3,192,0,3,2  
    55,255,255  
60 FOR R=3200 TO 3262  
70 READ A  
80 POKE R,A  
90 NEXT  
100 POKE 53287,14  
110 POKE 53248,170  
120 POKE 53249,135  
130 POKE 53269,255  
135 REM  
136 REM ***** SECOND SPRITE *****  
137 REM  
140 DATA 0,0,0,63,255,252,48,0,61,48,0,61,48,0,61  
    ,48,0,61,48,0,61,48,0,61,48  
150 DATA 0,61,48,0,61,48,0,61,48,0,61,48,0,61,48,  
    0,61,48,0,61,48,0,61,48,0,61  
160 DATA 48,0,61,63,255,252  
180 POKE 2041, 51  
190 FOR R=3264 TO 3320  
200 READ A  
210 POKE R,A  
220 NEXT  
230 POKE 53288,2  
240 POKE 53250,170  
250 POKE 53251,135
```

```
260 POKE 53269,3
265 REM
266 REM **** THIRD SPRITE ****
267 REM
270 POKE 2042, 52
280 FOR R=3328 TO 3391
290 POKE R,255
300 NEXT
310 POKE 53289,1
320 POKE 53252,170
330 POKE 53253,135
340 POKE 53269,7
```

## Blended Colors

This next method allows you to blend colors in much the way an artist takes two colors on a palette and mixes them to create a new color. This lets you go beyond the 16 colors built into the 64.

**Checkerboard dots.** There are a number of methods you can use to blend the colors of sprites, and they each have one thing in common: they place the dots of color that are to be blended very close together, preferably on adjacent lines.

Figure 2-2 represents a sprite with the simplest kind of pattern used for blending colors. Each of the small squares represents a single dot in the sprite. The squares with X's in them are on; they will be displayed as the sprite color. The open squares will show through whatever is behind them.

Here is a blended color routine that slides a pure, colored sprite behind a sprite that has a grid pattern.

```
10 POKE 2040, 50
20 POKE 53281, 15
30 PRINT "[CLR]{ 3 {DOWN}}"
40 DATA 170,170,170,85,85,85,170,170,170
50 FOR L=0TO63 STEP 9
60 FOR R=3200 TO 3208
70 READ A
80 POKE R+L,A
90 NEXT: RESTORE
100 NEXT
110 POKE 53248,170
120 POKE 53249,135
130 POKE 53269,1
140 POKE 2041, 51
150 FOR R=3264 TO 3327
160 POKE R, 255
170 NEXT
180 POKE 53250,170
```

# Color

```
185 POKE 53251,135
190 POKE 53251,135
200 POKE 53269,3
210 FOR B=1 TO 15
220 POKE 53288, B
230 FOR L=0 TO 14
240 POKE 53287, L
250 FOR R=170 TO 200
260 POKE 53250,R
270 NEXT
280 FOR G=0 TO 500
290 NEXT
300 FOR R=200 TO 170 STEP -1
310 POKE 53250,R
320 NEXT
330 FOR G=0 TO 500
340 NEXT
350 NEXT
360 NEXT
```

**Reverse color effect.** Putting a solid block of color A behind a grid of color B does not produce the same effect as putting a solid block of color B behind a grid of color A. In other words, placing a solid red sprite behind a green grid will not produce the same color as a red grid placed over a solid green sprite. This is because of the way televisions are made. This method provides us with a total of 256 different colors.

Finer color gradations can be achieved by making a more complex grid pattern and stacking up more than two sprites. Figure 2-3 shows a color pattern that allows us to mix colors even more finely. By designing overlapping sprites that fill the individual squares (either A,B,C, or D) you can get 4096 different colors and shadings — though few people would be able to detect some of the subtle differences.

## Shading

You can use multiple sprites to create an illusion of shading. By putting lines from different sprites next to each other, you can produce some very convincing effects. Here, for example, is a cylinder.

```
10 POKE 2040, 50
20 PRINT"CLR}{ 3 {DOWN}}"
30 DATA164,66,37,164,66,37,164,66,37
40 FOR L=0TO63 STEP 9
50 FOR R=3200 TO 3208
60 READ A
```

**Figure 2-2. Sprite with a Simple Checkerboard Pattern**

## **Color**

## **Figure 2-3. Sprite Four-Color Checkerboard Pattern**

# Color

```
70 POKE R+L,A  
80 NEXT: RESTORE  
90 NEXT  
110 POKE 53248,170  
120 POKE 53249,135  
140 POKE 2041, 51  
150 FOR R=3264 TO 3327  
160 POKE R, 255:NEXT  
180 POKE 53250,170  
185 POKE 53251,135  
190 POKE 53251,135  
200 POKE 53269,3  
220 POKE 53288, 1  
240 POKE 53287, 0  
250 POKE53271,3
```

You should be aware, however, that the dots on the screen are taller than they are wide. So this kind of shading works much better on tall objects. The same pattern displayed horizontally will not be quite as effective as it is vertically.

```
10 POKE 2040, 50  
20 PRINT"{}CLR{} 3 {DOWN}{}"  
25 DATA 255,255,255  
30 DATA 0,0,0,255,255,255,0,0,0,255,255,255,0,0,0  
     ,0,0,0,255,255,255,0,0,0,0,0,0,0  
40 DATA255,255,255,0,0,0,0,0,0,0,0,0,255,255,255,  
     0,0,0,0,0,0,255,255,255,0,0,0  
45 DATA 255,255,255,0,0,0,255,255,255  
50 FOR R=3200 TO 3263  
60 READ A  
70 POKE R,A  
80 NEXT  
110 POKE 53248,170  
120 POKE 53249,135  
140 POKE 2041, 51  
150 FOR R=3264 TO 3327  
160 POKE R, 255:NEXT  
180 POKE 53250,170  
185 POKE 53251,135  
190 POKE 53251,135  
200 POKE 53269,3  
220 POKE 53288, 1  
240 POKE 53287, 0  
250 POKE53277,3
```

## Screen Colors

How is color handled on the high-resolution screen? The screen gets its shapes from the bitmap, an 8000-byte block of memory

# Color

that we must designate when we set up for bitmapped mode. This frees up the thousand bytes of screen memory starting at location 1024. This memory is now used to individually control the *colors* of the bitmap. The regular color memory at 55296-56319 is not used.

Each byte of screen memory corresponds to a cell (eight-byte block) of the bitmap. Unlike color memory, which only has the lower nibble (bits 0-3) available, screen memory has eight bits per byte. Since it takes only four bits to hold the color value (0000 to 1111, or 0 to 15 decimal), each location in screen memory can separately control *two* colors. And that's exactly how the 64 handles color in the bitmapped mode. (See Figure 2-4.)

**Figure 2-4. High-Resolution Color Memory Byte**

High-Resolution Color Memory Byte	Color of ON Bits				Color of OFF Bits			
	128	64	32	16	8	4	2	1

The upper nibble (left four bits) of each screen memory location controls a cell's foreground color (those bits which are 1, or *on*). The lower nibble (right four bits) of each screen memory location controls a cell's background color (those bits which are 0, or *off*).

In other words, for each cell in the bitmap, there is a byte in screen memory that controls both the foreground and background colors in that cell. If that number were 1101 0110 (decimal 214), the upper nibble (1101) controls the foreground color and the lower nibble (0110) controls the background color. In this case, the foreground would be light green (13) and the background would be blue (6).

To turn the value for light green, 13 (binary 1101), into an upper nibble value, just multiply it by 16. To set one cell to a light green foreground and a green (5) background, you would use this statement:

**POKE SM,5 +16\*13**

The general formula is:

**POKE screen memory, background +16\* foreground**

(See Appendix E for a complete listing of low and high nibble color values.)

## Color

To see this in action, here is a routine that sets up a bit-mapped screen divided into three sections. The top section consists of alternating *on* and *off* dots, which will blend the foreground and background colors. The middle section consists entirely of *off* (0) dots, so that only the background color will be displayed. The last section is *on* (1) dots, displaying the foreground color.

```
10 POKE53272,29:POKE53265,59:POKE56576,198:A=1
20 FOR R=17408TO18407:POKE R,A:NEXT
30 FOR R=24576 TO 27135STEP2
40 POKE R,85:POKER+1,170::NEXT
50 FOR R=27136 TO 30015:POKE R,0:NEXT
60 FORR=30016 TO 32575:POKE R,255:NEXT
70 FOR M=0TO255
80 FOR R=17408TO18407:POKE R,M:NEXT
90 NEXT
```

The program then cycles through all 256 color combinations. You can see how the various colors mix on the screen. In each case, mixing the two lower colors will produce the upper one.

## **Adding Color to the High-Resolution Screen Editor**

Here is a modified version of the screen editor that allows you to change the color of a line as it is drawn on the screen.

To change the line colors, press one of the color keys along with the CTRL key or the Commodore key. All lines drawn thereafter, until the next color change, will be displayed in the new color — including the line currently being drawn, if you press the color key during the drawing process.

To change the background color, press the key combination for the color you want (while a line is *not* being drawn) and then press f1. The entire background will change. You'll notice that the function key diagram at the lower right will be changed automatically to contrast with the new background.

## **Program 2-1. Color Screen Editor**

```
1 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
     15,255,240,12,62,240,13,252  
2 DATA 240,12,126,240,141,254,241,141,252,113,143  
     ,255,241,128,0,1,140,62,113  
3 DATA 157,253,185,188,127,125,253,254,255,253,25  
     2,63,255,255,255,0,0,0,0,0,0  
4 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15,255,240  
     ,12,60,240,13,251,112,12,126
```

# Color

```
5 DATA 240,141,251,113,141,252,241,143,255,241,12
8,0,1,140,60,241,157,250,249
6 DATA 188,112,125,253,254,255,253,254,255,255,25
5,255,0,0,0,0,0,0,0,0,0,0,0,0
7 DATA 0,0,0,0,0,0,0,0,0,0,0,15,255,240,12,60,48,
13,253,240,12,124,112,141,255
8 DATA 177,141,252,113,143,255,241,128,0,1,140,62
,113,157,253,249,188,124,125
9 DATA 253,253,191,253,254,127,255,255,255,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0
10 DATA 0,0,0,0,0,15,255,240,12,60,48,13,255,176,
12,127,112,141,254,241,141,254
11 DATA 241,143,255,241,128,0,1,140,62,113,157,25
3,185,188,126,125,253,253,191
12 DATA 253,254,127,255,255,255,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,219,187,164,146
13 DATA 187,180,83,34,44,218,187,164,0,0,0,0,221,59
,130,149,42,191,149,43,2,221
14 DATA 186,128,0,3,164,0,2,180,173,90,172,205,21
0,164,168,138,191,172,154,182
15 DATA 0,3,164,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,247,24,239
16 DATA 132,165,8,199,60,204,132,164,40,244,165,2
07,0,0,0,0,0,231,25,16,148
17 DATA 165,16,151,61,80,148,165,80,228,164,160,0
,0,0,0,0,0,0,0,0,0,0,0,0,0
18 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,100,164,206,138
,170,168,74,174,204,44,170,168
19 DATA 198,74,174,0,0,0,0,0,0,116,57,156,132,34,
82,132,51,220,132,34,84,119
20 DATA 186,82,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
21 DATA 240,130,100,128,130,84,224,130,76,128,242
,68,240,0,0,0,0,0,0,117,199
22 DATA 71,133,40,68,133,200,70,133,40,68,117,39,
119,0,0,0,0
23 DIM JH(255),SZ(16):ZT=1
24 JH(144)=1:JH(5)=2:JH(28)=3:JH(159)=4:JH(156)=5
:JH(30)=6:JH(31)=7:JH(158)=8
25 JH(129)=9:JH(149)=10:JH(150)=11:JH(151)=12:JH(
152)=13:JH(153)=14:JH(154)=15
26 JH(155)=16
30 FOR R=16384 TO 16895:READ A:POKE R,A: NEXT
31 SZ(0)=1:SZ(1)=6:SZ(2)=15:SZ(3)=11:SZ(4)=2:SZ(5
)=13:SZ(6)=14:SZ(7)=9:SZ(8)=7
32 SZ(9)=8:SZ(10)=0:SZ(11)=12:SZ(12)=11:SZ(13)=5:
SZ(14)=6:SZ(15)=0
35 POKE 53264,15:NC=14
40 POKE53271,255:POKE53277,255
```

```
50 FOR R=7 TO 0 STEP -1:POKE18424+R,R:NEXT:POKE 5
    3269,255
60 DATA 36,102,36,136,36,170,36,205,231,102,231,1
    36,231,170,231,205
80 FORR=53248TO53263:READA:POKER,A:NEXT
90 FOR R=53287 TO 53294: POKE R,14:NEXT
100 PRINT"[CLR]";:X=160:Y=100
110 POKE53272,29:POKE53265,59:POKE56576,198
120 FOR R=17408TO18407:POKE R,230:NEXT
125 IF W=1THENW=0:GOTO140
130 FOR R=24576 TO 32767:POKE R,0:NEXT
140 GET A$:IF A$=""THEN 140
141 IF JH(ASC(A$))<>0THEN NC=JH(ASC(A$))-1
142 GOSUB 5000
145 IF A$="{F-1}" THEN FOR R=17408TO18407:POKER,(PEEK(R)AND240)ORNC:NEXT
146 IF A$="{F-1}" THEN FOR R= 53287TO53294:POKER,
    SZ(NC):NEXT:GOTO 140
147 IF A$="{F-5}" THEN S=1
148 IF A$="C" AND S=2 THEN X2=X:Y2=Y:S=0:GOTO 310
149 IF A$="C" AND S=1 THEN X1=X:Y1=Y:S=2:GOTO500
150 IF A$="{UP}" THEN Y=Y-1:GOTO 200
155 IF A$="{F-2}" ANDZT=1THEN POKE 53269,0:ZT=ZT*
    -1:GOTO140
156 IF A$="{F-2}" ANDZT=-1THEN POKE 53269,255:ZT=
    ZT*-1:GOTO140
157 IF A$="{F-7}" THEN L=1
158 IF A$="C" AND L=1 THEN X1=X:Y1=Y:L=2:GOTO500
159 IF A$="C" AND L=2 THEN X2=X:Y2=Y:L=0:GOTO700
160 IF A$="{DOWN}" THEN Y=Y+1:GOTO 200
165 IF A$="{F-3}" THEN M=1
167 IF A$="{F-8}" THEN C=1
168 IF A$="C" AND C=1 THEN X1=X:Y1=Y:C=2:GOTO500
169 IF A$="C" AND C=2 THEN X2=X:Y2=Y:C=0:GOTO900 .
170 IF A$="{RIGHT}" THEN X=X+1:GOTO 200
175 IF A$="{F-4}" THEN M=0
180 IF A$="{LEFT}" THEN X=X-1:GOTO 200
185 IF A$="{F-6}" THEN 130
190 GOTO 140
200 P=(24576+(INT(Y/8))*320+(INT(X/8))*8+(YAND7))
210 POKEP,PEEK(P)OR2↑(7-(XAND7)):IFM=0THEN140
220 POKEP,PEEK(P)AND(255-(2↑(7-(XAND7)))):GOSUB50
    00:GOTO140
310 Y=Y1
320 FOR X=X1 TO X2 STEP(X1>X2OR1):GOSUB400:NEXT
330 FOR Y=Y1 TO Y2 STEP(Y1>Y2OR1):GOSUB400:NEXT
340 FOR X=X2 TO X1 STEP(X2>X1OR1):GOSUB400:NEXT
350 FOR Y=Y2 TO Y1 STEP(Y2>Y1OR1):GOSUB400:NEXT:
    GOTO 140
```

# Color

```
400 P=(24576+(INT(Y/8))*320+(INT(X/8))*8+(YAND7))
410 POKEP,PEEK(P)OR2↑(7-(XAND7)):GOSUB5000:RETURN
500 P=(24576+(INT(Y/8))*320+(INT(X/8))*8+(YAND7))
510 POKEP,PEEK(P)OR2↑(7-(XAND7)):GOSUB5000:GOTO14
    0
700 Y=Y1
710 SP=(ABS(Y1-Y2))/(ABS(X1-X2))
720 FORX=X1 TO X2 STEP.3*(X1>X2OR1):Y=Y+(SP*.3*(Y
    1>Y2OR1)):GOSUB400:NEXT:GOTO140
900 R1=SQR(((X1-X2)↑2)+((Y1-Y2)↑2)):R2=R1*.72
910 FORT=0TO2*{↑}STEP(1.5/R1):X=(R1*(COS(T)))+X1:
    Y=(R2*(SIN(T)))+Y1:GOSUB400:NEXT
920 GOTO 140
1000 REM
1010 REM ***** SAVE SCREEN ROUTINE *****
1020 REM
1030 INPUT"[CLR]{ 2 {DOWN}}NAME OF SCREEN";SN$
1040 INPUT"{ 2 {DOWN}}SAVE ON DISK OR CASSETTE (D
    /C)";DC$
1050 IF DC$="D"THEN 1080
1060 IF DC$="C"THEN 1090
1070 INPUT"[UP]PLEASE ENTER D OR C{ 12 {SPACES}}"
    ;DC$
1080 OPEN1,8,4,"@:"+SN$+".SCN,W":GOTO1100
1090 OPEN1,1,1,SN$+".SCN"
1100 FOR R=24576 TO 32767:PRINT#1,PEEK(R):NEXT:
    CLOSE 1:GOTO 30
2000 REM
2010 REM ***** LOAD SCREEN ROUTINE *****
2020 REM
2030 INPUT"[CLR]{ 2 {DOWN}}NAME OF SCREEN";SN$
2040 INPUT"{ 2 {DOWN}}LOAD FROM DISK OR CASSETTE
    (D/C)";DC$
2050 IF DC$="D"THEN 2080
2060 IF DC$="C"THEN 2090
2070 INPUT"[UP]PLEASE ENTER D OR C{ 12 {SPACES}}"
    ;DC$
2080 OPEN1,8,4,"@:"+SN$+".SCN,R":GOTO2100
2090 OPEN1,1,0,SN$+".SCN"
2100 FOR R=24576 TO 32767:INPUT#1,H:POKER,H:NEXT:
    CLOSE 1:W=1:GOTO 30
5000 NB=((INT(Y/8))*40)+(INT(X/8)+17408)
5010 POKE NB,(PEEK(NB)AND15) OR (NC*16):RETURN
```

## Multicolor Mode

While you can create many different tints and hues by overlapping and mixing colors, you can't have more than two colors within a single character or cell. This means that, in effect, while

# Color

your line resolution can be as fine as a single dot, your color resolution is eight bits high and eight bits wide — not very precise!

Multicolor mode, which is available for both the text screen and the bitmapped screen, solves the problem by allowing you to have up to four colors in a single character or cell. There is a price, however. Your line resolution is now two dots wide (though it is still one dot high), which makes standard characters in multicolor text mode almost unreadable. So for extremely accurate fine lines or readable text, the standard mode is better. But for custom character sets and really effective color displays, multicolor text mode and multicolor bitmapped mode are excellent alternatives.

## Entering Multicolor Mode

To switch into either multicolor mode, you will need to turn on bit 4 of register 53270:

**POKE 53270, PEEK(53270) OR 16**

To switch out of multicolor mode, enter this line:

**POKE 53270, PEEK(53270) AND (255-16)**

or simply press RUN/STOP-RESTORE.

## Bit-Pairs and Double Dots

With both the bitmapped and text screens, patterns have been drawn with *on* and *off* bits. A 1 always placed a single dot of the foreground color on the screen, and a 0 always placed a single dot of the background color. However, using one bit to paint one dot allows only two color choices. It can't work for multicolor mode.

Multicolor mode, therefore, does not use a single bit for a color instruction. It uses a bit-pair for each instruction. In regular mode, a single byte would include eight *on-off* color instructions. The binary number 10101010 (decimal 170) would look like Figure 2-5.

**Figure 2-5. A High-Resolution Byte**

1	0	1	0	1	0	1	0
F	B	F	B	F	B	F	B

F = foreground color, B = background color

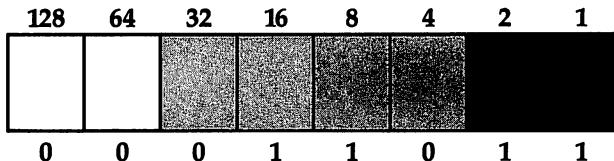
# Color

In multicolor mode, each byte includes only four color instructions, but each color instruction is two bits long. Think of the byte as four bit-pairs, like this: 10 10 10 10.

Each bit-pair has four possible values: 00, 01, 10, 11. When multicolor mode has been enabled, the VIC-II chip expects to read bytes as groups of four bit-pairs instead of eight bits.

However, using bit-pairs creates another problem. There are now only four color instructions per byte instead of eight. We have doubled the number of available colors, but cut in half the number of dots that can be separately controlled. The solution is that now each bit-pair in a pattern controls two dots, instead of one. Each eight-bit byte still controls eight dots on the screen. But you can't split a double dot in multicolor mode. Horizontal resolution has been cut in half. Compare Figure 2-6 with Figure 2-5 to see the difference.

**Figure 2-6. A Multicolor Byte**



## Multicolor Text Mode

With multicolor text mode, the shapes still come from the character set. Let's look at how the bit-pairs and double dots alter the appearance of a character. Ordinarily, the asterisk (\*) is displayed on the screen as shown in Figure 2-7. The white areas (*off* bits) in the illustration would be displayed in the background color and the black areas (*on* bits) in the foreground color.

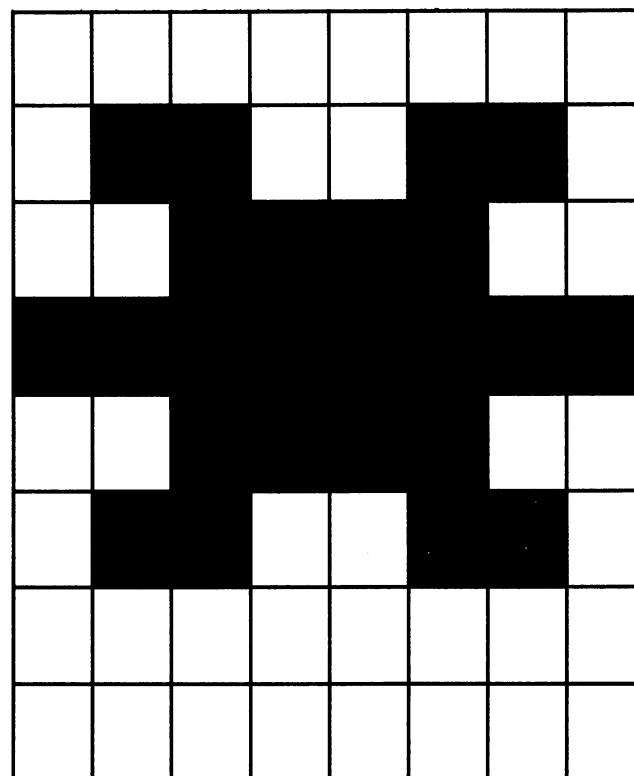
In multicolor mode, the display changes, as shown in Figure 2-8. The areas shown in white (00 bit-pairs) are the background color, the light grey areas are the first auxiliary color (01 bit-pairs), the dark grey areas are the second auxiliary color (10 bit-pairs), and the black areas are the foreground color (11 bit-pairs).

To show this in action, here is a program that will fill the screen with asterisks and then toggle between standard color mode and multicolor mode.

```
10 PRINT "{CLR}{RVS}" ;
20 FOR R=0 TO 998
30 PRINT "*";
```

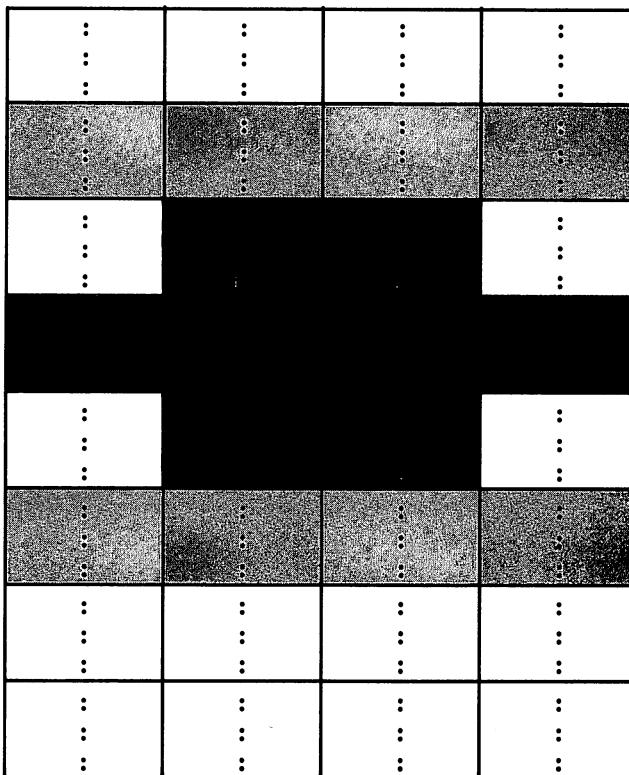
```
40 NEXT
50 PRINT "[HOME]";
60 FOR R=0 TO 1023
70 POKE R+55296, R AND 15
80 NEXT
90 POKE 53270, 216
100 FOR G=0 TO 500
110 NEXT
120 POKE 53270, 200
130 FOR G=0 TO 500
140 NEXT
150 GOTO 90
```

**Figure 2-7. Displayed Bits of \* in Standard Color Mode**



# Color

**Figure 2-8. Displayed Bits of \* in Multicolor Mode**



**Selecting Background, Auxiliary 1, and Auxiliary 2.** In standard text mode, the foreground color comes from color memory and the background color from location 53281. In multicolor text mode, the background color still comes from 53281. The first auxiliary color comes from 53282 and the second auxiliary color from 53283. Therefore, the background and the two auxiliary colors are the same for every character position on the screen — they cannot be individually assigned. Regular low nibble color values are POKEd into these locations to assign one of the colors from 0 to 15. (See Appendix E for the low nibble color values.)

**Having it both ways in color memory.** As with standard text mode, the foreground color (11 bit-pairs) is assigned in color memory (55296-56319), and each character position on the screen is individually controlled by a single byte in color memory.

# Color

However, because the standard character set is almost completely unreadable in multicolor mode, the system has been designed to give you the option of switching multicolor mode on or off for each character position individually. That means that once multicolor mode has been enabled at location 53270, you still have the choice of switching only certain characters to multicolor mode, and leaving other characters in standard text mode.

This feature is controlled in color memory. Each byte in color memory has only the lower four bits (bits 0-3). The highest or left-most of these, bit 3, is used as a toggle to switch back and forth between multicolor and standard text mode. If bit 3 is *on* (1), then that character will be displayed in multicolor mode; if bit 3 is *off* (0), then that character will be in standard mode.

This means, however, that we only have three bits left for color selection. We can only choose colors from 0 to 7, the colors selected from the keyboard by pressing CTRL-1 through CTRL-8. In multicolor mode, then, the foreground color can only be a color from 0 to 7, but it can be individually selected for each character, while the background and two auxiliary colors can be colors from 0 to 15, but must be the same for the whole screen.

How do you calculate the number to POKE into color memory? First, choose the color you want, from 0 to 7. If you want standard text mode, just POKE that number into color memory. If you want multicolor text mode for that character, you must OR it with 8 (or add it to 8 — in this case it makes no difference). In practice this means that for standard mode you POKE in the numbers 0-7 to get colors 0-7, and for multicolor mode you POKE in the numbers 8-15 to get colors 0-7. (See Table 2-4.)

**Mixing modes from the keyboard.** This also works from the keyboard. Enter this line:

**POKE 53270,PEEK(53270)OR 16**

You are now in multicolor mode. Now type something on the keyboard. Hard to read, isn't it? (But you can still type commands and list programs, the same as ever — your computer doesn't care what the character looks like, just what the code number is in screen memory.)

Now, press CTRL plus any number from 1 to 8. What happens to text you type in now? Unless you selected the background color, you'll see standard characters.

To get back to multicolor mode, press Commodore plus any number from 1 to 8. You'll see multicolor mode again — but the

# Color

foreground color won't be the color you would usually get from pressing Commodore and the number key you chose. Instead, it will be the color you would usually get by pressing CTRL and that number key. To see it more clearly, get into reversed character mode and press the space bar in both multicolor and text mode. This will put blocks of pure foreground color on the screen, and you can compare colors more easily.

Table 2-4 lists this process concisely. You may want to refer to it during multicolor text programming.

## Table 2-4. Setting Up Multicolor Characters

1. Turn on multicolor mode: POKE 53270, PEEK(53270)OR 16
2. Decide which locations will be multicolor and which will not..
3. Locations that will *not* be multicolor must have a color code from 0 to 7 in color memory. These are the first eight color values:

**0 BLACK**  
**1 WHITE**  
**2 RED**  
**3 CYAN**  
**4 PURPLE**  
**5 GREEN**  
**6 BLUE**  
**7 YELLOW**

4. Locations that *will* be multicolor must have a color code from 8 to 15 in color memory (locations 55296 to 56319). Even though the code numbers are 8 to 15, these still display as the first eight color values:

**8 BLACK**  
**9 WHITE**  
**10 RED**  
**11 CYAN**  
**12 PURPLE**  
**13 GREEN**  
**14 BLUE**  
**15 YELLOW**

(This is because in multicolor character mode only the lowest three bits of color memory, bits 0-2, are read as color codes. Bit 3 is read as toggle between regular and multicolor mode. A 1 in bit 3 switches on multicolor mode for that character; a 0 switches it off.)

5. Four colors can be visible at the same time in each multicolor character.

The colors are assigned as follows:

Bit-pair	Control register	Location	Colors available	Characters controlled
00	Background color #0	53281	16	all
01	Auxiliary color #1	53282	16	all
10	Auxiliary color #2	53283	16	all
11	Color memory	55296-56319	8	1

**Positions, not patterns.** Remember that when you POKE a number into color memory to define a color and select multicolor or standard display, you are setting those parameters for the character *position*, not the character *pattern*. For example, if you are POKEing the character \* into screen location 0, and you want it to be multicolor with a black foreground, you will POKE the number 8 into the first color memory location, 55296. That color assignment stays with the character position, not with the character pattern. You can put an \* anywhere else on the screen, but it won't carry those color instructions with it. However, if you put any other character at screen location 0, it will still follow the color instructions you assigned when you put the asterisk there.

Also, using PRINT in multicolor mode will work as before — which means the screen editor will change color memory wherever it PRINTs a character, by POKEing the most recently selected foreground color into color memory. You can use this feature by PRINTing color instructions as CHR\$ values or quote mode characters, or you can defeat it by POKEing characters directly into screen memory.

## Custom Multicolor Characters

In general, the built-in character set isn't particularly useful with multicolor mode. You will usually use multicolor text mode with a custom character set specifically designed to work with four colors and double-dot resolution. If you want to use standard text along with special four-color characters, just start with a copy of the standard character set and only redefine the graphics characters or other characters you don't intend to use in standard mode.

## Multicolor Character Design

The strength and weakness of custom multicolor character sets is

# Color

that you are restricted to a limited number of different shapes. This is a drawback if you want to create a single elaborate and detailed picture — then you might be better off using bitmapped mode. However, if you want to be able to create many different displays using the same basic picture elements, the multicolor character set is ideal. If you design characters so they can fit together in many different ways to make different pictures, each character set you create will become a set of building blocks.

For instance, you might create characters that are different patterns of stonework which, in combination, can be made into castles. Or you might create different pieces of railroad tracks or highways to make transportation layouts. Or wall units and furniture to make houseplans, or map features and symbols to create detailed maps, or bricks and roof units to make houses — once your character set is created, you can combine characters to make any number of different displays.

Program 2-2 is a multicolor character editor. It works almost exactly like the character editor from Chapter 1, except that now you can select from among four colors and design building blocks that will fit together to create elaborate multicolored drawings.

## Using the Multicolor Character Editor

The instructions from the character editor in Chapter 1 still apply, except for these changes:

**The character display.** The large character grid is still eight dots by eight dots, except that now the dots are controlled in pairs.

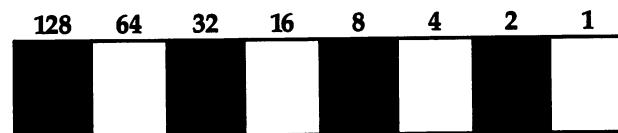
**The edited character.** As before, the character that is being edited will be displayed just to the right of the editing grid. Since this is a multicolor editor, the character will be displayed in multicolor mode, allowing you to see the changes on the character as you make them.

**Changing colors.** In the earlier editor, the space bar was used to toggle the individual bits *on* and *off*. In multicolor mode, we must select one of four colors for each double dot. To do this, press one of the number keys 1-4 while the editing cursor is over a space. This will change the color of that location to the selected color. Note: Actual colors are not displayed on the editing grid since the final character colors will be determined by you when you use the characters in your programs. Instead, the colors are represented by shading on the grid.

**Scrolling.** In multicolor mode, the bit patterns change *both* the shape and color of the characters. Scrolling the characters left

and right changes the bit patterns and can cause color changes. If you use the bit pattern in Figure 2-9, the color displayed will be auxiliary color 1. Scrolling this pattern left or right will result in the pattern shown in Figure 2-10, which displays auxiliary color 2.

**Figure 2-9. Multicolor Bit Pattern for Auxiliary Color 1**



**Figure 2-10. Multicolor Bit Pattern for Auxiliary Color 2**



This is one feature of the program that will require some experimentation. To scroll left or right *without* changing the color, simply scroll twice in the desired direction. This reestablishes the former color arrangement, since it moves the entire shape two bits in the desired direction. (Scrolling up and down produces no unexpected results.)

**Program 2-2. Multicolor Character Editor**

```
10 REM *** CHANGE SCREEN POINTERS ***
20 REM
21 PRINT CHR$(8);:POKE53283,12
25 POKE 53270,PEEK(53270)OR16:PRINT"CYAN";
30 POKE 56578,PEEK(56578) OR 3
40 POKE 56576,(PEEK(56576) AND 252)OR 0
50 POKE 53272,(PEEK(53272) AND 240)OR 2
60 POKE648,196: FF=0:POKE 53282,2
70 REM
80 REM *** TRANSFER CHARACTER SET ***
90 REM
100 POKE 56334,PEEK(56334)AND254
110 POKE 1, PEEK(1)AND 251
120 FOR R=53248 TO 55296
130 POKE R-2048, PEEK(R): NEXT
```

## Color

# Color

```
370 K$(N) = "[ 2 {RIGHT}]{ 5 {I}}{DOWN}{ 5 {LEFT}}
    {RVS} F"+STR$(N)+" {DOWN}{ 7 {LEFT}}{OFF}{M}
    {RVS}{ 5 {I}}{OFF} {G}"
380 K$(N+1) = "{M}{RVS}F F"+STR$(N+1)+" {*} {OFF}{G}"
390 P$(N) = "[ 2 {RIGHT}]{A}{ 3 *} {S}{DOWN}
    { 5 {LEFT}}{BF}"+STR$(N)+"B{DOWN}{ 7 {LEFT}}
    {OFF}{M} {Z}{ 3 *} {X}{OFF} {G}"
400 P$(N+1) = "{M}N F"+STR$(N+1)+" M{OFF}{G}{DOWN}
    { 8 {LEFT}}{ 7 {T}}{RIGHT}{UP}": NEXT
405 REM
410 REM *** DEFINE KEY POSITIONS ***
415 REM
420 FOR R=1TO8:K$(R) = "[ 7 {DOWN}]{ 31 {RIGHT}}"+K
    $(R):NEXT
430 FOR R=1TO7 STEP2: K$(R) = "[HOME]"+SP$+K$(R):SP
    $=SP$+"{ 4 {DOWN}}":NEXT
440 SP$ = "[ 3 {DOWN}]":FOR R=2TO8 STEP2: K$(R) =
    [HOME]"+SP$+K$(R):SP$=SP$+"{ 4 {DOWN}}":NEXT
    :SP$=""
450 FOR R=1TO8:P$(R) = "[ 7 {DOWN}]{ 31 {RIGHT}}"+P
    $(R):NEXT
460 FOR R=1TO7 STEP2: P$(R) = "[HOME]"+SP$+P$(R):SP
    $=SP$+"{ 4 {DOWN}}":NEXT
470 SP$ = "[ 3 {DOWN}]":FOR R=2TO8 STEP2: P$(R) =
    [HOME]"+SP$+P$(R):SP$=SP$+"{ 4 {DOWN}}":NEXT
480 REM
490 REM ***** DEFINE MESSAGES *****
500 REM
510 M$(1) = "[HOME]{ 8 {DOWN}}{ 22 {RIGHT}}EDIT...
    { 2 {DOWN}}{ 8 {LEFT}}{SAVE...}{ 2 {DOWN}}
    { 8 {LEFT}}"
520 M$(1) = M$(1) + "LOAD...{ 2 {DOWN}}{ 8 {LEFT}}CO
    PY...{ 2 {DOWN}}{ 8 {LEFT}}CLEAR...
    { 2 {DOWN}}{ 8 {LEFT}}"
530 M$(1) = M$(1) + "FILL...{ 2 {DOWN}}{ 8 {LEFT}}WO
    RK...{ 2 {DOWN}}{ 8 {LEFT}}FUNCTION"
540 M$(2) = "[HOME]{ 8 {DOWN}}{ 22 {RIGHT}}REVERSE.
    { 2 {DOWN}}{ 8 {LEFT}}INVERT..{ 2 {DOWN}}
    { 8 {LEFT}}"
550 M$(2) = M$(2) + "FLIP...{ 2 {DOWN}}{ 8 {LEFT}}SC
    ROLL R{ 2 {DOWN}}{ 8 {LEFT}}SCROLL L
    { 2 {DOWN}}{ 8 {LEFT}}"
560 M$(2) = M$(2) + "SCROLL U{ 2 {DOWN}}{ 8 {LEFT}}SC
    ROLL D{ 2 {DOWN}}{ 8 {LEFT}}FUNCTION"
570 REM
580 REM *** DEFINE RULER LINES ***
590 REM
600 L$ = "[HOME]{ 4 {DOWN}}{ 40 *}{DOWN}"
610 L$ = L$ + "[ 20 _]{R}{ 19 _}{RIGHT}"
```

# Color

```
620 L$=L$+"{ 19 {RIGHT}}-{DOWN}{LEFT}-{DOWN}
    {LEFT}-{DOWN}{LEFT}-{DOWN}{LEFT}-{DOWN}
    {LEFT}-{DOWN}{LEFT}-{DOWN}{LEFT}-{DOWN}
    {LEFT}-{DOWN}{LEFT}{{W}{DOWN}{LEFT}}-{DOWN}
    {LEFT}-{DOWN}{LEFT}-{DOWN}{LEFT}"-
630 L$=L$+"{DOWN}{LEFT}-{DOWN}{LEFT}-{DOWN}{LEFT}
    -{DOWN}{LEFT}-{ 8 {UP}}{ 21 {LEFT}}{ 20 *_}
    {HOME}"
900 REM
910 REM ***** DISPLAY EDIT SCREEN ****
920 REM
930 PRINT"{CLR}";CD$:PRINTCG$:PRINTL$:PRINTM$(1)
940 REM
941 REM *** EDIT CHARACTERS ***
942 REM
945 A$="@"
1000 REM
1010 REM *** DISPLAY FUNCTION KEYS ***
1020 REM
1025 POKE 55753, 14
1030 PRINT"{HOME}";:FOR R=1TO8: PRINTK$(R);: NEXT
    : PRINT"{HOME}";
1040 A(0)=1:A(1)=3:A(2)=5:A(3)=7:A(4)=2:A(5)=4:A(
    6)=6:A(7)=8
1045 GOTO 1087
1050 REM
1060 REM *** GET KEYBOARD ENTRY ***
1070 REM
1080 PRINT"{HOME}{ 5 {DOWN}}ENTER FUNCTION:
    { 23 {SPACES}}"
1081 POKE55753,11
1082 GETA$:IF A$=""THEN1080
1084 REM
1085 REM --- SKIP INVALID KEYS ---
1086 REM
1087 VA=ASC(A$)
1088 IF (VA<32 OR VA>223) THEN 1080
1089 IF (VA>95 AND VA<133) THEN 1080
1090 IF (VA>140 AND VA<161) THEN 1080
1091 IF (VA>140 OR VA<132) THEN1205
1100 REM
1110 REM --- GET FUNCTION KEYS ---
1120 REM
1130 PRINTP$(A(ASC(A$)-133))
1140 FOR R=0TO99:NEXT
1150 PRINTK$(A(ASC(A$)-133));"{HOME}";
1160 GOTO 1390
1180 REM
1190 REM - CONVERT ASC TO SCREEN CODE -
```

```
1200 REM
1205 LT=VA
1210 IF VA>31 AND VA<64 THEN SC=VA: GOTO 1240
1220 IF VA>63 AND VA<193 THEN SC=VA-64: GOTO 1240
1230 IF VA>191 AND VA<224 THEN SC=VA-128: GOTO 12
    40
1240 REM
1250 REM -- DISPLAY EDITED CHARACTER --
1260 REM
1262 LT$=STR$(LT):LT$=RIGHT$(LT$,LEN(LT$)-1)
1265 PRINT "{HOME}{ 5 {DOWN}}ENTRY MODE: CHR$(";LT
    $;"){ 17 {SPACES}}"
1270 POKE50633, SC
1273 PRINT "{HOME}{ 7 {DOWN}}{ 6 {RIGHT}}";
1275 PRINT "{HOME}{ 5 {DOWN}}ENTRY MODE: CHR$(";LT
    $;"){ 17 {SPACES}}"
1276 POKE50633, SC
1277 PRINT "{HOME}{ 7 {DOWN}}{RIGHT}{DOWN}
    { 6 {RIGHT}}";
1278 FOR R=0 TO 7:RR=PEEK(R+(51200+8*SC))
1279 PRINTZZ$(2*(ABS((RRAND128)=128))+(ABS((RRAND
    64)=64)));
1280 PRINTZZ$(2*(ABS((RRAND32)=32))+(ABS((RRAND16
    )=16)));
1281 PRINTZZ$(2*(ABS((RRAND8)=8))+(ABS((RRAND4)=4
    )));
1282 PRINTZZ$(2*(ABS((RRAND2)=2))+(ABS((RRAND1)=1
    )));
1283 PRINT "{DOWN}{ 8 {LEFT}}";:NEXT:GOTO1080
1350 GOTO 1080
1360 REM
1370 REM -- SPECIAL FUNCTION ROUTINES--
1380 REM
1390 IF FF=1 THEN 1500
1400 ON A(ASC(A$)-133) GOTO 2005,1433,1443,1453,1
    462,1472,1483,1493
1410 GOTO 1170
1430 REM
1431 REM --- SAVE A CHARACTER SET ---
1432 REM
1433 PRINT "{HOME}{ 5 {DOWN}}SAVE ON CASSETTE OR D
    ISK? (C/D):{ 6 {SPACES}}"
1434 GET Q$:IFQ$=""THEN1434
1435 IF Q$<>"C" AND Q$<>"D" THEN 1087
1436 GOTO 1760
1440 REM
1441 REM --- LOAD A CHARACTER SET ---
1442 REM
1443 PRINT "{HOME}{ 5 {DOWN}}LOAD FROM CASSETTE OR
    DISK? (C/D):{ 4 {SPACES}}"
```

# Color

```
1444 GET Q$:IF Q$=""THEN1444
1445 IF Q$<>"C" AND Q$<>"D" THEN 1087
1446 GOTO 1910
1450 REM
1451 REM --- COPY A CHARACTER ---
1452 REM
1453 PRINT "{HOME}{ 5 {DOWN}}ENTER CHARACTER TO CO
    PY:{ 14 {SPACES}}"
1454 GET CA$:IF CA$="" THEN 1454
1455 GOTO 1605
1456 IF CA$="" THEN 1454
1459 GOTO 1275
1460 REM
1461 REM ----- CLEAR A CHARACTER -----
1462 PRINT "{HOME}{ 5 {DOWN}}CLEAR CHAR: CHR$(";LT
    $;"){ 17 {SPACES}}"
1463 H=51200+8*SC
1464 FOR R=H TO H+7: POKE R,0: NEXT
1465 PRINTCG$:GOTO 1080
1470 REM
1471 REM ----- FILL A CHARACTER -----
1472 PRINT "{HOME}{ 5 {DOWN}}FILL{ 2 {SPACES}}CHAR
    : CHR$(";LT$;"){ 17 {SPACES}}"
1473 H=51200+8*SC
1474 FOR R=H TO H+7: POKE R,255: NEXT
1475 PRINTCF$:GOTO 1080
1480 REM
1481 REM ---- GOTO WORK SPACE ---
1482 REM
1483 PRINT "{HOME}{ 5 {DOWN}}ENABLE WORK SPACE
    { 22 {SPACES}}"
1484 GOTO 2705
1490 REM
1491 REM --- SWITCH FUNCTION SET ---
1492 REM
1493 FF=1: PRINTM$(2):GOTO 1080
1499 RETURN
1500 ON A(ASC(A$)-133) GOTO 1523,1532,1542,1551,1
    561,1572,1582,1593
1510 GOTO 1170
1520 REM
1521 REM --- REVERSE CHARACTER BITS---
1522 REM
1523 PRINT "{HOME}{ 5 {DOWN}}REVERSE CHARACTER:
    { 20 {SPACES}}"
1524 H=51200+8*SC
1525 FOR R=H TO H+7: POKE R,255-PEEK(R): NEXT
1529 GOTO 1275
1530 REM
```

# Color

```
1531 REM --- INVERT CHARACTER BITS -----
1532 PRINT" {HOME}{ 5 {DOWN}}INVERTING CHARACTER:
{ 18 {SPACES}}"
1533 H=51200+8*SC
1534 FOR R=H TO H+7: T(R-H)=PEEK(R): NEXT
1535 FOR R=H TO H+7: POKE R,T(7-(R-H)): NEXT:GOTO
1275
1540 REM
1541 REM ----- FLIP CHARACTER BITS -----
1542 PRINT" {HOME}{ 5 {DOWN}}FLIPPING CHARACTER:
{ 19 {SPACES}}"
1543 FOR U=51200+8*SC TO (51200+8*SC)+7
1544 Z=PEEK(U)
1545 R=128*(ABS((ZAND1)=1))+64*(ABS((ZAND2)=2))+3
2*(ABS((ZAND4)=4))
1546 R=R+16*(ABS((ZAND8)=8))+8*(ABS((ZAND16)=16))
+4*(ABS((ZAND32)=32))
1547 R=R+2*(ABS((ZAND64)=64))+1*(ABS((ZAND128)=12
8))
1548 POKE U,R
1549 NEXT: GOTO1275
1550 REM ----- SCROLL RIGHT -----
1551 PRINT" {HOME}{ 5 {DOWN}}SCROLLING RIGHT:
{ 22 {SPACES}}"
1552 FOR U=51200+8*SC TO (51200+8*SC)+7
1553 R=(PEEK(U)/2)
1554 IF PEEK(U)/2<>INT(PEEK(U)/2) THEN R=R+128
1555 POKE U,R
1556 NEXT: GOTO1275
1560 REM ----- SCROLL LEFT -----
1561 PRINT" {HOME}{ 5 {DOWN}}SCROLLING LEFT:
{ 23 {SPACES}}"
1562 FOR U=51200+8*SC TO (51200+8*SC)+7
1563 R=(PEEK(U)*2)
1564 IF PEEK(U)>=128 THEN R=R+1
1565 IF R>255 THEN R=R-256
1566 POKE U,R
1567 NEXT: GOTO1275
1570 REM
1571 REM ----- SCROLL UP -----
1572 PRINT" {HOME}{ 5 {DOWN}}SCROLLING UP:
{ 25 {SPACES}}"
1573 FOR R=0TO7
1574 Y(R)=PEEK(R+51200+8*SC):NEXT
1575 FOR R=0TO6
1576 POKE R+51200+8*SC, Y(R+1): NEXT
1577 POKE 51207+8*SC, Y(0)
1579 GOTO 1275
1580 REM
```

# Color

```
1581 REM ----- SCROLL DOWN -----
1582 PRINT" {HOME}{ 5 {DOWN}}SCROLLING DOWN:
{ 23 {SPACES}}"
1583 FOR R=0TO7
1584 Y(R)=PEEK(R+51200+8*SC):NEXT
1585 FOR R=1TO7
1586 POKE R+51200+8*SC, Y(R-1): NEXT
1587 POKE 51200+8*SC, Y(7)
1589 GOTO 1275
1590 REM
1591 REM --- SWITCH FUNCTION SET ---
1592 REM
1593 FF=0: PRINTM$(1):GOTO 1080
1599 RETURN
1600 REM
1601 REM *** COPY CHARACTER ROUTINE ***
1602 REM
1605 DA=ASC(CA$)
1610 IF (DA<32 OR DA>223) THEN 1456
1620 IF (DA>95 AND DA<133) THEN 1456
1630 IF (DA<141 AND DA>132) THEN A$=CA$:GOTO 1080
1640 IF (DA>140 AND DA<161) THEN RETURN
1650 PRINT" {HOME}{ 5 {DOWN}}{ 25 {RIGHT}}";CA$
1660 IF DA>31 AND DA<64 THEN SD=DA: GOTO 1690
1670 IF DA>63 AND DA<193 THEN SD=DA-64: GOTO 1690
1680 IF DA>191 AND DA<224 THEN SD=DA-128: GOTO 16
90
1690 VJ=51200+8*SD:JJ=51200+8*SC
1700 FOR R=0 TO 7
1710 POKE JJ+R,PEEK(VJ+R): NEXT
1720 GOTO 1456
1730 REM
1740 REM *** SAVE CHAR SET ROUTINE ***
1750 REM
1760 PRINT" {HOME}{ 5 {DOWN}}SAVE: FILE NAME -----
--.CHR{ 8 {SPACES}}{HOME}{ 5 {DOWN}}
{ 16 {RIGHT}}";
1770 LL=0:NM$=""
1775 FOR R=0TO30: PRINT" {RVS}-{OFF}{LEFT}";
1780 GET A$:IFA$=""THEN NEXT
1790 IFA$<>""THEN 1840
1800 FOR R=0TO30: PRINT"-{LEFT}";
1810 GET A$:IFA$=""THEN NEXT
1820 IFA$<>""THEN 1840
1830 GOTO 1775
1840 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$=CHR$(34)ORA$="{UP}"THEN 1775
1845 IFA$="{DOWN}"ORA$="{RIGHT}"ORA$="{LEFT}"THEN
1775
```

```
1847 IF ASC(A$)>132 AND ASC(A$)<141 THEN 1080
1850 PRINTA$;
1855 NM$=NM$+A$:LL=LL+1:IFLL=6THEN1870
1860 GOTO 1775
1870 NM$=NM$+.CHR"
1875 IF Q$="C"THEN1890
1880 OPEN 1,8,4,NM$+",W"
1885 FOR R=51200 TO 52224
1887 PRINT#1,PEEK(R):NEXT
1889 CLOSE1:GOTO 1275
1890 OPEN 1,1,1,NM$:GOTO 1885
1900 REM
1901 REM *** SAVE CHAR SET ROUTINE ***
1902 REM
1910 PRINT"{HOME}{ 5 {DOWN}}LOAD: FILE NAME -----
- .CHR{ 8 {SPACES}}{HOME}{ 5 {DOWN}}
{ 16 {RIGHT}}";
1915 LL=0:NM$=""
1920 FOR R=0TO30: PRINT" {RVS}-{OFF}{LEFT}";
1925 GET A$:IFA$=""THEN NEXT
1930 IFA$<>"THEN 1955
1935 FOR R=0TO30: PRINT"-{LEFT}";
1940 GET A$:IFA$=""THEN NEXT
1945 IFA$<>"THEN 1955
1950 GOTO 1920
1955 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$
=CHR$(34)ORA$="{UP}"THEN1920
1960 IFA$="{DOWN}"ORA$="{RIGHT}"ORA$="{LEFT}"THEN
1920
1965 IF ASC(A$)>132 AND ASC(A$)<141 THEN 1080
1970 PRINTA$;
1975 NM$=NM$+A$:LL=LL+1:IFLL=6THEN1980
1976 GOTO 1920
1980 NM$=NM$+.CHR"
1985 IF Q$="C"THEN1999
1987 OPEN 1,8,4,NM$+",R"
1988 FOR R=51200 TO 52224
1989 INPUT#1,VL:POKE R,VL:NEXT
1990 CLOSE1:GOTO 1275
1999 OPEN 1,1,1,NM$:GOTO 1885
2000 REM
2001 REM ** CHARACTER EDIT ROUTINE **
2002 REM
2005 PRINT" {HOME}{ 5 {DOWN}}EDIT{ 2 {SPACES}}MODE
: CHR$(",LT$;"){ 17 {SPACES}}"
2010 PRINT" {HOME}{ 8 {DOWN}}{ 8 {RIGHT}}";
2015 LO=50503: SV=PEEK(LO) :SM=(51200+(8*SC)): EX
= 7 :GT=SC
2020 FOR R=0TO30:POKE LO,SV:POKELO+1,SV
```

# Color

```
2030 GET A$:IFA$=""THEN NEXT
2040 IFA$<>""THEN 2090
2050 FORR=0TO30:POKELO,102:POKELO+1,102
2060 GET A$:IFA$=""THEN NEXT
2070 IFA$<>""THEN 2090
2080 GOTO 2020
2090 IFA$="{UP}" THEN 2200
2091 IFA$="{DOWN}" THEN 2100
2092 IFA$="{LEFT}" THEN 2300
2093 IFA$="{RIGHT}" THEN 2400
2094 IF A$="1"THEN 2500
2095 IF A$="2"THEN 2570
2096 IF A$="3"THEN 2600
2097 IF A$="4"THEN 2670
2098 IF(ASC(A$)>132ANDASC(A$))<141THEN POKE LO,SV
:VA=ASC(A$):GOTO 1087
2099 GOTO 2020
2100 POKE LO,102:POKELO+1,102:FORR=0TO40:NEXT
2110 IF GT=SC+7 THEN 2020
2115 IF SV=79 THEN 2130
2120 POKE LO,SV:POKELO+1,SV:LO=LO+40: GT=GT+1: SV
=PEEK (LO) :SM=SM+1: GOTO 2020
2130 POKE LO,SV:POKELO+1,119:LO=LO+40: GT=GT+1: S
V=PEEK (LO) :SM=SM+1: GOTO 2020
2200 POKE LO,102:POKELO+1,102:FORR=0TO40:NEXT
2210 IF GT=SC THEN2020
2215 IF SV=79 THEN 2230
2220 POKE LO,SV:POKELO+1,SV:LO=LO-40: GT=GT-1: SV
=PEEK(LO): SM=SM-1: GOTO 2020
2230 POKE LO,SV:POKELO+1,119:LO=LO-40: GT=GT-1: S
V=PEEK(LO): SM=SM-1: GOTO 2020
2300 POKE LO,102:POKELO+1,102:FORR=0TO40:NEXT
2310 IF EX=7 THEN 2020
2315 IF SV=79 THEN 2330
2320 POKE LO,SV:POKELO+1,SV:LO=LO-2: EX=EX+2: SV=
PEEK(LO): GOTO 2020
2330 POKE LO,SV:POKELO+1,119:LO=LO-2: EX=EX+2: SV
=PEEK(LO): GOTO 2020
2400 POKE LO,102:POKELO+1,102:FORR=0TO40:NEXT
2410 IF EX=1 THEN 2020
2415 IF SV=79 THEN 2430
2420 POKE LO,SV:POKELO+1,SV:LO=LO+2: EX=EX-2: SV=
PEEK(LO): GOTO 2020
2430 POKE LO,SV:POKELO+1,119:LO=LO+2: EX=EX-2: SV
=PEEK(LO): GOTO 2020
2500 POKESM,(PEEK(SM)AND(255-(2↑EX+2↑(EX-1))))ORE
1(EX):POKELO,32:POKELO+1,32
2505 SV=32:GOTO 2020
```

```
2570 POKESM,(PEEK(SM)AND(2↑EX+2↑(EX-1))))ORE
2571 2(EX):POKELO,128:POKELO+1,128
2575 SV=128:GOTO 2020
2600 POKESM,(PEEK(SM)AND(2↑EX+2↑(EX-1))))ORE
2601 3(EX):POKELO,129:POKELO+1,129
2605 SV=129:GOTO 2020
2670 POKESM,(PEEK(SM)AND(2↑EX+2↑(EX-1))))ORE
2671 4(EX):POKELO,160:POKELO+1,160
2675 SV=160:GOTO 2020
2700 REM
2701 REM *** WORK SPACE ROUTINE ***
2702 REM
2705 HZ=0:LZ=0
2710 PRINT" {HOME}{LT/GREY}{ 17 {DOWN}}";
2720 FOR R=0TO30: PRINT" {RVS} {OFF}{LEFT}";
2730 GET A$:IFA$=""THEN NEXT
2740 IFA$<>""THEN 2790
2750 FOR R=0TO30: PRINT" {LEFT}";
2760 GET A$:IFA$=""THEN NEXT
2770 IFA$<>""THEN 2790
2780 GOTO 2720
2790 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$
=CHR$(34)THEN2720
2810 IF ASC(A$)>132 AND ASC(A$)<141 THENPRINT"
{CYAN}";:GOTO 1087
2820 IFA$="{UP}" THEN 3200
2830 IFA$="{DOWN}" THEN 3100
2840 IFA$="{LEFT}" THEN 3300
2850 IFA$="{RIGHT}" THEN 3400
2860 IF HZ<18 AND LZ<6 THEN PRINTA$;:HZ=HZ+1:GOTO
2720
2870 IF HZ=18 AND LZ<6 THEN 2720
2880 IF HZ=18 AND LZ=6 THEN 2720
3100 IF LZ=6 THEN2720
3110 PRINT" {LEFT}{DOWN}";:LZ=LZ+1:GOTO 2720
3200 IF LZ=0 THEN2720
3210 PRINT" {LEFT}{UP}";:LZ=LZ-1:GOTO 2720
3300 IF HZ=0 THEN2720
3310 PRINT" { 2 {LEFT}}";:HZ=HZ-1:GOTO 2720
3400 IF HZ=18 THEN2720
3410 PRINT" {LEFT}{RIGHT}";:HZ=HZ+1:GOTO 2720
```

## Multicolor Sprites

The rules for multicolor sprites are basically the same as those for multicolor characters. In multicolor mode, sprites have half the normal horizontal resolution, (making them 12 dots wide by 21

# Color

dots high), but instead of displaying only one or two colors, they can display as many as four colors at once.

```
10 POKE 53276, 1
20 POKE 53269, 1
30 POKE 53248, 160
40 POKE 53277, 1
50 POKE 53271, 1
60 POKE 53249, 100
70 POKE 2040, 200
80 POKE 53285, 2
90 POKE 53286, 7
100 POKE 53287, 5
110 FOR R=12800 TO 12863 STEP 3
120 POKE R, 168
130 POKE R+1, 5
140 POKE R+2, 127
150 NEXT
```

The primary differences between multicolor sprites and multicolor characters are where the extra colors are assigned and how they are turned on. Table 2-5 outlines the procedure. Notice that the colors selected by bit-pairs 01 and 11 are the same for all sprites, while the color for bit-pair 10 is individually selected for each sprite.

Multicolor sprites do have one major advantage over multicolor characters. Once a sprite has been programmed, its shapes, color and multicolor characteristics stay with it no matter where it is positioned on the screen.

## A Multicolor Sprite Editor

Program 2-3 is very much like the one-color sprite editor from Chapter 1. The new features from Program 2-2 have been added; there are several other changes as well:

**Special color functions.** The special work area (available using the Work function) allows you to move and expand the sprites as before. In addition to this, you can change two of the sprite colors from within the program. These colors are the sprite foreground color and the screen (background) color. The two auxiliary colors are preset in the program on line 205. If you want to change them, LIST the line, change the color values to the number you want (from 0 to 15), press RETURN to enter the line, and then RUN the program again.

**Table 2-5. Setting Up Multicolor Sprites**

1. Turn on multicolor mode:

Sprite	Statement
#0	POKE 53276, PEEK(53276)OR 1
#1	POKE 53276, PEEK(53276)OR 2
#2	POKE 53276, PEEK(53276)OR 4
#3	POKE 53276, PEEK(53276)OR 8
#4	POKE 53276, PEEK(53276)OR 16
#5	POKE 53276, PEEK(53276)OR 32
#6	POKE 53276, PEEK(53276)OR 64
#7	POKE 53276, PEEK(53276)OR 128

2. Assign colors. Three colors can be visible at one time in any one sprite. Bit-pair 00 does not cause a color to be displayed. This bit-pair is transparent — it allows whatever is on the screen behind the sprite to show through. All 16 colors are available in all registers. Colors are assigned as follows:

Bit-pair	Sprite number	Control register (transparent)	Location	Sprites controlled
00			none	all
01		Auxiliary color #0	53285	all
10	#0		53287	1
	#1		53288	1
	#2		53289	1
	#3		53290	1
	#4		53291	1
	#5		53292	1
	#6		53293	1
	#7		53294	1
11		Auxiliary color #1	53286	all

**Program 2-3. Multicolor Sprite Editor**

```

10 REM *** CHANGE SCREEN POINTERS ***
20 REM
21 PRINT CHR$(8);:LZ$(0)="O":LZ$(1)="{RVS} {OFF}"
25 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,27,237,
     213,0,255,200,0,19,127,240,0,0,0
26 DATA 255,0,254,246,241,242,243,244,999
27 R3=PEEK(53272):POKE53276,255
30 POKE 56578,PEEK(56578) OR 3
40 POKE 56576,(PEEK(56576) AND 252)OR 0
50 POKE 53272,(PEEK(53272) AND 240)OR 2
60 POKE648,196: FF=0

```

# Color

```
70 REM
71 REM *** SET - UP SPRITE REGISTERS ***
72 FOR R=53287 TO 53294: POKE R,14:NEXT
73 FOR R= 0 TO 7: POKE 51192+R, R: NEXT: POKE 532
    69, 255: POKE 53264,0
74 FOR R=49152 TO 49663 STEP 6:POKE R,170:POKER+1
    ,170:POKER+2,170:NEXT
75 FOR R=49155 TO 49663 STEP 6:POKE R,85:POKER+1,
    85:POKER+2,85:NEXT
76 FOR R=0 TO 14 STEP 2:POKE R+53248,25+R*15:POKE
    R+53249,52:NEXT
80 REM *** TRANSFER CHARACTER SET ***
90 REM
100 POKE 56334,PEEK(56334)AND254
110 POKE 1, PEEK(1)AND 251
120 FOR R=53248 TO 55296
130 POKE R-2048, PEEK(R): NEXT
140 POKE 1, PEEK(1)OR 4
150 POKE 56334,PEEK(56334)OR 1
160 REM
170 REM *** DEFINE CHARACTER GRID ***
171 DIM G(47),ZX(47),T(64),Y(64),CS(16)
180 REM
190 CG$="O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}
    O{Y}O{Y}{G}{DOWN}{ 25 {LEFT}}"
192 CF$="{RVS}{ 24 {SPACES}}{OFF}{DOWN}
    { 24 {LEFT}}}"
195 BK$="{ 24 {SPACES}}{DOWN}{ 24 {LEFT}}"
196 BE$="{ 24 {SPACES}}"
197 X=51928:Y=239:D=221:POKEX,Y:POKEX+1,D:POKEX+2
    ,Y:POKEX+3,D:POKEX+4,Y
198 POKEX+5,D:POKEX+6,Y:POKEX+7,D
199 X=51936:Y=85:D=170:POKEX,Y:POKEX+1,D:POKEX+2,
    Y:POKEX+3,D:POKEX+4,Y
200 POKEX+5,D:POKEX+6,Y:POKEX+7,D
205 POKE 53285,10: POKE 53286,5
210 ZZ$(0)="{ 2 {SPACES}}":ZZ$(1)="{ 2 {-}}":ZZ$(
    2)="{ 2 +}":ZZ$(3)="{RVS}{ 2 {SPACES}}{OFF}"
220 E4(7)=192:E4(5)=48:E4(3)=12:E4(1)=3
222 E3(7)=128:E3(5)=32:E3(3)=8:E3(1)=2
224 E2(7)=64:E2(5)=16:E2(3)=4:E2(1)=1
226 E1(7)=0:E1(5)=0:E1(3)=0:E1(1)=0
230 REM *** ENABLE SPRITE DISPLAY ***
240 REM
250 POKE 53269,255
340 REM *** DEFINE FUNCTION KEYS ***
350 REM
351 FOR R=1TO30:S$=S$+"{RIGHT)":NEXT
360 FOR N=1TO8 STEP2
```

```

370 K$(N) = " { 2 {RIGHT}}{ 5 {I}}{DOWN}{ 5 {LEFT}}
    {RVS} F"+STR$(N)+" {DOWN}{ 7 {LEFT}}{OFF}{M}
    {RVS}{ 5 {I}}{OFF} {G}"
380 K$(N+1) = "{M}{RVS} {F"+STR$(N+1)+" {*} {OFF}}{G}
    {HOME}"
390 P$(N) = " { 2 {RIGHT}}{A}{ 3 *} {S}{DOWN}
    { 5 {LEFT}}BF"+STR$(N)+" B{DOWN}{ 7 {LEFT}}
    {OFF}{M} {Z}{ 3 *} {X}{OFF} {G}"
400 P$(N+1) = "{M}N F"+STR$(N+1)+" M{OFF}{G}{HOME}"
    : NEXT
405 REM
410 REM *** DEFINE KEY POSITIONS ***
415 REM
420 FOR R=1TO8:K$(R) = " { 8 {DOWN}}{ 31 {RIGHT}}"+K
    $(R):NEXT
430 FOR R=1TO7 STEP2: K$(R) = "{HOME}"+SP$+K$(R):SP
    $=SP$+" { 4 {DOWN}}":NEXT
440 SP$ = " { 3 {DOWN}}":FOR R=2TO8 STEP2: K$(R) =
    {HOME}"+SP$+K$(R):SP$=SP$+" { 4 {DOWN}}":NEXT
    :SP$=""
450 FOR R=1TO8:P$(R) = " { 8 {DOWN}}{ 31 {RIGHT}}"+P
    $(R):NEXT
460 FOR R=1TO7 STEP2: P$(R) = "{HOME}"+SP$+P$(R):SP
    $=SP$+" { 4 {DOWN}}":NEXT
470 SP$ = " { 3 {DOWN}}":FOR R=2TO8 STEP2: P$(R) =
    {HOME}"+SP$+P$(R):SP$=SP$+" { 4 {DOWN}}":NEXT
480 REM
490 REM ***** DEFINE MESSAGES *****
500 REM
510 M$(1) = "{HOME}{ 8 {DOWN}}{RIGHT}{DOWN}
    { 24 {RIGHT}}EDIT.{ 2 {DOWN}}{ 5 {LEFT}}SAVE
    .{ 2 {DOWN}}{ 5 {LEFT}}"
520 M$(1) = M$(1) + "LOAD.{ 2 {DOWN}}{ 5 {LEFT}}COPY
    { 2 {DOWN}}{ 5 {LEFT}}CLEAR{ 2 {DOWN}}
    { 5 {LEFT}}"
530 M$(1) = M$(1) + "FILL.{ 2 {DOWN}}{ 5 {LEFT}}WORK.
    { 2 {DOWN}}{ 5 {LEFT}}FNCTN"
540 M$(2) = "{HOME}{ 9 {DOWN}}{ 25 {RIGHT}}REVRS
    { 2 {DOWN}}{ 5 {LEFT}}INVRT{ 2 {DOWN}}
    { 5 {LEFT}}"
550 M$(2) = M$(2) + "FLIP.{ 2 {DOWN}}{ 5 {LEFT}}SCL R
    { 2 {DOWN}}{ 5 {LEFT}}SCL L{ 2 {DOWN}}
    { 5 {LEFT}}"
560 M$(2) = M$(2) + "SCL U{ 2 {DOWN}}{ 5 {LEFT}}SCL D
    { 2 {DOWN}}{ 5 {LEFT}}FNCTN"
570 REM
580 REM ***** DEFINE RULER LINES *****
590 REM

```

# Color

```
600 L$="{}HOME}{ 3 {DOWN}}{ 30 *}{E}{LEFT}{UP}-
    {LEFT}{UP}-{LEFT}{UP}-{ 3 {DOWN}}{ 9 *}"-
610 L$=L$+"{ 16 {LEFT}}{ 4 {DOWN}}{ 16 *}"-
900 REM
910 REM **** DISPLAY EDIT SCREEN ****
920 REM
930 PRINT"[CLR]";CD$:PRINTL$:PRINTM$(1)
932 PRINT"[HOME]{ 4 {DOWN}}";:FOR R=0TO19:PRINTCG
    $;:NEXT:PRINT"O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O
    {Y}O{Y}O{Y}O{Y}O{Y}O{G}[HOME]";
1000 REM
1010 REM *** DISPLAY FUNCTION KEYS ***
1020 REM
1030 PRINT"[HOME]";:FOR R=1TO8: PRINTK$(R);: NEXT
    : PRINT"[HOME]";
1040 A(0)=1:A(1)=3:A(2)=5:A(3)=7:A(4)=2:A(5)=4:A(
    6)=6:A(7)=8
1050 REM
1060 REM *** GET KEYBOARD ENTRY ***
1070 REM
1080 PRINT"[HOME]{ 4 {DOWN}}{ 25 {RIGHT}}ENTER FU
    NCTION:{ 2 {DOWN}}{ 15 {LEFT}}";
1081 PRINT"{ 15 {SPACES}}"
1082 GETA$:IFA$=""THEN1082
1084 REM
1085 REM --- SKIP INVALID KEYS ---
1086 REM
1087 VA=ASC(A$)
1088 IF (VA>132 AND VA<141) THEN 1130
1089 IF (VA<49 OR VA>56) THEN 1080
1090 GOTO 1205
1100 REM
1110 REM --- GET FUNCTION KEYS ---
1120 REM
1130 PRINTPS(A(ASC(A$)-133))
1140 FORR=0TO99:NEXT
1150 PRINTK$(A(ASC(A$)-133));"[HOME]";
1160 GOTO 1390
1180 REM
1190 REM --- PRINT EDITED SPRITE ---
1200 REM
1205 POKE SP*2+53248,25+SP*30
1210 POKE 53264,2↑(VA-49):POKE53248+(VA-49)*2,40:
    SP=VA-49
1220 GOTO 1265
1265 PRINT"[HOME]{ 4 {DOWN}}{ 25 {RIGHT}}ENTRY MO
    DE:{ 4 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1267 PRINT"SPRITE #";SP+1;"{LEFT}{ 6 {SPACES}}"
```

# Color

```
1268 PRINT" {HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINTC
    G$;:NEXT:PRINT"O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O{Y}O
    {Y}O{Y}O{Y}O{Y}O{Y}{G}{HOME}";
1270 PRINT" {HOME}{ 4 {DOWN}}";
1272 LZ$(0)="O":LZ$(1)="{RVS} {OFF}"
1275 FOR R=0 TO 62
1277 IF (R<>0) AND (R/3=INT(R/3)) THEN GOSUB 1310
1278 RR=PEEK(R+(49152+64*SP))
1279 PRINTZZ$(2*(ABS((RRAND128)=128))+ABS((RRAND
    64)=64)));
1280 PRINTZZ$(2*(ABS((RRAND32)=32))+ABS((RRAND16
    )=16)));
1281 PRINTZZ$(2*(ABS((RRAND8)=8))+ABS((RRAND4)=4
    )));
1282 PRINTZZ$(2*(ABS((RRAND2)=2))+ABS((RRAND1)=1
    )));
1283 NEXT: GOTO 1080
1310 PRINT" {DOWN}{ 24 {LEFT}}";:RETURN
1360 REM
1370 REM -- SPECIAL FUNCTION ROUTINES--
1380 REM
1390 IF FF=1 THEN 1500
1400 ON A(ASC(A$)-133) GOTO 2005,1432,1442,1452,1
    461,1471,2700,1493
1410 GOTO 1170
1430 REM
1431 REM ----- SAVE SPRITES -----
1432 POKE SP*2+53248,25+SP*30:POKE 53264,0
1433 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SAVE ON
    DISK OR{ 2 {DOWN}}{ 15 {LEFT}}";
1434 PRINT"CASSETTE? (C/D)"
1435 GET A$:IF A$=""THEN1435
1436 IF A$<>"C" AND A$<>"D" THEN 1087
1437 Q$=A$:GOTO 1760
1440 REM
1441 REM ----- LOAD SPRITES -----
1442 POKE SP*2+53248,25+SP*30:POKE 53264,0
1443 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}LOAD ON
    DISK OR{ 2 {DOWN}}{ 15 {LEFT}}";
1444 PRINT"CASSETTE? (C/D)"
1445 GET A$:IF A$=""THEN1445
1446 IF A$<>"C" AND A$<>"D" THEN 1087
1447 Q$=A$:GOTO 1910
1450 REM
1451 REM ----- COPY A SPRITE -----
1452 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SPRITE T
    O COPY?{ 2 {DOWN}}{ 15 {LEFT}}";
1453 PRINT"ENTER (1-8){ 4 {SPACES}}"
1454 GET CA$:IF CA$="" THEN 1454
```

# Color

```
1455 GOTO 1605
1456 IF CA$="" THEN 1454
1459 GOTO 1275
1460 REM ----- CLEAR A SPRITE -----
1461 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}CLEARING
{ 7 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1462 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1463 H=49152+64*SP
1464 FOR R=H TO H+63: POKE R,0: NEXT
1465 PRINT" {HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINTC
G$;:NEXT:PRINT" { 24 _}{G}{HOME}";
1467 GOTO 1080
1470 REM ----- FILL A SPRITE -----
1471 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}FILLING
{ 8 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1472 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1473 H=49152+64*SP
1474 FOR R=H TO H+63: POKE R,255: NEXT
1475 PRINT" {HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINTC
F$;:NEXT:PRINT" {RVS}{ 24 {SPACES}}{OFF}{G}
{HOME}"
1477 GOTO 1080
1490 REM
1491 REM --- SWITCH FUNCTION SET ---
1492 REM
1493 FF=1: PRINTM$(2); "{HOME}";:GOTO 1080
1499 RETURN
1500 ON A(ASC(A$)-133) GOTO 1522,1531,1541,1551,1
561,1572,1582,1593
1510 GOTO 1170
1520 REM
1521 REM --- REVERSE SPRITE BITS---
1522 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}REVERSIN
G:{ 5 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1523 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1524 H=49152+64*SP
1525 FOR R=H TO H+63: POKE R,255-PEEK(R): NEXT
1529 GOTO 1270
1530 REM --- INVERT SPRITE BITS---
1531 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}INVERTIN
G:{ 5 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1532 PRINT"SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1533 H=49152+64*SP
1534 FOR R=H TO H+62: T(R-H)=PEEK(R): NEXT
1535 FOR R=H TO H+60 STEP 3
1536 POKE R,T(60-(R-H))
1537 POKE R+1,T(61-(R-H))
1538 POKE R+2,T(62-(R-H)): NEXT:GOTO 1270
1540 REM --- FLIP SPRITE BITS---
```

```

1541 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}FLIPPING
 :{ 6 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1542 PRINT" SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1543 H=49152+64*SP
1544 FOR R=H TO H+62: T(R-H)=PEEK(R): NEXT
1545 FOR R=H TO H+60STEP 3:POKE R+2,T(R-H):POKE R+
 1,T(R+1-H):POKE R,T(R+2-H):NEXT
1546 FOR U=H TO H+63:Z=PEEK(U):R=128*(ABS((ZAND1)
 =1))+64*(ABS((ZAND2)=2))
1547 R=R+32*(ABS((ZAND4)=4))+16*(ABS((ZAND8)=8))+8*(ABS((ZAND16)=16))
1548 R=R+4*(ABS((ZAND32)=32))+2*(ABS((ZAND64)=64))
 )+1*(ABS((ZAND128)=128)):POKEU,R
1549 NEXT:GOTO1270
1550 REM ----- SCROLL RIGHT -----
1551 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SCROLLIN
 G RIGHT{ 2 {DOWN}}{ 15 {LEFT}}";
1552 PRINT" SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1553 FOR U=49152+64*SP TO (49152+64*SP)+62 STEP 3
1554 R=(PEEK(U)/2):R1=(PEEK(U+1)/2):R2=(PEEK(U+2)
 /2)
1555 IF PEEK(U)/2<>INT(PEEK(U)/2) THEN R1=R1+128
1556 IF PEEK(U+1)/2<>INT(PEEK(U+1)/2) THEN R2=R2+
 128
1557 IF PEEK(U+2)/2<>INT(PEEK(U+2)/2) THEN R=R+12
 8
1558 POKE U,R:POKEU+1,R1:POKEU+2,R2
1559 NEXT: GOTO1270
1560 REM ----- SCROLL LEFT -----
1561 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SCROLLIN
 G LEFT { 2 {DOWN}}{ 15 {LEFT}}";
1562 PRINT" SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1563 FORU=49152+64*SPTO(49152+64*SP)+62STEP3:R=
 PEEK(U)*2:R1=PEEK(U+1)*2:R2=PEEK(U+2)*2
1564 IF PEEK(U)=>128 THEN R2=R2+1
1565 IF R2>255 THEN R2=R2-256
1566 IF PEEK(U+1)=>128 THEN R=R+1
1567 IF R>255 THEN R=R-256
1568 IF PEEK(U+2)=>128 THEN R1=R1+1
1569 IF R1>255 THEN R1=R1-256
1570 POKE U,R:POKEU+1,R1:POKEU+2,R2:NEXT: GOTO 12
 70
1571 REM ----- SCROLL UP -----
1572 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SCROLLIN
 G UP{ 3 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
1573 PRINT" SPRITE #{ 7 {SPACES}}{ 7 {LEFT}}";SP+1
1574 U=49152+64*SP:Y=PEEK(U):Y1=PEEK(U+1):Y2=PEEK
 (U+2)
1575 FOR R=0 TO 59 STEP 3

```

# Color

```
1576 POKE U+R,PEEK(U+R+3):POKE U+R+1,PEEK(U+R+4):  
    POKE U+R+2,PEEK(U+R+5):NEXT  
1577 POKEU+60,Y:POKEU+61,Y1:POKEU+62,Y2  
1578 GOTO 1270  
1580 REM  
1581 REM ----- SCROLL DOWN -----  
1582 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SCROLLIN  
    G DOWN { 2 {DOWN}}{ 15 {LEFT}}";  
1583 PRINT"SPRITE #{ 5 {SPACES}}X { 7 {LEFT}}";SP  
    +1  
1584 U=49152+64*SP:Y=PEEK(U+60):Y1=PEEK(U+61):Y2=  
    PEEK(U+62)  
1585 POKEU+62,PEEK(U+62-3)  
1586 FOR M=59 TO 2 STEP -3  
1587 POKE U+M,PEEK(U+M-3):POKEU+M+1,PEEK(U+M-2):  
    POKE U+M+2,PEEK(U+M-1):NEXT  
1588 POKEU,Y:POKEU+1,Y1:POKEU+2,Y2  
1589 GOTO 1270  
1590 REM  
1591 REM --- SWITCH FUNCTION SET ---  
1592 REM  
1593 FF=0: PRINTM$(1);"{HOME}";:GOTO 1080  
1599 RETURN  
1600 REM  
1601 REM *** COPY SPRITE ROUTINE **  
1602 REM  
1605 DA=ASC(CA$)  
1610 IF (DA>132 AND DA<141) THEN VA=DA:GOTO 1130  
1620 IF (DA<49 OR DA>56) THEN 1080  
1690 VJ=49152+64*SP:JJ=49152+64*(DA-49)  
1700 FOR R=0 TO 63  
1710 POKE VJ+R,PEEK(JJ+R): NEXT  
1720 GOTO 1080  
1730 REM  
1740 REM *** SAVE SPRITES ROUTINE ***  
1750 REM  
1760 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SAVE:(NA  
    ME){ 4 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";  
1765 PRINT"-----.SPR{ 5 {SPACES}}{ 15 {LEFT}}";  
1770 LL=0:NM$=""  
1775 FOR R=0 TO 30: PRINT" {RVS}-{OFF}{LEFT}";  
1780 GET A$:IFA$=""THEN NEXT  
1790 IFA$<>""THEN 1840  
1800 FOR R=0TO30: PRINT"-{LEFT}";  
1810 GET A$:IFA$=""THEN NEXT  
1820 IFA$<>""THEN 1840  
1830 GOTO 1775  
1840 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$  
    =CHR$(34)ORA$="{UP}"THEN1775
```

```
1845 IFA$=" {DOWN}" ORA$=" {RIGHT}" ORA$=" {LEFT}" THEN  
1775  
1847 IF ASC(A$)>132 AND ASC(A$)<141 THEN 1080  
1850 PRINTA$;  
1855 NM$=NM$+A$:LL=LL+1:IFLL=6THEN1870  
1860 GOTO 1775  
1870 NM$=NM$+.SPR"  
1872 FOR R=53248 TO 53294:ZX(R-53248)=PEEK(R):  
    NEXT  
1874 POKE SP*2+53248,25+SP*30: POKE 53264, 0  
1875 IF Q$="C"THEN1890  
1880 RESTORE:OPEN 1,8,4,"@:"+NM$+",W": R=0  
1881 READ F: IF F<>999 THEN POKE R+53248,F:R=R+1:  
    GOTO 1881  
1882 FOR R=49152 TO 49663  
1883 PRINT" {HOME}{ 4 {DOWN}}{ 36 {RIGHT}}";R-4915  
    2  
1884 PRINT#1,PEEK(R):NEXT  
1885 FOR R=53248 TO 53294: POKE R,ZX(R-53248):  
    NEXT  
1888 CLOSE1: PRINT" {HOME}{ 6 {DOWN}}{ 25 {RIGHT}}  
    { 14 {SPACES}}";:GOTO 1080  
1890 OPEN 1,1,1,NM$:GOTO 1881  
1900 REM  
1901 REM *** LOAD SPRITES ROUTINE ***  
1902 REM  
1910 PRINT" {HOME}{ 4 {DOWN}}{ 25 {RIGHT}}LOAD:(NA  
    ME){ 4 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";  
1913 RESTORE  
1915 PRINT"-----.SPR{ 5 {SPACES}}{ 15 {LEFT}}";  
1920 LL=0:NM$=""  
1925 FOR R=0 TO 30: PRINT" {RVS}-{OFF}{LEFT}";  
1930 GET A$:IFA$=""THEN NEXT  
1935 IFA$<>" "THEN 1960  
1940 FOR R=0TO30: PRINT"-{LEFT}";  
1945 GET A$:IFA$=""THEN NEXT  
1950 IFA$<>" "THEN 1960  
1955 GOTO 1925  
1960 IFA$=CHR$(20)ORA$=CHR$(148)ORA$=CHR$(13)ORA$  
    =CHR$(34)ORA$="{UP}"THEN1925  
1965 IFA$=" {DOWN}" ORA$=" {RIGHT}" ORA$=" {LEFT}" THEN  
    1925  
1970 IF ASC(A$)>132 AND ASC(A$)<141 THEN 1080  
1975 PRINTA$;  
1980 NM$=NM$+A$:LL=LL+1:IFLL=6THEN1990  
1985 GOTO 1925  
1990 NM$=NM$+.SPR"  
1991 FOR R=53248 TO 53294:ZX(R-53248)=PEEK(R):  
    NEXT
```

# Color

```
1992 POKE SP*2+53248,25+SP*30: POKE 53264, 0
1993 IF Q$="C"THEN 2001
1994 OPEN 1,8,4,NM$+",R": R=0
1995 READ F: IF F<>999 THEN POKE R+53248,F:R=R+1:
GOTO 1995
1996 FOR R=49152 TO 49663
1997 PRINT "{HOME}{ 4 {DOWN}}{ 36 {RIGHT}}";R-4915
2
1998 INPUT#1,A:POKE R,A:NEXT
1999 FOR R=53248 TO 53294: POKE R,ZX(R-53248):
NEXT
2000 CLOSE1: PRINT "{HOME}{ 6 {DOWN}}{ 25 {RIGHT}}"
{ 14 {SPACES}}";:GOTO 1080
2001 OPEN 1,1,0,NM$:GOTO 1995
2002 REM
2003 REM ** SPRITE EDIT ROUTINE **
2004 REM
2005 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}EDIT MOD
E:{ 5 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
2010 IF PEEK(53264)=0 THEN PRINT "{RVS}WHICH SPRIT
E?{OFF}{ 2 {SPACES}}{ 7 {LEFT}}";:GOTO 1082
2015 LO=50336:SV=PEEK(LO):SM=(49152+(SP*64)):EX=7
:GT=SC:BY=0
2020 FOR R=0TO30:POKELO,SV:POKELO+1,SV
2030 GET A$:IFA$=""THEN NEXT
2040 IFA$<>" "THEN 2090
2050 FORR=0TO30:POKELO,102:POKELO+1,102
2060 GET A$:IFA$=""THEN NEXT
2070 IFA$<>" "THEN 2090
2080 GOTO 2020
2090 IFA$="{UP}" THEN 2200
2091 IFA$="{DOWN}" THEN 2100
2092 IFA$="{LEFT}" THEN 2300
2093 IFA$="{RIGHT}" THEN 2400
2094 IFA$="1" THEN 2500
2095 IFA$="2" THEN 2570
2096 IFA$="3" THEN 2600
2097 IFA$="4" THEN 2670
2098 IF (ASC(A$)>132 AND ASC(A$)<141) THEN POKE L
O,SV:VA=ASC(A$):GOTO 1087
2099 GOTO 2020
2100 POKE LO,102:POKE LO+1,102:FORR=0TO40:NEXT
2110 IF GT>=SC+60 THEN 2020
2115 IF SV=79THEN2130
2120 POKE LO,SV:POKE LO+1,SV:LO=LO+40: GT=GT+3:SV
=PEEK (LO) :SM=SM+3: GOTO 2020
2130 POKE LO,SV:POKE LO+1,119:LO=LO+40: GT=GT+3:S
V=PEEK (LO) :SM=SM+3: GOTO 2020
2200 POKE LO,102:POKE LO+1,102:FORR=0TO40:NEXT
```

```

2210 IF GT<=SC+2THEN2020
2215 IF SV=79THEN2230
2220 POKE LO,SV:POKE LO+1,SV:LO=LO-40: GT=GT-3: S
V=PEEK(LO): SM=SM-3: GOTO 2020
2230 POKE LO,SV:POKE LO+1,119:LO=LO-40: GT=GT-3:
SV=PEEK(LO): SM=SM-3: GOTO 2020
2300 POKE LO,102:POKE LO+1,102:FORR=0TO40:NEXT
2310 IF EX=7 AND BY=0 THEN 2020
2312 IF EX<>7 THEN 2317
2314 IF SV=79THEN2316
2315 POKELO,SV:POKELO+1,SV:LO=LO-2:EX=1:SM=SM-1:G
T=GT-1:SV=PEEK(LO):BY=BY-1:GOTO2020
2316 POKELO,SV:POKELO+1,119:LO=LO-2:EX=1:SM=SM-1:
GT=GT-1:SV=PEEK(LO):BY=BY-1:GOTO2020
2317 IF SV=79THEN2330
2320 POKE LO,SV:POKE LO+1,SV:LO=LO-2: EX=EX+2: SV
=PEEK(LO): GOTO 2020
2330 POKE LO,SV:POKE LO+1,119:LO=LO-2: EX=EX+2: S
V=PEEK(LO): GOTO 2020
2400 POKE LO,102:POKELO+1,102:FORR=0TO40:NEXT
2410 IF EX=1 AND BY=2 THEN 2020
2414 IF EX<>1 THEN 2420
2415 IF SV=79THEN2419
2416 POKELO,SV:POKELO+1,SV:LO=LO+2:EX=7:SM=SM+1:G
T=GT+1:SV=PEEK(LO):BY=BY+1:GOTO2020
2419 POKELO,SV:POKELO+1,119:LO=LO+2:EX=7:SM=SM+1:
GT=GT+1:SV=PEEK(LO):BY=BY+1:GOTO2020
2420 IF SV=79THEN2430
2425 POKE LO,SV:POKE LO+1,SV: LO=LO+2: EX=EX-2: S
V=PEEK(LO): GOTO 2020
2430 POKE LO,SV:POKE LO+1,119: LO=LO+2: EX=EX-2:
SV=PEEK(LO): GOTO 2020
2500 POKESM,(PEEK(SM)AND(255-(2↑EX+2↑(EX-1))))ORE
1(EX):POKELO,79:POKELO+1,119
2505 SV=32:GOTO2020
2570 POKESM,(PEEK(SM)AND(255-(2↑EX+2↑(EX-1))))ORE
2(EX):POKELO,92:POKELO+1,92
2575 SV=92:GOTO2020
2600 POKESM,(PEEK(SM)AND(255-(2↑EX+2↑(EX-1))))ORE
3(EX):POKELO,91:POKELO+1,91
2605 SV=91:GOTO2020
2670 POKESM,(PEEK(SM)AND(255-(2↑EX+2↑(EX-1))))ORE
4(EX):POKELO,160:POKELO+1,160
2675 SV=160:GOTO2020
2700 REM
2701 REM *** WORK SPACE ROUTINE ***
2702 REM
2705 PRINT"{{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}}ENABLE S
PRITE{ 2 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";

```

# Color

```
2707 PRINT"WORK AREA{ 6 {SPACES}}"  
2709 FOR R=0 TO 14 STEP 2:POKE R+53248,25+R*15:  
    POKE R+53249,52:NEXT:POKE53264,0  
2710 W$(1)={"HOME}{ 8 {DOWN}}{RIGHT}{DOWN}  
    { 24 {RIGHT}}SEL..{ 2 {DOWN}}{ 5 {LEFT}}COLO  
    R{ 2 {DOWN}}{ 5 {LEFT}}"  
2711 C$(0)={"BLACK}":C$(1)={"WHITE}":C$(2)={"RED}  
    ":C$(3)={"CYAN}":C$(4)={"PURPLE}":C$(5)="  
    {GREEN}":C$(6)={"BLUE}"  
2712 C$(7)={"YELLOW}":C$(8)={"ORANGE}":C$(9)="  
    {BROWN}":C$(10)={"LT/RED}":C$(11)={"LT/GREY}  
    ":C$(12)={"MED/GREY}":C$(13)={"LT/GREEN}"  
2713 C$(14)={"LT/BLUE}":C$(15)={"DK/GREY}"  
2720 W$(1)=W$(1)+"BCKGD{ 2 {DOWN}}{ 5 {LEFT}}AD/D  
    L{ 2 {DOWN}}{ 5 {LEFT}}MOVE { 2 {DOWN}}  
    { 5 {LEFT}}"  
2730 W$(1)=W$(1)+"2X HZ{ 2 {DOWN}}{ 5 {LEFT}}2X V  
    T{ 2 {DOWN}}{ 5 {LEFT}}FNCTN"  
2740 PRINT"{HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINTB  
    K$;:NEXT:PRINTBE$;W$(1)  
2745 REM  
2747 REM -- GET WORK SPACE COMMANDS --  
2749 REM  
2750 PRINT"{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}ENTER FU  
    NCTION { 2 {DOWN}}{ 15 {LEFT}}";  
2751 PRINT"{ 15 {SPACES}}"  
2759 GETW$:IFW$=""THEN2759  
2760 WA=ASC(W$)  
2770 IF (WA>132 AND WA<141) THEN 3000  
2780 IF (WA<48 OR WA>56) THEN 2750  
2785 WS=WA-49  
2787 IF AD=1 THEN 2800  
2788 IF MV=1 THEN 3450  
2789 GOTO 2750  
2791 REM  
2792 REM --- GET SPRITES ROUTINE ---  
2793 REM  
2800 IF PEEK(53249+WS*2)=>85 THEN 2900  
2810 POKE 53248+WS*2,24:POKE 53249+WS*2,85:GOTO 2  
    759  
2900 POKE 53248+(WS*2),25+(WS*2)*15:POKE53249+(WS  
    *2),52:GOTO2759  
2910 GETW$:IFW$=""THEN2910  
2920 WA=ASC(W$)  
2930 IF (WA>132 AND WA<141) THEN 3000  
2940 IF (WA<48 OR WA>57) THEN 2750  
2950 WS=WA-49  
2960 IF AD=1 THEN 2800  
3000 ON WA-132 GOTO 3150,3325,3420,3620,3222,3510  
    ,3720,3800
```

```
3100 REM
3101 REM ---- SELECT ROUTINE ----
3102 REM
3110 IF I=15 THEN I=-1
3120 I=I+1:RETURN
3150 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SELECT D
ISABLED{ 2 {DOWN}}{ 15 {LEFT}}";
3160 PRINT "USE A/D OR MOVE"
3170 FOR R=0 TO 1500: NEXT: GOTO 2750
3200 REM
3210 REM --- SPRITE COLOR ROUTINE ---
3220 REM
3222 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}CHANGE S
PRITE{ 2 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3223 PRINT "COLOR (USE SEL)"
3230 GET W$: CL$=W$: IF W$="" THEN 3230
3231 WA=ASC(W$)
3232 IF (WA>133 AND WA<141) THEN 3000
3233 IF (WA>47 AND WA<57) THEN WS=WA-49:GOTO 3270
3240 IF W$<>"{F-1}" THEN 2760
3250 GOSUB 3110
3260 POKE 53287+WS,I: GOTO 3230
3270 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}SELECTED
{ 7 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3275 PRINT "SPRITE # { 7 {SPACES}}{ 7 {LEFT}}"; WS+1
3280 GOTO 3230
3300 REM
3310 REM -- BACKGROUND COLOR ROUTINE --
3320 REM
3325 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}USE SEL
TO{ 5 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3326 PRINT "CHANGE BACKGRND"
3330 GET CL$: IF CL$="" THEN 3330
3340 IF CL$<>"{F-1}" THEN W$=CL$:GOTO 2760
3350 GOSUB 3100
3360 PRINT "{HOME}";CS(I);
3370 PRINT "{HOME}{ 4 {DOWN}}";:FOR R=0TO19:PRINT"
{RVS}";BK$,:NEXT:PRINTBE$;"{HOME}{LT/BLUE}
{OFF}";: GOTO 3330
3400 REM
3410 REM --- MOVE SPRITES ROUTINE ---
3420 PRINT "{HOME}{ 4 {DOWN}}{ 25 {RIGHT}}MOVING
{ 9 {SPACES}}{ 2 {DOWN}}{ 15 {LEFT}}";
3421 PRINT "SPRITE # { 7 {SPACES}}{ 7 {LEFT}}"; WS+1
3425 GETW$: IF W$="" THEN 3425
3427 WA=ASC(W$)
3429 IF W$="{DOWN}" OR W$="{UP}" OR W$="{RIGHT}" OR W$=
"{LEFT}" THEN 3435
3430 IF (WA>133 AND WA<141) THEN 3000
3432 IF (WA>47 AND WA<57) THEN WS=WA-49:GOTO 3480
```

## Color

```
3830 PRINTM$(1);:FOR R=53287 TO53294: POKE R,14:  
NEXT:POKE53277,0:POKE53271,0  
3840 GOTO 1080
```

## Extended Color Mode

Often in programming, especially when you are writing applications programs that include menus and text displays, you will want to create different colored *windows* — sections of the screen or particular messages printed on different background colors. In standard and multicolor text modes this is not possible — the background color is assigned by one location, 53281. Extended color mode allows up to four different background colors on the screen at the same time.

### How It Works

**Partial character set.** Extended color mode is a text mode — it uses the character set. However, it does not use the whole thing. Instead, you can only use the first 64 characters in the character set, the characters numbered from 0 to 63 in the screen codes. They consist of the numbers from 0 to 9, the capital letters from A to Z, and the punctuation marks and symbols.

Why only 64 characters? Because it is possible to express all these characters' screen codes in only six bits. The highest code number, 63, looks like this in binary: 0011111. Those two leftmost bits, bits 6 and 7, are not used.

**The background bit-pair.** Extended color mode uses those two high bits to assign the background color. The foreground color is still mapped in color memory, in the regular way, but the background color is selected by those two bits. The VIC-II chip looks at the lower six bits to get the screen code for the character, a number from 0 to 63; then it looks at the upper two bits to get the background color instruction for that character position.

**The background registers.** The four background registers from 53281 to 53284 contain the actual color assignments. The low nybble color value is POKEd into them, so that each register can have any color from 0 to 15.

A 00 bit-pair selects register 0 (53281), 01 selects register 1 (53282), 10 selects register 2 (53283), and 11 selects register 3 (53284). See Appendix E for a complete listing of colors.

# Color

**Enable extended color mode.** To tell the VIC-II chip that you want extended BASIC mode, you must set bit 6 of location 53265:

**POKE 53265,PEEK(53265)OR 64**

## Assigning Background Colors

Since the background colors are controlled by bits 6 and 7, you can change those locations without changing their contents. Here is a routine that divides the screen into several overlapping boxes. When you type it in, don't clear the screen. When you RUN it, you'll see how you can create windows without disturbing the text on the screen.

When you RUN the program, you can then redraw any window by pressing a number from 1 to 4. Colors 1 through 3 are redrawn when you press the corresponding number key. Pressing 4 redraws the window in color 0.

```
20 POKE 53265,PEEK(53265)OR 64
30 GOSUB 100:GOSUB 200:GOSUB 300:GOSUB 400
47 REM
48 REM INPUT LOOP
49 REM
50 GET A$:IF A$="" THEN 50
60 A=VAL(A$):IF A<1 OR A>4 THEN 50
70 ON A GOSUB 100,200,300,400:GOTO 50
97 REM
98 REM DRAW WINDOW IN COLOR 1
99 REM
100 FOR I=1065 TO 1604 STEP 40:FOR J=I TO I+29:POK
    E J,PEEK(J)AND 63 OR 64
110 NEXT:NEXT:RETURN
197 REM
198 REM DRAW WINDOW IN COLOR 2
199 REM
200 FOR I=1356 TO 1995 STEP 40:FOR J=I TO I+26:POK
    E J,PEEK(J)AND 63 OR 128
210 NEXT:NEXT:RETURN
297 REM
298 REM DRAW WINDOW IN COLOR 3
299 REM
300 FOR I=1706 TO 1905 STEP 40:FOR J=I TO I+33:POK
    E J,PEEK(J)AND 63 OR 192
310 NEXT:NEXT:RETURN
397 REM
398 REM DRAW WINDOW IN COLOR 0
399 REM
400 FOR I=1274 TO 1393 STEP 40:FOR J=I TO I+19:POK
    E J,PEEK(J)AND 63:NEXT:NEXT
410 RETURN
```

# Color

**From the keyboard.** You can use extended color mode from the keyboard. Background color 0 will be selected by the screen editor when you type regular characters. When you type SHIFTed graphics characters, you get the letter whose key you pressed, but with background color 1. To get the numbers and symbols with color 1, press Commodore and a graphics key.

To access background colors 2 and 3, press CTRL-9. This turns on reverse mode, which will continue until you enter RETURN. Now background color 2 can be entered just like regular text, while background color 3 is entered as graphics characters.

**POKEing screen codes.** In your programs, one method of using extended color mode is to POKE characters directly into screen memory. It is relatively easy to handle the relationship between ASCII values and screen codes, since you only have to worry about the screen codes from 0 to 63. It happens that the ASCII and screen codes from 32 to 63 are identical. And the ASCII codes from 64 to 95 are identical to the screen codes from 0 to 31. All you have to do to convert from one to the other in extended color mode is to subtract 64 from all ASCII codes greater than 63. Then OR the resulting value with the background color value you want (color 1 = 64, color 2 = 128, and color 3 = 192) and POKE the result into screen memory.

Here is a demonstration program that uses this method with the windows we created in the last example. When you RUN it, a brief setup will be followed by the prompt PRESS A KEY. When you press a key, all letters will disappear from the screen. Then you can enter any text you want in the blue window in the middle of the screen. When you press RETURN or run out of room, the program will echo your input statement in the other three windows, then prompt you to press a key to start over. The program itself is useless, of course, but you can see how this technique might be applied to programs that use menus and messages a lot.

```
10 PRINT "{CLR}":DIM SC(3),SM(3)
15 SM(0)=1308:SM(1)=1105:SM(2)=1436:SM(3)=1746
20 POKE 53265,PEEK(53265)OR 64
30 FOR I=1065 TO 1604 STEP 40:FOR J=I TO I+29:POKE
   J,96
35 NEXT:NEXT
40 FOR I=1356 TO 1995 STEP 40:FOR J=I TO I+26:POKE
   J,160
45 NEXT:NEXT
50 FOR I=1706 TO 1905 STEP 40:FOR J=I TO I+33:POKE
   J,224
```

# Color

```
55 NEXT:NEXT
60 FOR I=1268 TO 1387 STEP 40:FOR J=I TO I+25:POKE
   J,32:NEXT:NEXT
70 FOR I=1 TO 25:FOR J=0 TO 3:POKE 55296+(SM(J)-10
   24)+I,32+64*J:NEXT:NEXT
80 GOTO 240
99 REM INPUT CONTROL LOOP
100 GET A$:IF A$="" THEN 100
110 L=ASC(A$):IF L=13 AND LEN(B$)>0 THEN 200
115 IF L=20 AND LEN(B$)>0 THEN GOSUB 600:GOTO 100
120 IF (L<32) OR (L>95) THEN 100
125 T=T+1:IF T=25 THEN 200
130 B$=B$+A$:GOSUB 310:POKE SM(0)+T,L
140 GOTO 100
199 REM OUTPUT CONTROL LOOP
200 FOR I=1 TO LEN(B$):GOSUB 300:GOSUB 400:NEXT
240 GOSUB 700:B$="":T=0:A$="":GOSUB 500:GOTO 100
299 REM CONVERT TO SCREEN CODE
300 L=ASC(MID$(B$,I,1))
310 IF L>63 THEN L=L-64
320 RETURN
399 REM PRINT IN WINDOWS
400 FOR J=1 TO 3:SC(J)=L OR (64*J):NEXT
410 FOR J=1 TO 3:POKE SM(J)+I,SC(J):NEXT
420 RETURN
499 REM CLEAR WINDOWS
500 FOR I=0 TO 3:FOR J=1 TO 25:POKE SM(I)+J,32+I*6
   4:NEXT:NEXT:RETURN
599 REM DELETE CHARACTER
600 B$=LEFT$(B$,LEN(B$)-1):POKE SM(0)+T,32:T=T-1:R
   ETURN
699 REM READY TO GO ON?
700 BS="PRESS A KEY{14 SPACES}":FOR I=631 TO 640:P
   OKE I,0:NEXT
710 FOR I=1 TO 25:GOSUB 300:POKE SM(0)+I,L:NEXT
720 GET A$:IF A$="" THEN 720
730 RETURN
```

**Fast windows with PRINT.** PRINTing is almost always faster than POKEing when it comes to changing the screen display. By combining cursor commands with strings of characters in arrays, you can create windows almost instantaneously. The only tricky part is converting input strings from one color to another.

Color 0 is the background for ASCII codes 32 to 95. Color 1 is the background for ASCII codes from 96 to 127 and from 160 to 191. The ASCII codes from 96 to 127 will display the same characters as the ASCII codes from 64 to 95; the ASCII codes from 160 to 191 will display the same characters as the codes from 32 to 63.

# Color

In other words, if the code in color 0 is greater than 63, add 32 to it to get it into color 1; otherwise, add 128.

Color 2 uses the same codes as color 0, and color 3 uses the same codes as color 1. However, to PRINT in colors 2 and 3, the string must begin with the RVS ON character — CHR\$(18), or CTRL-9 in quote mode. You can end these colors with a RVS OFF character, or simply allow a RETURN to be executed before PRINTing a string with a different background color.

The next program does much the same thing as the program above, except that it PRINTs the windows instead of POKEing them into screen memory. Since PRINT is an extremely fast machine language routine, this technique is very much like adding machine language to your program. This way it is possible to redraw the windows often without long delays.

To use PRINT effectively, however, you need to set up strings in advance. Since this program draws and redraws four windows, the strings are set up in four-element arrays (0-3), one for each background color. SW\$(n) is a *string* consisting of a HOME character followed by the correct number of CURSOR DOWN characters to position the cursor at the right line. TW(n) is the *number* of spaces to TAB in to get to the starting column. LW\$(n) is a *string* consisting of space characters, either CHR\$(32) or CHR\$(160); for colors 1 and 3, the string begins with CHR\$(18), which turns on reverse mode. LN(n) is the *number* of lines in the window. Once these variables are set up, all of the windows can be drawn with the same formula: PRINT SW\$(n);:FOR J = 1 TO LN(n):PRINT TAB(TW(n)) LW\$(n):NEXT

Here is the program listing:

```
5 POKE 53265, PEEK(53265)OR 64:PRINT "{CLR}{BLACK}"
10 DIM SW$(3), LW$(3), TW(3), LN(3), SL$(3), LL$(3), CH$(3)
15 SW$(0)="{HOME}{ 6 {DOWN}}":SW$(1)="{HOME}{DOWN}"
  :"SW$(2)=SW$(0)+"{ 2 {DOWN}}":SW$(3)=SW$(2)+"
  { 10 {DOWN}}"
20 FOR I=1 TO 27:LW$(0)=LW$(0)+CHR$(32):NEXT
25 FOR I=1 TO 31:LW$(1)=LW$(1)+CHR$(160):NEXT
30 LW$(2)="{RVS}":FOR I=1 TO 27:LW$(2)=LW$(2)+CHR$(32):NEXT
35 LW$(3)="{RVS}":FOR I=1 TO 32:LW$(3)=LW$(3)+CHR$(160):NEXT
40 TW(0)=3:TW(1)=1:TW(2)=11:TW(3)=2
45 LN(0)=3:LN(1)=13:LN(2)=16:LN(3)=5
50 FOR I=0 TO 3:SL$(I)=SW$(I)+"{DOWN}":NEXT:SL$(2)=SL$(2)+"{DOWN}"
```

# Color

```
60 FOR I=1 TO 3:PRINT SW$(I)::FOR J=1 TO LN(I):PRINT TAB(TW(I))LW$(I):NEXT:NEXT
70 PRINT SW$(0)::FOR I=1 TO LN(0):PRINT TAB(TW(0))
LW$(0):NEXT
80 GOTO 240
99 REM INPUT CONTROL LOOP
100 GET A$:IF A$="" THEN 100
110 L=ASC(A$):IF L=13 AND LEN(B$)>0 THEN 200
115 IF L=20 AND LEN(B$)>0 THEN GOSUB 600:GOTO 100
120 IF (L<32) OR (L>95) THEN 100
125 T=T+1:IF T=26 THEN 200
130 BS=B$+A$:PRINT SL$(0)TAB(TW(0)+1)BS
140 GOTO 100
199 REM OUTPUT CONTROL LOOP
200 FOR I=1 TO 3:ON I GOSUB 300,350,400:NEXT
210 FOR I=1 TO 3:GOSUB 800
230 PRINT SL$(I)TAB(TW(I)+1)CH$(I):NEXT
240 GOSUB 700:B$="":T=0:A$="":GOSUB 500:GOTO 100
299 REM SET UP COLOR 1
300 FOR J=1 TO LEN(B$):L=ASC(MID$(B$,J,1))
310 IF L<64 THEN L=L+128:GOTO 330
320 L=L+32
330 CH$(I)=CH$(I)+CHR$(L):NEXT:RETURN
349 REM SET UP COLOR 2
350 CH$(I)="{RVS}":CH$(I)=CH$(I)+B$:CH$(I)=CH$(I)+
" {OFF}":RETURN
399 REM SET UP COLOR 3
400 CH$(I)="{RVS}":GOSUB 300:CH$(I)=CH$(I)+" {OFF}"
:RETURN
499 REM CLEAR WINDOWS
500 FOR I=0 TO 3:PRINT SL$(I)TAB(TW(I))LW$(I):NEXT
:RETURN
599 REM DELETE CHARACTER
600 PRINT SL$(0)TAB(TW(0)+LEN(B$))" {LEFT}":B$=LEF
T$(B$,LEN(B$)-1):T=T-1:RETURN
699 REM READY TO GO ON?
700 B$=" PRESS A KEY":FOR I=631 TO 640:POKE I,0:NE
XT
710 PRINT SW$(0)::FOR J=1 TO LN(0):PRINT TAB(TW(0))
LW$(0):NEXT
720 PRINT SL$(0)TAB(TW(0))BS
730 GET A$:IF A$="" THEN 730
740 PRINT SL$(0)TAB(TW(0))LW$(0):GOSUB 500
750 FOR I=1 TO 3:CH$(I)"":NEXT:RETURN
799 REM OPTIONAL WINDOW REDRAW
800 PRINT SW$(I)::FOR J=1 TO LN(I):PRINT TAB(TW(I))
LW$(I):NEXT:RETURN
```

In programming with PRINT, you must be careful not to PRINT in the last line of the screen or the last character of any line, or the screen editor will start doing odd things to your display.

The relationship between screen codes, character codes, and background colors is shown in Table 2-6.

**Table 2-6. Extended Color Mode Values**

Color number	Bit-pair	Location	ASCII	Screen codes	ASCII	Screen codes
0	00	53281	32-63	32-63	64-95	0-31
1	01	53282	160-191	96-127	96-127	64-95
			RVS-ON		RVS-ON	
2	10	53283	32-63	160-191	64-95	128-159
			RVS-ON		RVS-ON	
3	11	53284	160-191	224-255	96-127	192-223

## Multicolor Bitmapped Mode

Multicolor bitmapped mode is set up by POKEing both the bitmapped mode instruction and the multicolor mode instruction:

**POKE 53265,PEEK(53265)OR 32:POKE 53270,PEEK(53270)OR 16**

The bitmap is set up in eight-byte cells, exactly as it was in standard bitmapped mode. Each byte of the bitmap is treated as four bit-pairs, each of which controls a double dot on the screen, just as in multicolor text mode.

## Assigning Colors

Now, however, color assignments are a bit more complex. The background color (00 bit-pairs) for the entire multicolor bitmapped screen is now taken from location 53281 — the background cannot be individually assigned for each cell.

The foreground colors (11 bit-pairs) are taken from color memory (55296 to 56319) — all 16 colors are available in each location, and multicolor mode is always in force in every cell.

The other two colors are individually assigned in screen memory, one byte for each cell in the bitmap. The high nibble controls 01 bit-pairs, and the low nibble controls 10 bit-pairs. (See Appendix E for a complete listing of high and low nibble color instructions.)

# Color

## Adjusting to Bitmapped Bit-Pairs

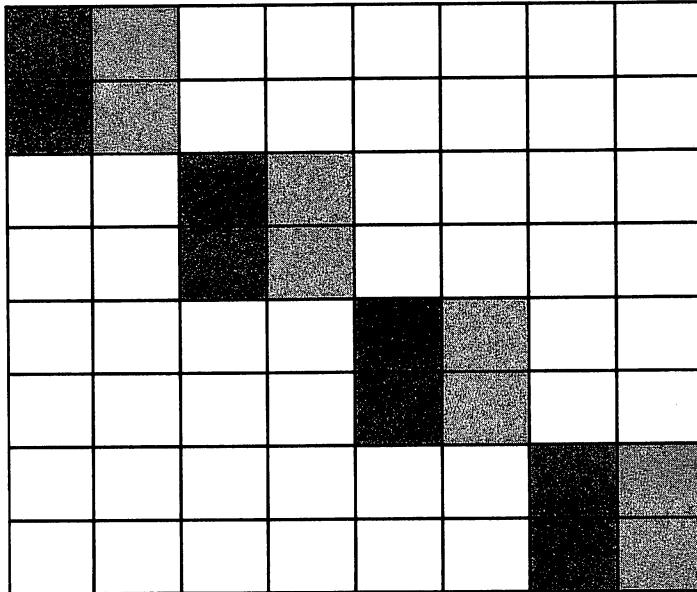
You'll quickly find that routines that worked in standard bit-mapped mode get startling results in multicolor bitmapped mode. Diagonals and circles will start looking ragged, because horizontal resolution is now much less precise. Furthermore, a routine that drew a diagonal line one bit wide will now draw a much less precise diagonal — and it will keep changing colors, as shown in Figure 2-11.

The only way to become completely comfortable with any of the modes is to write programs using it. Probably the most difficult modes to use effectively are the bitmapped modes, since they require so much manipulation of binary numbers. Text modes are much simpler to program and use. Armed with a character editor, you will find it quite easy to create effective displays — you can even set up character sets and text modes and then practice using them directly from the keyboard.

Chapter 4 includes some routines to help you put character sets and bitmapped screens in your own programs.

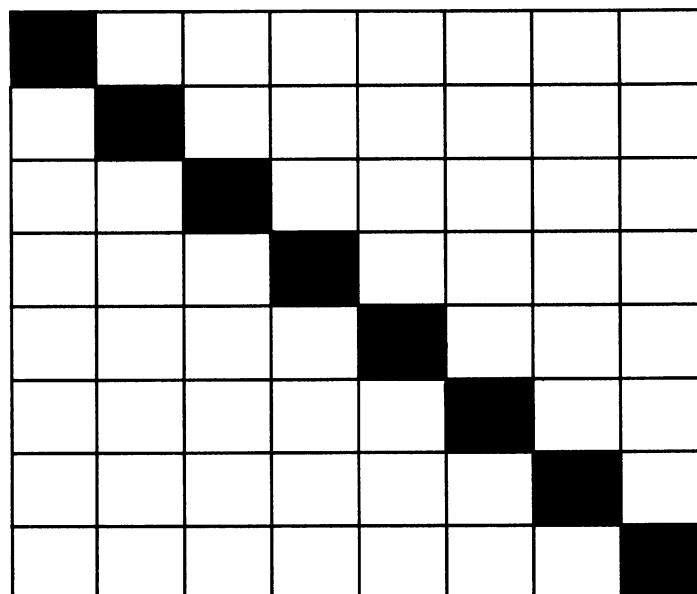
# Color

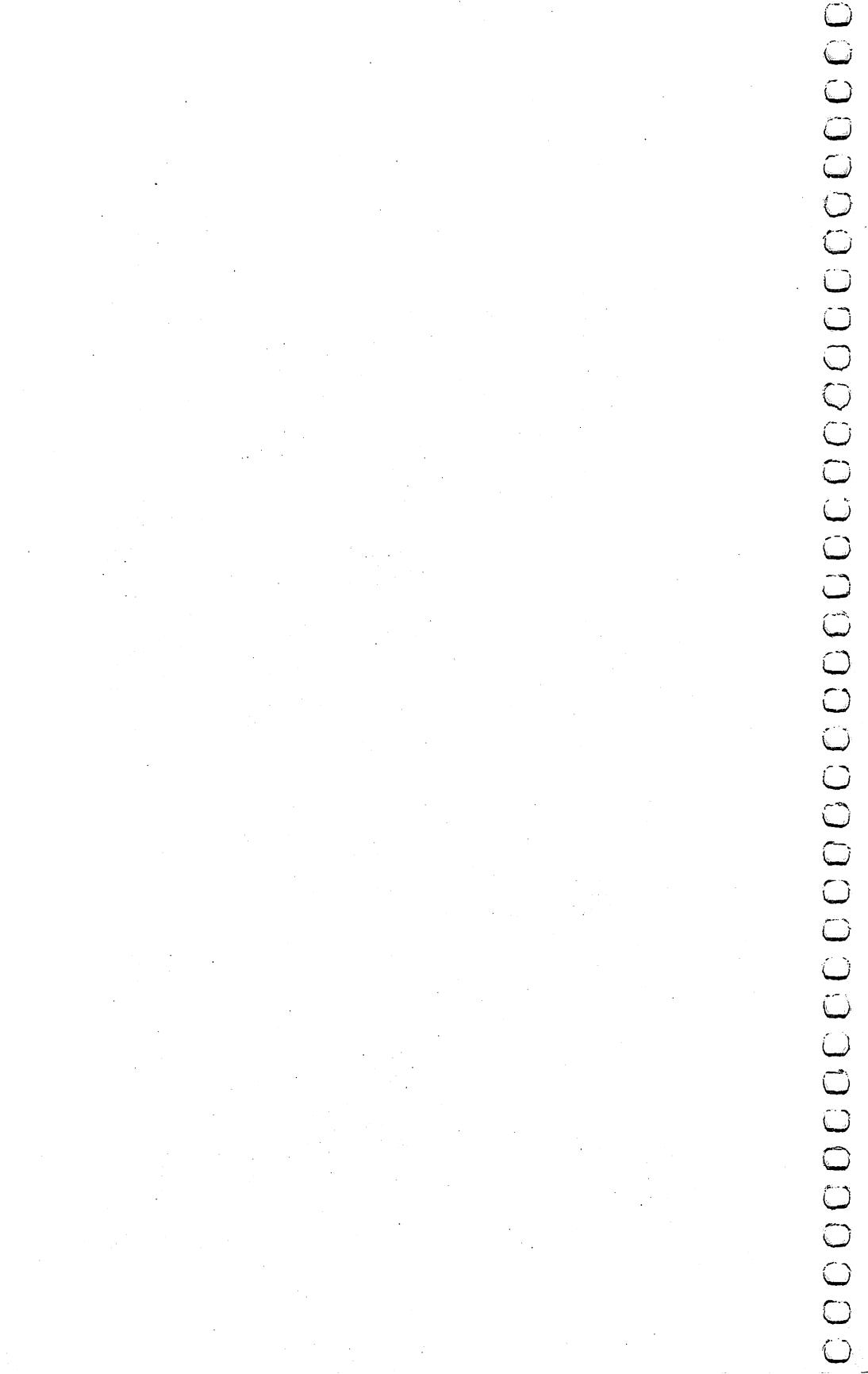
**Figure 2-11. Multicolored Bitmapped Mode**



A diagonal line in a cell of a standard bitmapped screen.

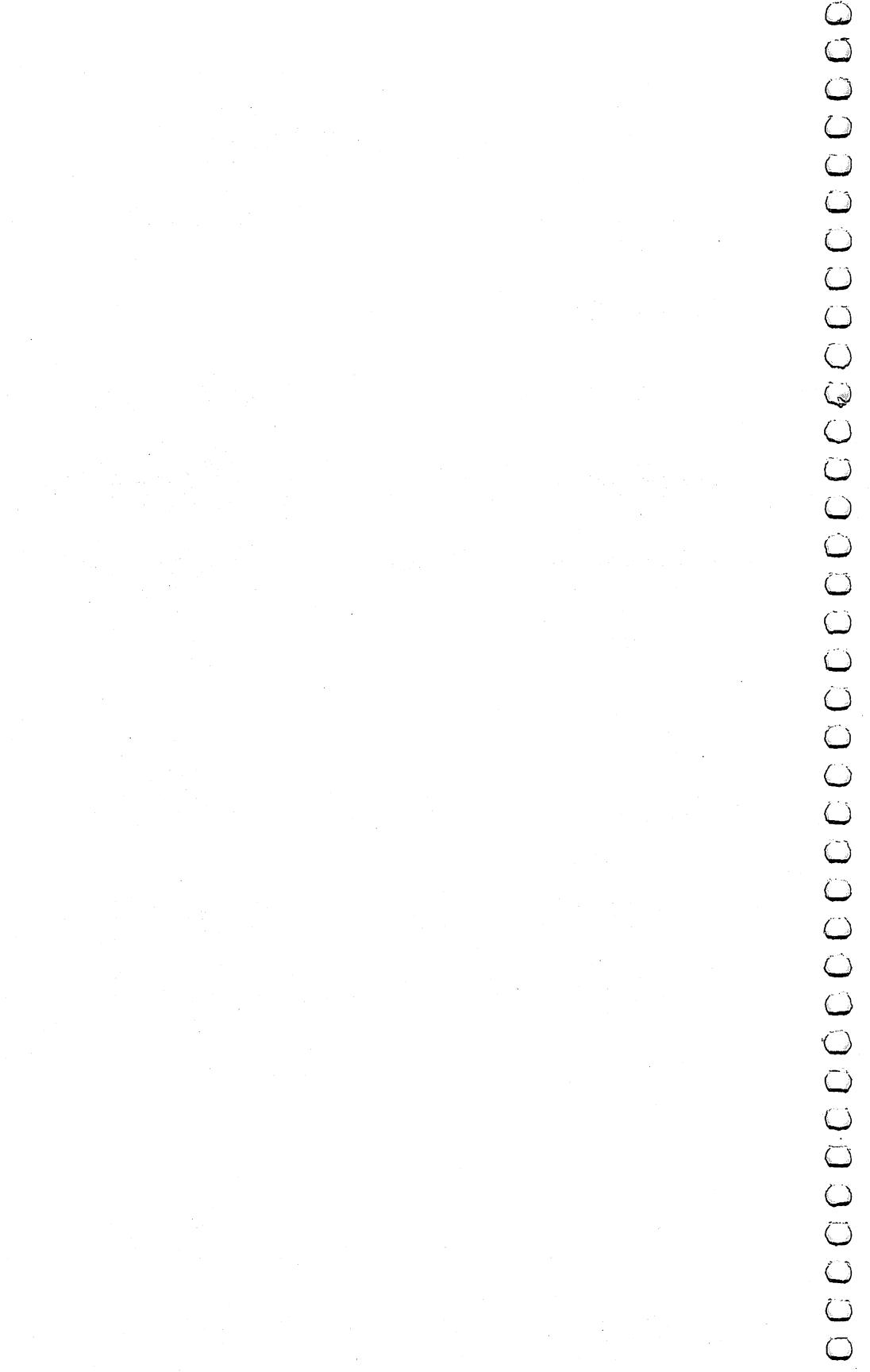
The same bit pattern when displayed as double dots in multicolor bitmapped mode.





# 3

# Animation



# Animation

**3** When you watch a movie, the images you see appear to be moving on the screen. The movement is an illusion produced by projecting a series of slightly different images on the screen in quick succession. Each image must be similar enough to the previous one that it seems to be the same person or scene, and different enough that it will seem that movement has taken place.

## Movement

We can accomplish the same effect with the computer by displaying shapes, erasing them, and displaying them again in positions that are progressively further and further away from their original positions.

Here is a program that produces a spot on the screen, erases it, and displays it one position to the right or left of its previous position. The program does this by preceding the displayed character with a string (A\$) that is made up of cursor-right characters. By increasing or decreasing the number of cursor-right characters contained in A\$, the dot appears to move left and right.

```
10 A$ = "{RIGHT}": B$ = "Q"
20 PRINT "{CLR}";
30 FOR R=0 TO 35
40 PRINT "{HOME}{ 7 {DOWN}}";A$;B$;"{ 2 {LEFT}} ";
50 A$=A$+"{RIGHT}"
60 FOR G=0 TO 10
70 NEXT
80 NEXT
90 FOR R=0 TO 35
100 PRINT "{HOME}{ 7 {DOWN}}";A$;B$;" ";
110 A$=LEFT$(A$,LEN(A$)-1)
120 FOR G=0 TO 10
130 NEXT
140 NEXT
150 GOTO 30
```

## Moving Objects

This is one of the simplest forms of animation for the 64 computer — moving objects. With this method, all you need to do is determine which character you wish to move and display it on the

# Animation

screen. By repeatedly erasing and printing the shape in adjacent positions on the screen, progressing in one direction or another, it produces the illusion of movement.

**Coherent and incoherent movement.** In order to represent movement realistically, it is necessary to distinguish between various kinds of movement. Coherent movement means any motion or apparent motion that suggests direction. For example, a bullet flashing out of the barrel of a gun has clear direction and should be represented on the computer by coherent movement. On the other hand, water boiling in a pot would have incoherent movement. Although the water is clearly moving, no real direction is implied. The movement is random.

```
1 PRINT "{CLR}{ 11 {DOWN}}{ 8 {RIGHT}}{A}{ 7 _}{S}
      { 23 {SPACES}}";
2 PRINT "{ 8 {SPACES}}-{ 6 {SPACES}}-{Q}{RVS}
      { 6 {SPACES}}{OFF} "
3 PRINT "{ 8 {SPACES}}{ 2 _}{ 5 {SPACES}}{ 2 _}
      { 8 {SPACES}}"
4 PRINT "{ 8 {SPACES}}-J{ 5 {SPACES}}K-
5 PRINT "{ 8 {SPACES}}J{ 7 *}{K"
6 PRINT "{ 8 {SPACES}}{I}{E}{ 5 {I}}{E}{I} "
7 PRINT "{ 5 {SPACES}}{ 15 {T}} "
20 B$=Q"
50 A$(0)="{HOME}{ 10 {DOWN}}{ 10 {RIGHT}}"
60 A$(1)="{HOME}{ 9 {DOWN}}{ 10 {RIGHT}}"
70 A$(2)="{HOME}{ 10 {DOWN}}{ 10 {RIGHT}}"
80 A$(3)="{HOME}{ 8 {DOWN}}{ 10 {RIGHT}}"
90 A$(4)="{HOME}{ 9 {DOWN}}{ 10 {RIGHT}}"
100 A$(5)="{HOME}{ 9 {DOWN}}{ 12 {RIGHT}}"
110 A$(6)="{HOME}{ 10 {DOWN}}{ 9 {RIGHT}}"
120 A$(7)="{HOME}{ 12 {DOWN}}{ 9 {RIGHT}}"
130 PRINT A$(RND(0)*7);B$;
140 FOR F=0TO5
150 NEXT
160 PRINT "{LEFT} ";
170 GOTO 130
```

Vibration is another example of incoherent movement. In vibration, however, the movement is regular, back and forth, and usually fast. For instance, the ringer in a telephone produces a kind of vibration, and by designing your images carefully, you can use shapes and movement to give the viewer the impression of seeing the object that is "causing" a sound.

```
10 PRINT "{CLR}{ 10 {DOWN}}{ 3 {RIGHT}}"
40 PRINT "{ 12 {SPACES}}{ 9 {@}}"
```

# Animation

```
50 PRINT" { 11 {SPACES}}N{ 2 {SPACES}}{A}{S}{@}{A}
    {S}{ 2 {SPACES}}M
60 PRINT" { 10 {SPACES}}{M}{ 2 {SPACES}}-{ 3 {@}}
    -{ 2 {SPACES}}{G}
70 PRINT" { 11 {SPACES}}{ 2 {T}}N{ 5 {SPACES}}M
    { 2 {T}}
80 PRINT" { 12 {SPACES}}{N} { 5 {T}} {G}
90 PRINT" { 12 {SPACES}}{N}{ 7 {P}}{H}
91 PRINT" { 13 {SPACES}}JK{ 3 {SPACES}}JK
92 PRINT" {HOME}{ 8 {DOWN}}{ 14 {RIGHT}}"
93 PRINT" { 9 {SPACES}}M{ 14 {SPACES}}N
94 PRINT" { 10 {SPACES}}M{ 2 {SPACES}}-
    { 6 {SPACES}}-{ 2 {SPACES}}N
95 PRINT" { 6 {SPACES}}{ 3 *}{ 3 {SPACES}}{ 9 {@}}
    { 3 {SPACES}}{ 3 _}
97 FOR Y=0 TO 15
99 PRINT" {HOME}{ 7 {DOWN}}{ 3 {RIGHT}}{ 3 {DOWN}}
    "
100 PRINT" { 6 {SPACES}}{ 3 *}{ 4 {SPACES}}
    { 9 {@}}{ 2 {SPACES}}{ 3 *}
110 PRINT" { 12 {SPACES}}N {A}{S}{@}{A}{S}{@}
    { 2 {SPACES}}M
120 PRINT" { 11 {SPACES}}{M}{ 2 {SPACES}}-{ 4 {@}}
    -{ 2 {SPACES}}{G}
130 PRINT" { 12 {SPACES}}{T}N{ 5 {SPACES}}M
    { 2 {T}}
135 PRINT" {HOME}{ 7 {DOWN}}{ 3 {RIGHT}}
    { 3 {DOWN}}"
140 PRINT" { 6 {SPACES}}{ 3 *}{ 3 {SPACES}}
    { 9 {@}}{ 3 {SPACES}}{ 3 *}
150 PRINT" { 11 {SPACES}}N{ 2 {SPACES}}{A}{S}{@}
    {A}{S}{ 2 {SPACES}}M{ 2 {SPACES}}"
160 PRINT" { 10 {SPACES}}{M}{ 2 {SPACES}}-
    { 3 {@}}-{ 2 {SPACES}}{G} "
170 PRINT" { 11 {SPACES}}{ 2 {T}}N{ 5 {SPACES}}M
    { 2 {T}}{ 3 {SPACES}}"
180 NEXT
185 PRINT" {HOME}{ 8 {DOWN}}{ 13 {RIGHT}}"
186 PRINT" { 27 {SPACES}}"
187 PRINT" { 27 {SPACES}}"
188 PRINT" { 12 {SPACES}}{ 9 {@}}{ 6 {SPACES}}"
190 FOR G=0 TO 1500:NEXT
200 GOTO 92
```

## How Fast (and When) Should It Move?

Movements are generally not just simple actions that are repeated, but are more often whole scenarios that unfold to create a story that leaves a total impression on the viewer.

# Animation

When a ball falls, for example, it may start out falling slowly, but as it drops, it gains momentum and eventually falls with sufficient speed that it rebounds into the air, bouncing to a height that may be only slightly lower than where it began. As it rises, it loses energy and slows down. Once it reaches its apex (the top of the bounce), it stops for a moment and then begins its descent again, bouncing less and less until it finally comes to rest on the ground. Here is a program that does that.

```
10 PRINT"[CLR]";
20 A$="{HOME}{ 12 {RIGHT}}": B$="Q"
25 FOR C=23 TO 0 STEP-1
30 FOR D=D TO C
40 PRINTA$;B$;"{LEFT}{UP} ";
50 A$=A$+"{DOWN}"
60 FOR W=0 TO (C-D)*4
70 NEXT
80 NEXT
90 FOR D=C TO 0 STEP-1
100 PRINTA$;B$;"{LEFT}{DOWN} ";
110 A$=LEFT$(A$,LEN(A$)-1)
120 FOR W=0 TO (C-D)*4
130 NEXT
140 NEXT
150 FOR R=0 TO 5*C : NEXT
170 NEXT
180 PRINT"[CLR]";A$;"{DOWN}";B$;"{HOME}"
```

## TO PRINT or to POKE

Up to this point, all of the characters that we placed on the screen were displayed using the PRINT command. This was done for two reasons. First, the PRINT statement does not require you to update the screen and color memory as you display characters, since the screen editor takes care of that. With POKE, you have to do the housekeeping yourself. Second, PRINT is much faster than POKE, as we've already seen in the example programs in the section of Chapter 2 on extended color mode.

Unfortunately, the PRINT function does have some disadvantages. For one thing, it uses up a great deal of memory because getting the characters to their proper locations on the screen requires a series of cursor movement characters for every possible position on the screen. Also, when the cursor reaches the bottom of the screen, the screen scrolls up, which can spoil a display.

POKE allows you to position objects on the screen without the danger of scrolling, but POKE is slow, and you must remember

# Animation

to POKE the characters *and* their corresponding color locations each time you display a character on the screen.

**POKE and screen memory.** Again, the screen is arranged as 25 rows of 40 characters. Each character is represented by a corresponding byte in screen memory. Screen memory usually resides between locations 1024 and 2023 (unless you change it by re-configuring memory).

To display a character, you must POKE the screen value for that character into screen memory and the value for a color that contrasts with the screen background in that character's color memory (between 55296 and 56295). So to POKE a character into the upper left-hand corner of the screen, you would POKE a screen code into location 1024 and a color code into 55296.

```
10 FOR F=0 TO 1023  
20 POKE F+1024,81  
30 POKE F+55296,14  
40 NEXT
```

**Programming movement.** To make a character seem to move, you must POKE a blank into the old location and POKE the character into an adjacent location. To move one position left, you subtract 1 from the current character position. To move right, you add one. To move up one position, you subtract 40 from the current character address, and to move down, you add 40. The address changes by 40 bytes for each vertical movement because a row on the screen is exactly 40 characters wide. The character position exactly one row up or down is always exactly 40 bytes away in screen memory. (See Figure 3-1.).

**Figure 3-1. Adjacent Screen Positions**

	1024	1025	1026	1027	1028	1029	1030	...
1024								
1064				1067 (up)				
1104			1106 (left)	1107	1108 (right)			
1144				1147 (down)				

```
10 PRINT "[CLR]{ 9 {RIGHT}}HORIZONTAL MOVEMENT"  
20 FOR R=500 TO 519  
30 POKE 55296+R,14  
40 POKE 1024+R,81
```

# Animation

```
50 FOR N=0 TO 10:NEXT
60 POKE R+1024,32
70 NEXT
80 FOR R=519 TO 480 STEP -1
90 POKE 55296+R,14
100 POKE 1024+R,81
110 FOR N=0 TO 10:NEXT
120 POKE R+1024,32
130 NEXT
140 FOR R=480 TO 500
150 POKE 55296+R,14
160 POKE 1024+R,81
170 FOR N=0 TO 10:NEXT
180 POKE R+1024,32
190 NEXT
200 PRINT"CLR){ 9 {RIGHT}}VERTICAL MOVEMENT"
210 FOR R=500 TO 60 STEP -40
220 POKE 55296+R,14
230 POKE 1024+R,81
240 FOR N=0 TO 10:NEXT
250 POKE R+1024,32
260 NEXT
270 FOR R=60 TO 980 STEP 40
280 POKE 55296+R,14
290 POKE 1024+R,81
300 FOR N=0 TO 10:NEXT
310 POKE R+1024,32
320 NEXT
330 FOR R=980 TO 500 STEP -40
340 POKE 55296+R,14
350 POKE 1024+R,81
360 FOR N=0 TO 10:NEXT
370 POKE R+1024,32
380 NEXT
390 GOTO 10
```

**Diagonal movement.** Diagonal movement is only a little more complex. A movement upward and to the left is programmed just the way it sounds; an upward movement plus a leftward movement. Since left is  $-1$  and up is  $-40$ , an up-left move is  $-41$ . An up-right move is  $-40 + 1$ , or  $-39$ ; a down-right move is  $40 + 1$ , or  $41$ , and a down-left move is  $40 - 1$ , or  $39$ .

## Character Animation

All the movement we have programmed so far has been movement of a single character from one location to another. But that is only part of animation. To give a more complete illusion of reality,

# Animation

there needs to be change *within* the character as well. For example, when the bouncing ball hit the ground, it would have looked more realistic if it had flattened somewhat on the bottom.

If you wanted to create an animated sequence of a pitcher delivering his fastball, you probably wouldn't want to move the figure of the pitcher from one place in screen memory to another. Instead, you would want to change the *shape* of the figure to suggest the windup, the delivery, and the recovery from the pitch.

The terms *animation* and *movement* can be used almost interchangeably, but for clarity let's use *movement* to mean moving a figure from one place on the screen to another and *animation* to mean changing the shape of the figure to give the illusion of smooth action. If we were programming a human figure running across the screen, the process of making the figure move from one spot to the next would be movement, and the process of changing the shape to make the figure's arms and legs change position would be animation.

How do you change the shape displayed at a certain character position on the screen?

You can POKE or PRINT different characters in sequence at the same location. The characters can be built-in characters or characters you designed yourself to be effective in suggesting motion.

Or you can relocate the character set and, while the program is running, change the patterns stored in the eight-byte block of memory that defines the character's shape.

## Redefining Characters

**Programming characters on the fly.** With this method, the screen codes in screen memory don't change. If the character you are animating is, say, an ampersand (&), the code for an ampersand, 38, remains in the same location in screen memory the whole time.

Instead, you change the shape by altering the eight-byte pattern in character memory. Of course, you cannot change the character patterns in the ROM character set — those memory locations are built into the hardware and can't be changed. So you must first put character memory into RAM, where you can change it. (See the early part of Chapter 1 for methods of moving character memory.)

**Find the pattern.** To change a particular character's shape, you have to find the eight-byte pattern in character memory. The

# Animation

screen code is your index to the character set. Patterns are stored in the same order as the screen code. The pattern for screen code 0 (the @ character) is at the start of the character set; the pattern for screen code 1 (A) is next. To get the actual address of the first byte of the pattern, start with the screen code, multiply it by eight, and add it to the starting address of character memory. If character memory has been relocated at 14336, you will find the first byte of the pattern for X (screen code 24) at location 14528, or  $8*24 + 14336$ .

Once you have the start of the pattern, you can change it however you like, either all or part. Here is a program that animates a single character by changing its pattern while the program is running. A small wheel seems to be rotating — the illusion of movement all within a single character.

```
10 POKE 56578, PEEK(56578) OR 3
20 POKE 56576, (PEEK(56576) AND 252) OR 0
30 POKE 53272, (PEEK(53272) AND 240) OR 2
40 POKE 648, 196
50 POKE 56334, PEEK(56334) AND 254
60 POKE 1, PEEK(1) AND 251
70 FOR R=53248 TO 55296
80 POKE R-2048, PEEK(R): NEXT
90 POKE 1, PEEK(1) OR 4
100 POKE 56334, PEEK(56334) OR 1
110 PRINT "[CLR]{ 8 {DOWN}}{ 12 {RIGHT}}@"
120 DATA 30,33,33,33,33,33,30,0,0
122 DATA 60,66,66,66,66,60,0,0
124 DATA 120,132,132,132,132,120,0,0
126 DATA 0,120,132,132,132,132,120,0
128 DATA 0,0,120,132,132,132,132,120
130 DATA 0,0,60,66,66,66,66,60
132 DATA 0,0,30,33,33,33,33,30
134 DATA 0,30,33,33,33,33,33,30,0
136 DATA 30,33,33,33,33,30,0,0
150 FOR N=0 TO 7
160 FOR R=0 TO 7
165 READ A
170 POKE R+51200, A
180 NEXT
190 NEXT
200 RESTORE
210 GOTO 150
```

## Complex Characters

Sometimes you'll want to create a figure that's larger than one character's eight-by-eight grid can hold. You can always combine

# Animation

several characters to make one drawing. Here is a short program that combines the rotating wheel with other characters to make a car that moves across the screen while its animated wheels rotate. As a general rule, the more characters you combine into one large figure, the more sluggish and jerky its movement will appear. But by setting up figures as strings and PRINTing them, you can get reasonable speed, while the constant character redefinition that rotates the wheels improves the illusion.

```
10 POKE 56578, PEEK(56578) OR 3
20 POKE 56576, (PEEK(56576) AND 252) OR 0
30 POKE 53272, (PEEK(53272) AND 240) OR 2
40 POKE 648, 196
50 POKE 56334, PEEK(56334) AND 254
60 POKE 1, PEEK(1) AND 251
70 FOR R=53248 TO 55296
80 POKE R-2048, PEEK(R): NEXT
90 POKE 1, PEEK(1) OR 4
94 X$ = "{HOME}{ 8 {DOWN}}{ 63 {RIGHT}}"
95 A$ = "{ 8 {SPACES}}U{ 4 * }I{ 19 {SPACES}}"
96 A$=A$+"{ 15 {SPACES}}-{Y}-{ 2 {Y}}-{ 16 {SPACES}}"
97 A$=A$+"{ 13 {SPACES}}I{ 4 * }{W}{R}-{ 2 {O}}J
    { 2 * }I{ 13 {SPACES}}"
98 A$=A$+"{ 13 {SPACES}}K U*I {T}{ 3 {SPACES}}U*I
    -{ 13 {SPACES}}"
99 A$=A$+"{ 13 {SPACES}}J*K@J{ 5 * }K@JK "
100 POKE 56334, PEEK(56334) OR 1
110 PRINT "[CLR]"
120 DATA 30,33,33,33,33,30,0,0
122 DATA 60,66,66,66,66,60,0,0
124 DATA 120,132,132,132,132,120,0,0
126 DATA 0,120,132,132,132,132,120,0
128 DATA 0,0,120,132,132,132,132,120
130 DATA 0,0,60,66,66,66,66,60
132 DATA 0,0,30,33,33,33,33,30
134 DATA 0,30,33,33,33,33,30,0
136 DATA 30,33,33,33,33,30,0,0
140 FOR K=0 TO 2
150 FOR N=0 TO 7
160 FOR R=0 TO 7
165 READ A
170 POKE R+51200,A
180 NEXT
185 PRINT X$; A$: X$=LEFT$(X$, LEN(X$)-1)
190 NEXT
200 RESTORE
210 NEXT
```

# Animation

You'll notice that when this program finishes, the new character set remains. Pressing RUN/STOP-RESTORE doesn't bring everything back to normal. In fact, RUN/STOP-RESTORE makes it almost impossible to read the screen. Of course, you can always turn off the computer and start over, but in this case the solution is much simpler. Simply press RUN/STOP-RESTORE and type POKE 648,4. You may not be able to see the letters as you type them, but the computer will see them and obey. This command tells BASIC where the screen is located now. You may want to add it as a line at the end of the program, to make a more orderly exit from the program.

## Switching Characters

Another way to animate an individual character position on the screen is to define beforehand all the characters you'll need to create the illusion, and then POKE or PRINT them in sequence at the same location. With the rotating wheel, we altered the pattern in character memory. Now we have many patterns to choose from, and either POKE the screen codes or PRINT the characters. This process requires a longer setup time while the program is initializing, but the animation can be much faster and smoother when it's finally carried out. This program moves a small cart across the screen.

```
10 POKE 56578, PEEK(56578) OR 3
20 POKE 56576, (PEEK(56576) AND 252) OR 0
30 POKE 53272, (PEEK(53272) AND 240) OR 2
40 POKE 648, 196
50 POKE 56334, PEEK(56334) AND 254
60 POKE 1, PEEK(1) AND 251
70 FOR R=53248 TO 55296
80 POKE R-2048, PEEK(R): NEXT
90 POKE 1, PEEK(1) OR 4
95 PRINT "[CLR]"
100 POKE 56334, PEEK(56334) OR 1
110 P$ = "[HOME]{ 8 {DOWN}}{ 32 {RIGHT}}"
120 DATA 30,33,33,33,33,30,0,0
122 DATA 60,66,66,66,66,60,0,0
124 DATA 120,132,132,132,132,120,0,0
126 DATA 0,120,132,132,132,132,120,0
128 DATA 0,0,120,132,132,132,132,120
130 DATA 0,0,60,66,66,66,66,60
132 DATA 0,0,30,33,33,33,33,30
134 DATA 0,30,33,33,33,33,33,30,0
136 DATA 30,33,33,33,33,33,30,0,0
150 FOR N=0 TO 63
```

# Animation

```
165 READ A
170 POKE N+51200,A
190 NEXT
200 G$(0) = "@" : G$(1) = "A" : G$(2) = "B" : G$(3) = "C" : G$(4)
     = "D" : G$(5) = "E" : G$(6) = "F"
210 G$(7) = "G"
211 LS = "{ 11 {DOWN}}{ 40 {U}}}"
215 PRINT "{CLR}"; LS; : SS$ = "" : FOR U=0 TO 3
220 FOR I=0 TO 7
230 PRINT P$; SS$;
231 PRINT "{ 5 {SPACES}}{R}{ 28 {SPACES}}";
232 PRINT "{ 10 {SPACES}}U{E}*I ";
236 PRINT P$; SS$; "{ 3 {DOWN}}{UP}{LEFT}
     { 6 {SPACES}}"; G$(I); G$(((I+4) AND 7)); "
238 SS$=SS$+"{LEFT}"
240 NEXT:NEXT
250 GOTO 215
```

In fact, this program runs so fast that you may want to slow it down by inserting this line: 239 FOR Q = 0 TO 60:NEXT. To slow it down even more, change the 60 to a higher number.

## Smooth Movement with Characters

One of the limitations of character graphics is that you can't move a figure in smaller increments than one full character position on the screen. Your figures always jump a full eight dots with every movement. This results in the rather coarse and jerky movement you may have noticed in the examples above.

It is possible to make a figure move from one character position to the next as smoothly as we animated the rotating wheel *within* a character position. You must begin by defining several sub-characters that represent different stages in a character's movement, as illustrated in Figure 3-2.

To see this in action, let's use several of the built-in characters, the vertical bar graphics characters. These are already designed to do the job we want — each bar is in a position slightly different from all the others. First, this program shows the movement pattern within one character position:

```
10 PRINT "{CLR}"
20 A$ = "{HOME}{ 8 {DOWN}}{ 11 {RIGHT}}"
30 PRINT A$; "{G}"
40 PRINT A$; "G"
50 PRINT A$; "-"
60 PRINT A$; "H"
70 PRINT A$; "N"
80 PRINT A$; "
```

# Animation

```
90 FOR T=0TO100  
100 NEXT  
110 GOTO 20
```

Now, when the sequence brings the bar to the extreme edge of the character position, we'll change the character position by one, so that when the sequence starts over, it seems to be continuing a smooth movement across the full width of the screen.

```
10 PRINT"{}CLR{}"  
20 A$="{}{HOME}{ 8 {DOWN}}"  
25 FOR G=0 TO 39  
30 PRINTA$;"{}G{}"  
40 PRINTA$;"{}G"  
50 PRINTA$;"{}-"  
60 PRINTA$;"{}H"  
70 PRINTA$;"{}{N}  
80 PRINTA$;"{} "  
90 A$=A$+"{}{RIGHT}"  
100 NEXT  
110 GOTO 20
```

## Sprite Animation

Sprite movement is handled quite differently from character movement. The sprite isn't located in screen memory — you don't have to erase the sprite at its old location in order to move it to another. Instead, the VIC-II chip puts the sprite anywhere you want it on the screen, without disturbing anything in screen memory.

### Positioning Sprites

Each sprite has its own X and Y position register which determines how many dots in from the left edge (X) and down from the top (Y) the upper left-hand corner of the sprite should be placed. (See Table 3-1 for register locations.)

Instead of being limited to only 1000 possible positions on the screen, like a character, the sprite can be in any one of 64,000 positions. When it moves, it can move one dot at a time in any direction. Smooth movement is much, much simpler.

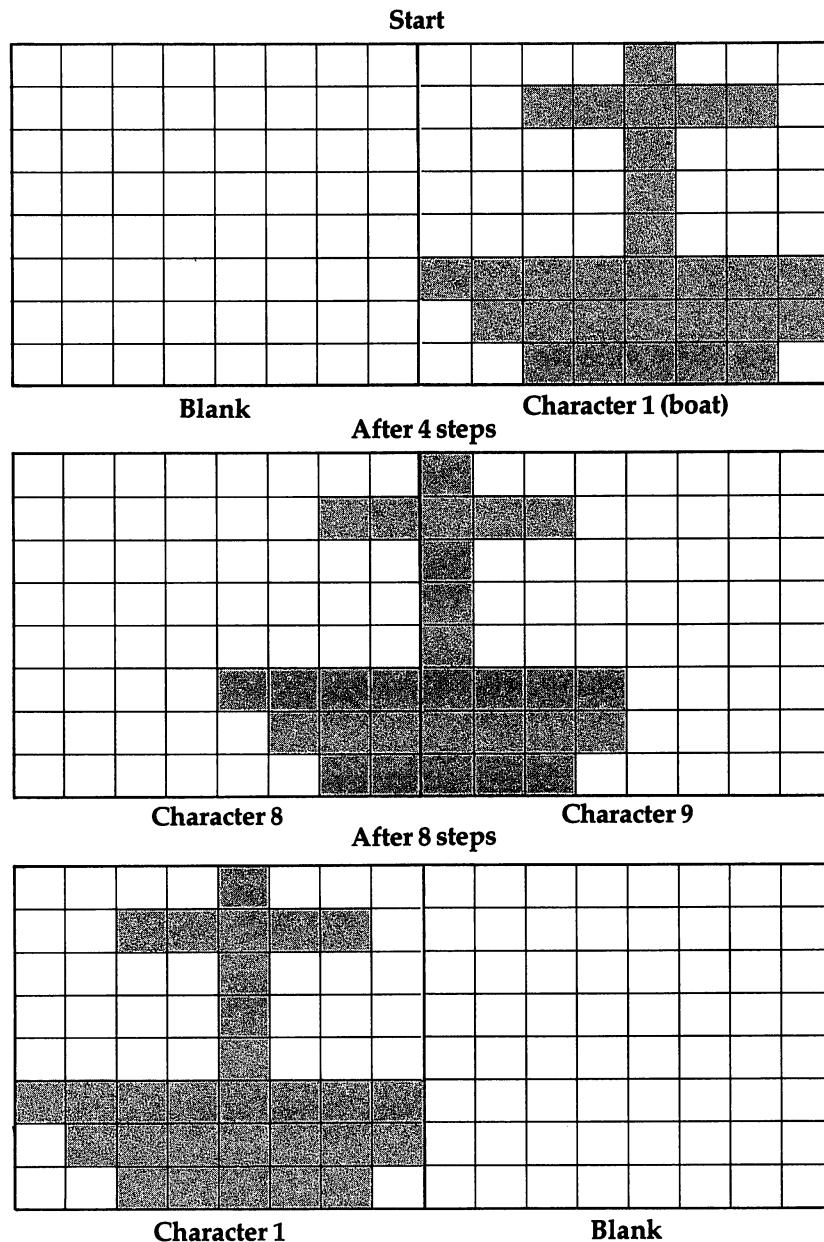
And since sprites are 21 bytes high and 24 bits wide (the same as *three* characters) you can create complex images that move all at once.

### Setting Up Sprites

First, we must enable the sprites by turning on the proper bit in

# Animation

**Figure 3-2. Smooth Character Movement**



This movement would require 16 characters for maximum smoothness.

# Animation

register 53269. For example, to enable sprite #0, turn on bit 0 of 53269:

**POKE 53269,1**

Next, we need to position the sprite somewhere on the screen. This is because the sprites are automatically positioned off-screen whenever you start up the computer and whenever you press RUN/STOP-RESTORE. POKEing the value 24 into the X position register and 50 into the Y position register will place sprite #0 precisely in the upper left-hand corner of the screen:

**POKE 53248,24:POKE 53249,50**

The last eight bytes of the 1K block that contains screen memory are used to tell the VIC-II chip where the sprite patterns can be found. Since you will undoubtedly be keeping track of the start of screen memory, you can always find the start of the sprite data pointers by adding 1016 to the screen memory start address. This will give you the location of the pointer for sprite #0.

What number do you use to point to the sprite data? Sprite data always starts at an address evenly divisible by 64. The pointer is always the address divided by 64. So if screen memory is located at 1024 and the data for sprite #0 begins at 12800, you would inform the VIC-II this way:

**POKE 2040,200**

The VIC-II is always informed of the starting address of the pattern for sprite *n* following this formula:

**POKE screen address + 1016 + *n*, data address / 64**

Of course, before the sprite will show up its pattern has to be defined. A sprite whose pattern consists entirely of zeros will be invisible. For now let's just fill the entire pattern with *on* bits by POKEing all 63 bytes of the pattern with 255. The sprite will be a solid rectangle:

```
10 PRINT"CLR"
20 POKE 53269,1: REM ENABLE SPRITE #0
30 POKE 53248,24: REM X POS
40 POKE 53249,50: REM Y POS
50 POKE 2040, 50
60 FOR R=3200 TO 3263
70 POKE R, 255
80 NEXT
```

Now we're ready to get this sprite moving. This routine calculates a circular path for the sprite.

# Animation

```
10 PRINT "[CLR]"
20 POKE 53269,1
30 X1=170:X2=250:Y1=130:Y2=130
50 POKE 2040, 250
70 FOR R=16000 TO 16063
80 POKE R, 255
90 NEXT
100 R1=SQR((X1-X2)↑2+((Y1-Y2)↑2)):R2=R1*.72
102 FORT=0TO2*{↑}STEP(1.5/R1):X=(R1*(COS(T)))+X1:
     Y=(R2*(SIN(T)))+Y1
110 POKE53248,X:POKE53249,Y:NEXT
140 GOTO 102
```

**Table 3-1. The Sprite Position Registers**

Memory Location	Sprite Control
53248	Sprite -0 X Position Register
53249	Sprite -0 Y Position Register
53250	Sprite -1 X Position Register
53251	Sprite -1 Y Position Register
53252	Sprite -2 X Position Register
53253	Sprite -2 Y Position Register
53254	Sprite -3 X Position Register
53255	Sprite -3 Y Position Register
53256	Sprite -4 X Position Register
53257	Sprite -4 Y Position Register
53258	Sprite -5 X Position Register
53259	Sprite -5 Y Position Register
53260	Sprite -6 X Position Register
53261	Sprite -6 Y Position Register
53262	Sprite -7 X Position Register
53263	Sprite -7 Y Position Register
53264	Sprite MSB Register

## Sprite Speed

One of the major problems in animating your pictures in BASIC is speed. BASIC makes for slow animation.

There are two reasons for this. First, BASIC is limited in how quickly it can move data from one place to another. POKE is a fairly slow function, and moving sprites requires you to POKE two bytes of data for every new sprite position.

Second, animated programs usually require one or more calculations per movement just to determine *where* to move. These calculations can be quite time-consuming, and they slow down movement.

# Animation

**Movement tables.** One way to increase the speed of a sprite is to eliminate the calculations that take place during the display time. This can be accomplished by calculating the values of the sprite's movements beforehand and placing the results of these calculations in a movement table. That way there's no waiting for hundreds or even thousands of calculations to be performed while the program is running. The values are immediately available. This routine takes a long time to set up, but the movement is relatively fast.

```
10 PRINT "[CLR]"
15 DIM X(360), Y(360)
20 POKE 53269, 1
30 X1=170:X2=250:Y1=130:Y2=130
40 R1=SQR(((X1-X2)↑2)+((Y1-Y2)↑2)):R2=R1*.72
50 POKE 2040, 250
70 FOR R=16000 TO 16063
80 POKE R, 255
90 NEXT
100 FORT=0TO2*{↑}STEP(1.5/R1):X(N)=(R1*(COS(T)))+
    X1:Y(N)=(R2*(SIN(T)))+Y1:N=N+1
105 NEXT
120 FOR R=0 TO N:POKE53248,X(R):POKE53249,Y(R):
    NEXT
140 GOTO 120
```

**Bigger steps.** Another way to increase sprite speed is to move the sprite farther with each step. This will reduce the number of setup calculations. It also halves the number of POKEs required to move the sprite from one location to another. However, there is some loss in smoothness.

```
100 FORT=0TO2*{↑}STEP(5/R1):X(N)=(R1*(COS(T)))+X1
    :Y(N)=(R2*(SIN(T)))+Y1:N=N+1
```

## Sprites and the wide Screen

The 64 screen is 320 dots wide by 200 dots high. Sprites should be able to occupy any of those 64,000 positions. However, it is impossible to POKE a value higher than 255 in any memory location in the 64, including sprite position registers. This makes no difference with the Y position registers, but the X register, by itself, can't position a sprite any farther to the right than column 255.

**Hiding in the wings.** When the 64 is first powered up, the position registers all contain zeros. This puts the sprites behind the border of the screen, since the upper left-hand corner of the display area is at sprite coordinates 24,50. The display area is like a stage, and the border is the curtain where the sprites can hide.

# Animation

when they exit from the screen. This program shows how sprites can move on and off the stage:

```
10 PRINT"[CLR]"
20 POKE 53269,1
30 FOR R=3200 TO 3263
40 POKE R, 255
50 NEXT
70 POKE 53249,100
80 FOR R=0 TO 255
100 POKE 53248, R
110 NEXT
120 FOR R=255 TO 0 STEP -1
130 POKE 53248, R
140 NEXT
150 FOR R=0 TO 500:NEXT
160 GOTO 80
```

This procedure allows you to move unwanted sprites off the screen instead of disabling them. This can save you from having to make your program calculate when to turn sprites on and off during a program. This is especially important if your program is moving a lot of sprites or doing a lot of calculations.

```
10 PRINT"[CLR]"
20 POKE 53269,1
30 FOR R=3200 TO 3263
40 POKE R, 255
50 NEXT
70 POKE 53249,100
80 POKE 53248,150
100 POKE 53248, 0
110 GOTO 80
130 GOTO 100
```

It is also much faster to toggle the position registers than it is to toggle the sprite enable register.

```
10 PRINT"[CLR]"
20 POKE 53269,1
30 FOR R=3200 TO 3263
40 POKE R, 255
50 NEXT
70 POKE 53249,100
80 POKE 53248,150
100 POKE 53269, 0
120 POKE 53269, 1
130 GOTO 100
```

# Animation

**Spanning the gap.** This still leaves us with the problem of getting the sprite past the invisible barrier toward the right-hand side of the screen. Since the screen is 320 bits wide and the position register has a capacity of 255, we must find a way of increasing the capacity of the position registers. One more bit would let us record values up to 511 — more than enough for our needs.

The 64 provides that extra bit. It can't be included right at the X register. Instead, the extra bits for all eight sprites are combined in one eight-bit byte at location 53264, called the MSB register. Bit 0 of this register holds the MSB (most significant bit) of the X location of sprite #0; bit 7 holds the MSB of sprite #7.

The VIC-II checks the MSB register to find out whether the sprite should be to the left or the right of the dividing line. If the sprite's MSB is *on* (1), the sprite is to the right of the line; if the MSB is *off* (0), the sprite is to the left.

Your program must adjust the MSB register whenever a sprite crosses the dividing line. If sprite *n* is moving from left to right and the value in the X register reaches 255, bit *n* of the MSB register must be turned on, while the X register is set back to 0 to start counting over again on the right side of the line. If sprite *n* is moving from right to left and the value in the X register reaches 0, and if bit *n* in the MSB register is *on*, then it must be turned *off* and the X register must be set back to 255 to begin counting down again.

```
10 PRINT"CLR"
20 POKE 53269,1:POKE 2040,50
30 FOR R=3200 TO 3263
40 POKE R, 255
50 NEXT
70 POKE 53249,100
80 FOR R=0 TO 350
100 POKE 53264,ABS(256<R):POKE53248,RAND255
110 NEXT
115 FOR R=0 TO 500:NEXT
120 FOR R=350 TO 0 STEP -1
130 POKE 53264,ABS(256<R):POKE53248,RAND255
140 NEXT
150 FOR R=0 TO 500:NEXT
160 GOTO 80
```

## Animating the Sprite

In addition to moving sprites from place to place on the screen, it is also possible to animate the sprites by changing their shape, just as we animated the car wheels in the character animation section of this chapter.

# Animation

It is possible but highly impractical to program sprite patterns on the fly, at least when you are programming from BASIC. Sprite patterns are made up of 63 bytes — 504 individual bits — and it would take an intolerably long time for any real animation to take place.

Fortunately, it is possible to program dozens of different sprite shapes starting at almost any of the thousand 64-byte boundaries in the Commodore 64. Though a sprite can only use one pattern *at a time*, it can be switched from pattern to pattern simply by POKEing a new pointer into the sprite pointer table at *screen memory* + 1016. For instance, let's say you have sprite patterns set up at 12800, 12864, 12928, 12992, 13056, and 13120. Screen memory is at 1024 so that the pointer for sprite #6 will be at 1024 + 1016 + 6, or 2046. This loop would cycle through the six patterns so quickly that the eye could hardly register the changes:

**FOR I = 200 TO 205:POKE 2046,I:NEXT**

(The numbers 200 through 205 are the addresses 12800 through 13120, divided by 64.)

Here is a program that switches back and forth between sprite patterns:

```
1 DATA 142,56,227,28,113,199,56,227,142,113,199,2
  8,227,142,56,199,28,113,142
2 DATA 56,227,28,113,199,56,227,142,113,199,28,22
  7,142,56,199,28,113,142,56,227
3 DATA 28,113,199,56,227,142,113,199,28,227,142,5
  6,199,28,113,142,56,227,28,113
4 DATA 199,56,227,142,85,113,199,28,227,142,56,19
  9,28,113,142,56,227,28,113,199
5 DATA 56,227,142,113,199,28,227,142,56,199,28,11
  3,142,56,227,28,113,199,56,227
6 DATA 142,113,199,28,227,142,56,199,28,113,142,5
  6,227,28,113,199,56,227,142
7 DATA 113,199,28,227,142,56,199,28,113,85,199,28
  ,113,142,56,227,28,113,199,56
8 DATA 227,142,113,199,28,227,142,56,199,28,113,1
  42,56,227,28,113,199,56,227
9 DATA 142,113,199,28,227,142,56,199,28,113,142,5
  6,227,28,113,199,56,227,142
10 DATA 113,199,28,227,142,56,199,28,113,142,56,2
  27,28,113,199
15 REM   FOR R=0 TO 998:PRINT"{}RVS}{BLUE} ":";NEXT
17 GOTO 700
20 FOR R=0 TO 190
30 READ A
40 POKE R+16000, A
```

# Animation

```
50 NEXT
60 POKE 53269,1
80 POKE 53248,137
90 POKE 53249,131
100 POKE 53271, 1
110 FOR R=250 TO 252
120 POKE 2040, R
125 FOR G=0TO100:NEXT
130 NEXT
150 GOTO 110
700 PRINT"{}CLR}{ 13 {DOWN}}";
705 PRINT"{}ORANGE}{ 15 {SPACES}})Q"
710 PRINT"{}ORANGE}{ 13 {SPACES}}){D}{ 3 {I}}{F}"
712 PRINT"{}WHITE}{ 15 {RIGHT}}){K}"
713 PRINT"{}WHITE}{ 15 {RIGHT}}){K}"
714 PRINT"{}WHITE}{ 15 {RIGHT}}){K}"
715 PRINT"{}WHITE}{ 15 {RIGHT}}){K}"
717 PRINT"{}WHITE}{ 15 {RIGHT}}){K}"
720 PRINT"{}ORANGE}{ 13 {SPACES}}){D}{ 3 {I}}{F}"
730 PRINT"{}BLACK}{RVS}{ 15 {RIGHT}})-"
740 PRINT"{}BLACK}{RVS}{ 15 {RIGHT}})-"
750 PRINT"{}RVS}{BLACK}{ 15 {RIGHT}})-"
760 PRINT"{}BLACK}{RVS}{ 15 {RIGHT}})-"
770 PRINT"{}RVS}{BLACK}{ 15 {RIGHT}})-"
780 PRINT"{}BLACK}{RVS}{ 15 {RIGHT}})-"
790 PRINT"{}RVS}{BLACK}{ 15 {RIGHT}})-"
800 PRINT"{}RVS}{BLACK}{ 15 {RIGHT}})-"
810 PRINT"{}RVS}{BLACK}{ 15 {RIGHT}})-";
820 GOTO 20
```

Also, you can always set the pointers of several sprites to read the *same* pattern in memory. The sprites will then be identical.

## Expanding Sprites

Sprites can be displayed at twice their normal size by using the X (horizontal) and Y (vertical) expand registers. By ORing and POKEing a 1 into bit *n* of the X or Y expand registers for sprite *n*, you will double the displayed size of that sprite.

X-Expand Register: Location 53277

Y-Expand Register: Location 53271

## Priorities

The television screen is flat, but you can still create an illusion of depth by having sprites seem to pass in front of or behind objects on the screen. Sprites have built-in priority. When two sprites overlap, the sprite with the higher priority is displayed, and the portion of the other sprite that overlaps is *not* displayed. This

# Animation

gives the illusion that the high-priority sprite is in front of the low-priority sprite.

Sprite 0 has the greatest priority among sprites, and sprite 7 has the least. This means that if sprite 0 and sprite 4 overlap, sprite 0 will be displayed, while if sprite 4 and sprite 7 overlap, sprite 4 will be "in front."

The priority register at 53275 allows you to set the priority of the sprites relative to the background. A 1 in bit 0 gives sprite #0 priority over background objects; a 0 in bit 7 gives the background priority over sprite #7; and so on. Even when the background has priority, sprites will never seem to be behind the color displayed wherever 0 bits or 00 bit-pairs are located in screen memory or character patterns. However, sprites with an *off* bit in the priority register will seem to be behind the colors displayed at 1 bits or 01, 10, and 11 bit-pairs in screen memory or character patterns.

This program creates a three-dimensional solar system. The planet seems to change size as it rotates around the sun, and disappears when it passes "behind" the sun. By adding more intermediate planet sizes you could create a very smooth illusion of three-dimensional rotation.

```
1 DATA 0,0,0,0,0,0,0,255,0,3,255,224,15,255,240,3  
    1,255,248,63,255,252,63,255  
2 DATA 252,127,255,254,127,255,254,127,255,254,12  
    7,255,254,127,255,254,63,255  
3 DATA 252,31,255,252,31,255,248,15,255,240,7,255  
    ,224,1,255,128,0,0,0,0,0,0,0  
4 DATA 0,0,0,1,255,128,15,255,240,31,255,248,63,2  
    55,252,63,255,252,127,255,254  
5 DATA 255,255,255,255,255,255,255,255,255,255,25  
    5,255,255,255,255,255,255  
6 DATA 255,255,255,127,255,255,63,255,254,63,255,  
    252,31,255,248,15,255,240,1  
7 DATA 255,128,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
    126,0,3,255,192,7,255,224,15  
8 DATA 255,240,15,255,240,31,255,248,31,255,248,3  
    1,255,248,31,255,248,15,255  
9 DATA 240,15,255,224,7,255,224,3,255,192,0,126,0  
    ,0,0,0,0,0,0,0,0,0,0,0,0,0  
10 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,126,0,1,255  
    ,128,3,255,192,7,255,224,7,255  
11 DATA 224,7,255,224,7,255,224,3,255,192,1,255,1  
    28,0,126,0,0,0,0,0,0,0,0,0,0,0  
12 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  
    ,0,0,0,0,0,0,0,0,0,0,0,124,0  
13 DATA 0,254,0,1,255,0,1,255,0,0,254,0,0,124,0,0  
    ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

# Animation

```
14 DATA 0,0,0,0,0,0,0,0
15 PRINT"CLR){WHITE}.{RIGHT}{ 2 {DOWN}}.
    { 4 {RIGHT}}.{DOWN}{RIGHT}.{ 3 {DOWN}}..{UP}
    {LEFT}.{ 2 {RIGHT}}.{ 5 {RIGHT}}.
    { 10 {DOWN}}.{RIGHT}.{UP}{LEFT}.{RIGHT}.
    { 3 {RIGHT}}.{ 4 {RIGHT}}.{ 8 {DOWN}}.
    { 2 {RIGHT}}.{ 2 {RIGHT}}.."
17 POKE 53281, 0: POKE 53280, 0
20 FOR R=16000 TO 16318
30 READ A: POKE R,A: NEXT
40 POKE 2041, 251: POKE 2040, 250
50 POKE 53269,7
60 POKE 53248,170: POKE 53249, 130
70 POKE 53250,170: POKE 53251, 130
75 POKE 53252,170: POKE 53253, 130
90 POKE 53287,11:POKE 53289, 11
100 POKE 53288,7
110 FOR R=170 TO 200
120 POKE 53248,R: FORG=0TO20:NEXT:NEXT
130 POKE 2040,252
140 FOR R=200 TO 230
150 POKE 53248,R: FORG=0TO20:NEXT:NEXT
160 POKE 2040, 253
170 FOR R=230 TO 200 STEP-1
180 POKE 53248,R: FORG=0TO20:NEXT:NEXT
190 POKE 2042, 254
200 POKE53248,0:FOR R=200 TO 170 STEP-1
210 POKE 53252,R: FORG=0TO20:NEXT:NEXT
220 FOR R=170 TO 140 STEP-1
230 POKE 53252,R: FORG=0TO20:NEXT:NEXT
240 POKE 2040, 253
250 POKE53252,0:FOR R=140 TO 110 STEP-1
260 POKE 53248,R: FORG=0TO20:NEXT:NEXT
270 POKE 2040, 252
280 FOR R=110 TO 140
290 POKE 53248,R: FORG=0TO20:NEXT:NEXT
300 POKE 2040, 250
310 FOR R=140 TO 170
320 POKE 53248,R: FORG=0TO20:NEXT:NEXT
330 GOTO 110
```

See Table 1-2 for other key sprite-control locations.

## Animating the Screen

Besides character and sprite movement and animation, you have another movement option — you can make the whole screen shift. There are three ways to animate the screen, just as characters and sprites can be animated in three different ways:

# Animation

**Redrawing the screen.** You can redefine the screen display while it is being displayed. In effect, this is what character movements do already.

**Screen flipping.** You can preprogram thousand-byte blocks of data as screen displays before starting the animation. Then you can switch back and forth from one data block to another, telling the VIC-II that each block in turn is now screen memory:

```
FOR I = 128 TO 240 STEP 16:POKE 53272,PEEK(53272)AND 15 OR  
I:NEXT
```

This line would tell the VIC-II to find screen memory at 8192, 9216, 10240, 11264, 12288, 13312, 14336, and 15360 in rapid succession. If displays were already set up at those locations, the illusion of movement would be very fast.

**Scrolling.** You can also move the entire screen as a unit by scrolling it. This is what the screen editor does when you PRINT or type something at the bottom of the screen. The screen then scrolls upward. You can also scroll from side to side.

Coarse scrolling is when the screen moves one entire character width or height (eight dots) at a time. This is the best scrolling method for a text screen, since it moves text on and off the screen a row or column of characters at a time.

Fine scrolling is when the screen moves only one dot at a time. This is the best technique for giving the illusion of animation, as in videogames where the player is supposed to be moving along a road, flying over a landscape, or zipping through space. Using this technique, a sprite can be standing still in the middle of the screen while the background moves "behind" it. The illusion of movement is very powerful.

Fine scrolling is such a time-consuming process that it can only be done effectively with machine language. Coarse scrolling, on the other hand, can easily be done from BASIC. This program demonstrates coarse horizontal scrolling.

```
10 POKE 648,60:PRINT" {CLR}":POKE648,4:PRINT" {CLR}  
"  
30 A$(1)="THIS IS AN EXAMPLE OF COARSE  
{ 12 {SPACES}}"  
40 A$(2)="TEXT SCROLLING. THE PURPOSE  
{ 13 {SPACES}}"  
50 A$(3)="OF THIS DEMONSTRATION IS TO  
{ 13 {SPACES}}"  
60 A$(4)="SHOW THE READERS HOW THEY MAY USE  
{ 7 {SPACES}}"
```

# Animation

```
70 A$(5) = "TWO SCREENS TO PRODUCE A SCROLLING
   { 6 {SPACES}}"
80 A$(6) = "EFFECT. THIS PROCESS DISPLAYS ONE SCREE
   N"
90 A$(7) = "WHILE PRINTING THE MODIFIED DATA ON THE
   "
92 A$(8) = "OTHER{ 35 {SPACES}}"
95 FOR M=0TO19
100 PRINT "{HOME}{ 6 {DOWN}}";:FOR R=1 TO 8
110 PRINTA$(R)
120 A$(R) = " "+LEFT$(A$(R),LEN(A$(R))-1)
130 NEXT
140 POKE 648,60
145 POKE 53272,(PEEK(53272)AND15)OR16
160 PRINT "{HOME}{ 6 {DOWN}}";:FOR R=1 TO 8
165 PRINTA$(R)
170 A$(R) = " "+LEFT$(A$(R),LEN(A$(R))-1)
180 NEXT
190 POKE 648,4
200 POKE 53272,(PEEK(53272)AND15)OR240
210 NEXT
220 POKE 53272,(PEEK(53272)AND15)OR16
230 PRINT "CLR"
```

It is also quite possible to scroll lines individually. This program sets up vertical columns of clubs and circles and then scrolls two lines in opposite directions across the screen. Notice that the moving lines are only a portion of a longer string. You could create a long message string and scroll it like a banner across a line of the screen.

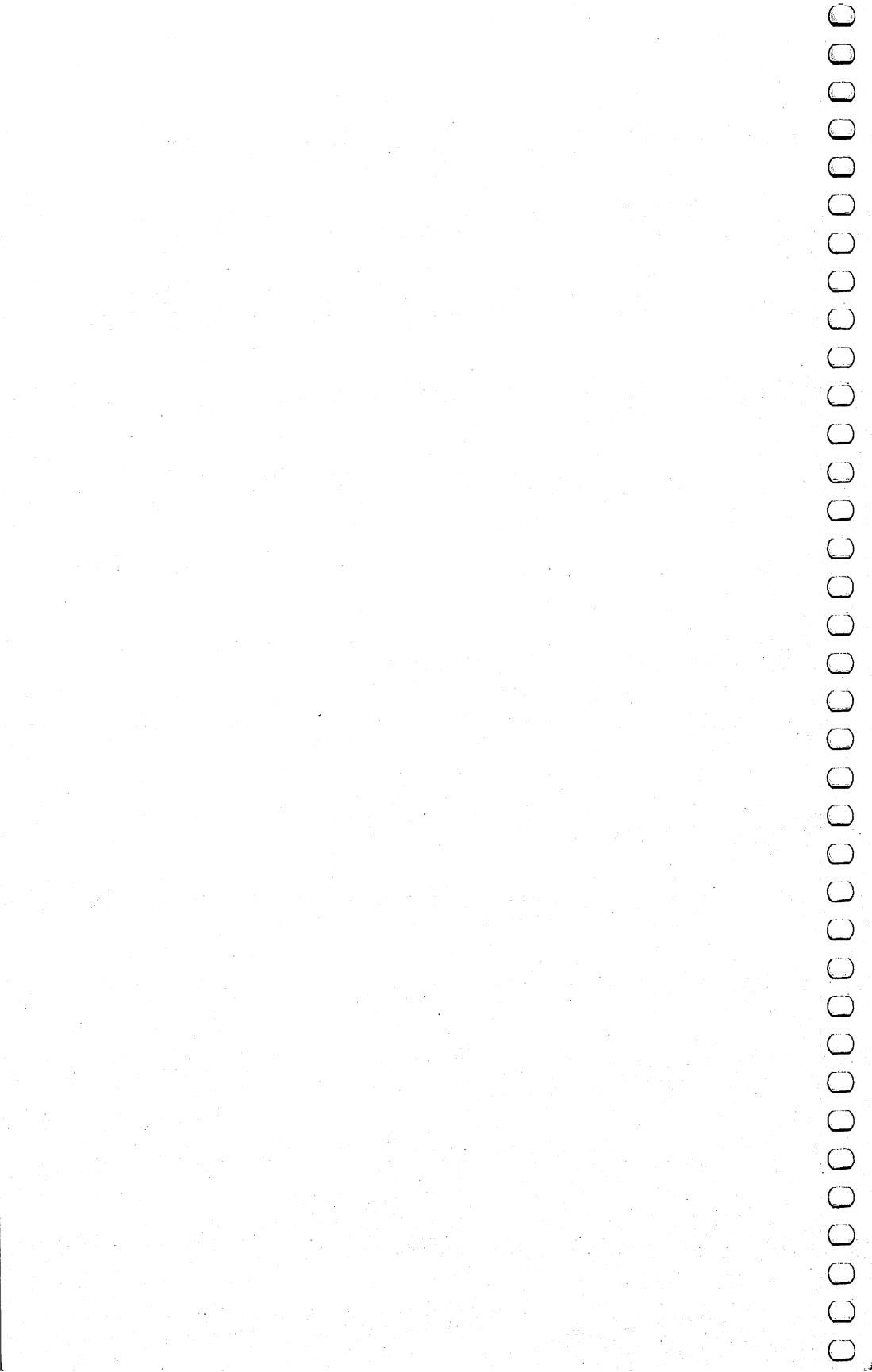
```
10 POKE 53280,13:POKE 53281,5
15 PRINT "[6]{CLR}"
20 B$ = " { 2 {SPACES}}X{ 4 {SPACES}}Q{ 4 {SPACES}}
   X{ 4 {SPACES}}Q{ 2 {SPACES}}"
25 S1$ = "{HOME}{ 9 {DOWN}}":S2$ = "{HOME}
   { 15 {DOWN}}"
30 C$ = B$ + B$ : C$ = C$ + B$
35 PRINT "{HOME}":FOR I=0 TO 22:PRINT LEFT$(C$,40)
   ;:NEXT
40 FOR I=1 TO 10
45 PRINT S1$;MID$(C$,I,40);S2$;MID$(C$,11-I,40);
50 NEXT:GOTO 40
```

# **4**

# **Advanced**

# **Graphics**

# **Appendices**



# Advanced Graphics

## 4

This book contains a number of utility programs that you can use to help you design your own graphics. A great many of the examples in this book were in fact programmed using these very utilities. Here are

some tips on using the utility programs to provide you with an integrated programming/editing library.

### Mixing Modes

The 64 allows you to mix standard and multicolor characters and sprites on the screen. To use this feature effectively, you must design a single set of characters or sprites with some patterns that use multicolor bit-pairs and some that use normal 1-bit dots. To do this, you need to program the multicolor elements and the standard patterns separately.

You can begin with either group. When you have finished editing that set, save it using the *save* command (f2) and end that program.

Then LOAD the other editor and RUN it. When the function screen is displayed, press LOAD (f3) and enter the name you gave the previously saved file. The data will be transferred to the current holding area in memory, and you will be able to edit the characters or sprites that you did not program in the first mode.

### Modifying Existing Files

You can use this same method to modify existing character sets or sprite sets. All you need to do is load the old file using the (f3) command while you are running the editor, and the previous file will be entered into the "holding" area of memory. You will then be able to edit that set just as if you were creating it the first time. This also allows you to save different versions of existing sets. All you need to do is use a different name when you save the modified version.

### Using Character and Sprite Patterns in Programs

The data that you save in the character and sprite files can be used in a number of different ways. The simplest way is to read it

# Advanced Graphics

directly from the disk or cassette file. This can be done from within any BASIC program by OPENing a file with the name of the data set and INPUTting the data.

```
1000 INPUT"START ADDRESS";CT
1010 INPUT"FILE NAME";F$
1020 OPEN 1,8,4,F$+.SPR,R"
1030 INPUT#1,A:BB=ST
1035 POKE CT,A: CT=CT+1
1040 IF BB<>0 THEN 1060
1050 GOTO 1030
1060 CLOSE 1
```

This routine was written to be used with a disk drive. To convert it for use with a Datasette, change line 1020 to

```
1020 OPEN 1,1,0,F$+.SPR"
```

## Converting Character and Sprite Data into DATA Statements

One problem that you may run into is that this process does not allow users without a disk drive or cassette to use your programs. Also, reading the data from disk or tape is quite slow and may become tedious if the process must be repeated each time the program is RUN.

To avoid these problems, you can convert the data on disk or cassette into DATA statements that become a part of your application program. Here is a routine that will read any file you specify and will display formatted and numbered BASIC lines.

```
1000 INPUT"FILE NAME";F$
1010 OPEN 1,8,4,F$+.SPR,R"
1020 X$=STR$(X+1)
1030 L$=RIGHT$(X$,LEN(X$)-1)+" DATA "
1040 INPUT#1,A$:BB=ST:CT=CT+1
1050 IF BB<>0 THEN 1090
1060 L$=L$+LEFT$(A$,LEN(A$)-1)+CHR$(44)
1070 IF LEN(L$)>76 THEN L$=LEFT$(L$,LEN(L$)-1):
    PRINTL$:X=X+1:GOTO1020
1080 GOTO 1040
1090 L$=LEFT$(L$,LEN(L$)-1):PRINTL$
1100 CLOSE 1: PRINT CT
```

As before, if you are using a Datasette, replace the line that OPENS the file with:

```
1010 OPEN 1,1,0,F$+.SPR"
```

If you use this routine for characters instead of sprites, you will need to change the file type from .SPR to .CHR.

# Advanced Graphics

As you run this program, you will need to press RUN/STOP to halt the program whenever the screen fills (every nine program lines). You can then add the displayed DATA lines to the routine by positioning the cursor over the line numbers and pressing RETURN for each line.

After you have entered the first group of lines, you will need to re-RUN the program, this time waiting until the next nine lines have been displayed before pressing RUN/STOP. To enter all of the data, simply repeat this process until all of the data has been converted. The program will automatically stop at the end of your data.

## Characters and Bitmaps Together

The 64 can display sprites and characters on the screen at the same time. It can also display custom characters and sprites at the same time, and if you copy the original character set, it can mix custom and standard characters on the screen. However, it cannot display characters on the bitmapped screen. This is because all of the bits displayed on the screen must be programmed individually, and characters are preprogrammed 8x8 blocks of data.

Of course, you could read each character pattern you want from the character set and then copy it byte for byte into the bitmap cell. But the process would be very time-consuming and would require a lot of programming to accomplish it.

It is possible, though, to create a small high-resolution area on the character screen that would allow you to bitmap right on the character screen. Set aside a group of characters. POKE or PRINT them on the text screen in the same order they have in character memory. Then program that section of patterns in *character memory* exactly as if it were a small bitmap. The drawings you create will show up on the TV screen where these characters are PRINTed.

```
10 POKE 56578, PEEK(56578) OR 3
20 POKE 56576, (PEEK(56576) AND 252) OR 0
30 POKE 53272, (PEEK(53272) AND 240) OR 2
40 POKE 648, 196: FF=0
50 POKE 56334, PEEK(56334) AND 254
60 POKE 1, PEEK(1) AND 251
70 FOR R=53248 TO 54272
80 POKE R-2048, PEEK(R): NEXT
85 FOR R=52224 TO 53247
87 POKE R, 0: NEXT
```

# Advanced Graphics

```
90 POKE 1, PEEK(1)OR 4
100 POKE 56334,PEEK(56334)OR 1
110 PRINT"CLR"
120 SP$=" 24 {RIGHT}"
125 PRINTSP$;" { 10 {@}}"
130 PRINTSP$;"{M}{RVS}@ABCDEFGHI{OFF}{G}"
140 PRINTSP$;"{M}{RVS}JKLMNOPQRS{OFF}{G}"
150 PRINTSP$;"{M}{RVS}TUVWXYZ[{OFF}{G}"
160 PRINTSP$;"{M}{RVS}↑← !";CHR$(34);"#$%&'{OFF}";" {G}"
170 PRINTSP$;"{M}{RVS}()*+,-./01{OFF}{G}{UP}{LEFT}{G}{DOWN}"
180 PRINTSP$;"{M}{RVS}23456789:;{OFF}{G}"
185 PRINTSP$;" { 10 {T}}"
190 X=40:Y=24
195 GOTO 400
200 P=(52224+(INT(Y/8))*80+(INT(X/8))*8+(YAND7))
210 POKEP,PEEK(P)OR2↑(7-(XAND7))
220 GET A$: IF A$="" THEN 220
230 IF A$="{DOWN}"THEN 300:REM DOWN
240 IF A$="{UP}"THEN 320:REM UP
250 IF A$="{RIGHT}"THEN 340:REM RIGHT
260 IF A$="{LEFT}"THEN 360:REM LEFT
265 IF A$="{F-1}"THEN 700:REM LEFT
270 GOTO 220
300 Y=Y+1:GOTO 200
320 Y=Y-1:GOTO 200
340 X=X+1:GOTO 200
360 X=X-1:GOTO 200
400 PRINT"HOME}{ 2 {DOWN}}THIS IS AN EXAMPLE OF
A"
410 PRINT"TEXT SCREEN MIXED WITH A"
420 PRINT"HIGH-RESOLUTION GRAPHICS"
430 PRINT"AREA. THE GRAPHICS AREA "
440 PRINT"CAN BE ENLARGED FOR BIG-"
450 PRINT"GER GRAPHICS NEEDS."
460 PRINT" { 3 {DOWN}}TO MOVE THE CURSOR, PRESS TH
E CURSOR"
470 PRINT"PRESS THE CURSOR CONTROL KEYS JUST AS"
480 PRINT"WOULD IF YOU WERE MOVING THE TEXT CUR-"
490 PRINT"SOR."
500 PRINT"BE CAREFUL NOT TO MOVE THE CURSOR
 { 5 {SPACES}}"
510 PRINT"BEYOND THE UPPER OR LOWER BOUNDRIES
 { 3 {SPACES}}"
520 PRINT"BECAUSE THIS EXAMPLE PROGRAM DOES NOT "
530 PRINT"PROTECT YOU FROM OVER-WRITING THE STAN-
"
540 PRINT"DARD CHARACTERS!"
```

# Advanced Graphics

```
560 PRINT "{DOWN} TO CLEAR THE GRAPHICS AND RE-POSITION"
570 PRINT "THE GRAPHICS CURSOR IN THE CENTER OF
      { 2 {SPACES}}"
580 PRINT "THE GRAPHICS SCREEN, PRESS [F1]. "
      { 6 {SPACES}}"
600 GOTO 200
700 FOR R=52224 TO 53247
710 POKE R,0: NEXT
720 GOTO 190
```

The example above only allows you to move the high-resolution graphics line. If you want to enter text as well, you can use the GET command and print characters in the area that the instructions now occupy. Of course, the graphics area can be positioned anywhere you like.

Another application for the tiny graphics area is in designing repeating patterns. Since the graphics are programmed into a block of custom characters, your program need only print that block repeatedly, and you will repeat the pattern that same number of times.

```
10 POKE 56578,PEEK(56578) OR 3
20 POKE 56576,(PEEK(56576) AND 252)OR 0
30 POKE 53272,(PEEK(53272) AND 240)OR 2
40 POKE 648,196: FF=0
50 POKE 56334,PEEK(56334) AND 254
60 POKE 1, PEEK(1) AND 251
70 FOR R=53248 TO 54272
80 POKE R-2048, PEEK(R): NEXT
85 FOR R=52224 TO 53247
87 POKE R,0: NEXT
90 POKE 1, PEEK(1) OR 4
100 POKE 56334,PEEK(56334) OR 1
110 PRINT "[CLR]"
130 A$ = "{RVS}#${%&'}()*+,{ 10 {LEFT}}{DOWN}"
140 A$ = A$ + "-./0123456{ 10 {LEFT}}{DOWN}"
150 A$ = A$ + "789:;<=>?*{ 10 {LEFT}}{DOWN}"
160 A$ = A$ + "ABCDEFGHIJ{ 10 {LEFT}}{DOWN}"
170 A$ = A$ + "KLMNOPQRST{ 10 {LEFT}}{DOWN}"
180 A$ = A$ + "UVWXYZ+{-}-{{↑}}"
182 PRINT "[HOME]"; :FOR R=0TO3
183 PRINT A$;"{ 5 {UP}}"; :NEXT:PRINT "[HOME]"
      { 6 {DOWN}}";
184 FOR R=0TO3
185 PRINT A$;"{ 5 {UP}}"; :NEXT:PRINT "[HOME]"
      { 12 {DOWN}}";
186 FOR R=0TO3
```

# Advanced Graphics

```
187 PRINTA$;"{ 5 {UP}}";:NEXT:PRINT" {HOME}
    { 18 {DOWN}}";
188 FOR R=0TO3
189 PRINTA$;"{ 5 {UP}}";:NEXT:PRINT" {HOME}
    { 24 {DOWN}}";
190 X=40:Y=24
200 P=(52504+((INT(Y/8))*80+(INT(X/8))*8+(YAND7)))
210 POKEP,PEEK(P)OR2↑(7-(XAND7))
220 GET A$: IF A$="" THEN 220
230 IF A$="{DOWN}"THEN 300:REM DOWN
240 IF A$="{UP}"THEN 320:REM UP
250 IF A$="{RIGHT}"THEN 340:REM RIGHT
260 IF A$="{LEFT}"THEN 360:REM LEFT
265 IF A$="{F-1}"THEN 700
270 GOTO 220
300 Y=Y+1:GOTO 200
320 Y=Y-1:GOTO 200
340 X=X+1:GOTO 200
360 X=X-1:GOTO 200
400 PRINT" {HOME}{ 2 {DOWN}}THIS IS AN EXAMPLE OF
    A"
410 PRINT"TEXT SCREEN MIXED WITH A"
420 PRINT"HIGH-RESOLUTION GRAPHICS"
430 PRINT"AREA. THE GRAPHICS AREA "
440 PRINT"CAN BE ENLARGED FOR BIG-"
450 PRINT"GER GRAPHICS NEEDS."
460 PRINT"{ 3 {DOWN}}TO MOVE THE CURSOR, PRESS TH
    E CURSOR"
470 PRINT"PRESS THE CURSOR CONTROL KEYS JUST AS"
480 PRINT"WOULD IF YOU WERE MOVING THE TEXT CUR-"
490 PRINT"SOR."
500 PRINT"BE CAREFUL NOT TO MOVE THE CURSOR
    { 5 {SPACES}}"
510 PRINT"BEYOND THE UPPER OR LOWER BOUNDRIES
    { 3 {SPACES}}"
520 PRINT"BECAUSE THIS EXAMPLE PROGRAM DOES NOT "
530 PRINT"PROTECT YOU FROM OVER-WRITING THE STAN-
    "
540 PRINT"DARD CHARACTERS!"
560 PRINT" {DOWN}TO CLEAR THE GRAPHICS AND RE-POSI
    TION"
570 PRINT"THE GRAPHICS CURSOR IN THE CENTER OF
    { 2 {SPACES}}"
580 PRINT"THE GRAPHICS SCREEN, PRESS [F1].
    { 6 {SPACES}}"
600 GOTO 200
700 FOR R=52224 TO 53247
710 POKE R,0: NEXT
720 GOTO 190
```

# Advanced Graphics

## Machine Language Routines

The more advanced your programs become, the more operations that will need to be carried out while the programs are running. The problem this poses is that BASIC is just too slow to move the data around in memory as quickly as you need. This book has not been written to teach you machine language programming, but the routines that follow will perform some simple functions that can help to speed up your BASIC programs.

### Moving Blocks of Data

This routine is a combination of BASIC and machine language and asks you three questions when it is RUN.

- 1) START MEM LOC TO READ?
- 2) START MEM LOC TO XFER?
- 3) NO. BYTES TO XFER?

The program will then copy the data from the first location to the second location using machine language.

```
10 REM --- MACHINE LANGUAGE DATA ---
15 REM
30 DATA 160,0,177,251,145,253,230,251,196,251,208
    ,2,230,252,230,253,196,253,208
40 DATA 2,230,254,206,252,3,204,252,3,208,8,204,2
    53,3,240,6,206,253,3,76,62,3,96
43 REM
44 REM --- POKE MACHINE CODE ROUTINE --
45 REM --- INTO CASSETTE BUFFER -----
46 REM
50 FOR R=828 TO 869
60 READ A: POKE R,A: NEXT
70 REM
80 REM ---- INPUT PARAMETERS -----
90 REM
100 INPUT"START MEM LOC TO READ";N(0)
110 INPUT"START MEM LOC TO XFER";N(1)
120 INPUT"NO. BYTES TO XFER";N(2)
125 REM FOR R=N(0) TO N(0)+N(2):POKE R,INT(RND(0
    )*255):NEXT
130 REM
140 REM --- HEX CONVERSION ROUTINE ---
150 REM
160 FOR R=0 TO 2
170 N=N(R)
175 H=INT(N/256): L=N-(H*256)
180 M(R)=L:M(R+3)=H
190 NEXT
```

# Advanced Graphics

```
195 REM
197 REM POKE IN MACHINE CODE VARIABLES
199 REM
200 POKE251,M(0):POKE252,M(3):POKE253,M(1):POKE25
    4,M(4):POKE1020,M(2):POKE1021,M(5)
207 REM
208 REM ---- RUN MACHINE CODE XFER ----
209 REM
210 SYS 828
```

Here is an assembly listing of the machine language program.

HEX ADDRESS	OP CODE	LO	HI	MNEMONIC, INSTRUCTION
033C	A0	00		LDY #\$00
033E	B1	FB		LDA (\$FB),Y
0340	91	FD		STA (\$FD),Y
0342	E6	FB		INC \$FB
0344	C4	FB		CPY \$FB
0346	D0	02		BNE \$034A
0348	E6	FC		INC \$FC
034A	E6	FD		INC \$FD
034C	C4	FD		CPY \$FD
034E	D0	02		BNE \$0352
0350	E6	FE		INC \$FE
0352	CE	FC	03	DEC \$03FC
0355	CC	FC	03	CPY \$03FC
0358	D0	08		BNE \$035F
035A	CC	FD	03	CPY \$03FD
035D	F0	06		BEQ \$0365
035F	CE	FD	03	DEC \$03FD
0362	4C	3E	03	JMP \$033E
0365	60			RTS

To use this block transfer in a real application, you will need to POKE the necessary values before switching to machine language and then SYS to the program. Here are the necessary lines to add or modify for the Character Editor from Chapter 1.

```
3 DATA 160,0,177,251,145,253,230,251,196,251,208,
    2,230,252,230,253,196,253,208
4 DATA 2,230,254,206,252,3,204,252,3,208,8,204,25
    3,3,240,6,206,253,3,76,62,3,96
5 FOR R=828 TO 869: READ A: POKE R,A: NEXT
120 POKE 251,255:POKE252,207:POKE253,255:POKE254,
    199:POKE1020,0:POKE1021,8
130 SYS 832
```

# Advanced Graphics

## Collision Detection

Keeping track of the sprites' positions is more than enough work to keep your BASIC program so busy that it won't have time to do much else. This routine looks at the sprite-to-character collision registers and makes a moving sprite change direction. (The sprite-to-character collision register is memory location 53279.)

```
10 PRINT"CLR}{ 7 {DOWN}}X{ 14 {RIGHT}}X":F=-1
20 POKE 53269,1:POKE 2040,50
30 FOR R=3200 TO 3263
40 POKE R, 255
50 NEXT
70 POKE 53249,100: R=70:POKE 53248, R
80 R=R+F
100 POKE 53264,ABS(256<R):POKE53248,RAND255
105 A=PEEK(53279):IF(AAND1=1)THEN 170
110 GOTO 80
115 FOR R=0 TO 500:NEXT
120 FOR R=350 TO 0 STEP -1
130 POKE 53264,ABS(256<R):POKE53248,RAND255
140 NEXT
150 FOR R=0 TO 500:NEXT
160 GOTO 80
170 F=F*-1:R=R+(F*4):POKE53264,ABS(256<R):POKE532
48,RAND255:K= PEEK(53279):GOTO 80
```

Sprites can also detect collisions with other sprites; this routine works just as the previous one did, but bounces off sprites. (The sprite-to-sprite collision register is memory location 53278.)

```
10 F=-1:PRINT"CLR"
20 POKE 53269,7:POKE 2040,50:POKE 2041,50:POKE204
2, 50
30 FOR R=3200 TO 3263
40 POKE R, 255
50 NEXT
70 POKE 53249,100: R=70:POKE 53248, R
72 POKE 53251,100:POKE 53250, 30
73 POKE 53253,100:POKE 53252, 130
80 R=R+F
100 POKE 53264,ABS(256<R):POKE53248,RAND255
105 A=PEEK(53278):IF(AAND1=1)THEN 170
110 GOTO 80
115 FOR R=0 TO 500:NEXT
120 FOR R=350 TO 0 STEP -1
130 POKE 53264,ABS(256<R):POKE53248,RAND255
140 NEXT
150 FOR R=0 TO 500:NEXT
```

# Advanced Graphics

```
160 GOTO 80
170 F=F*-1:R=R+(F*4):POKE53264,ABS(256<R):POKE532
     48,RAND255:K= PEEK(53278):GOTO80
```

## Smooth Scrolling

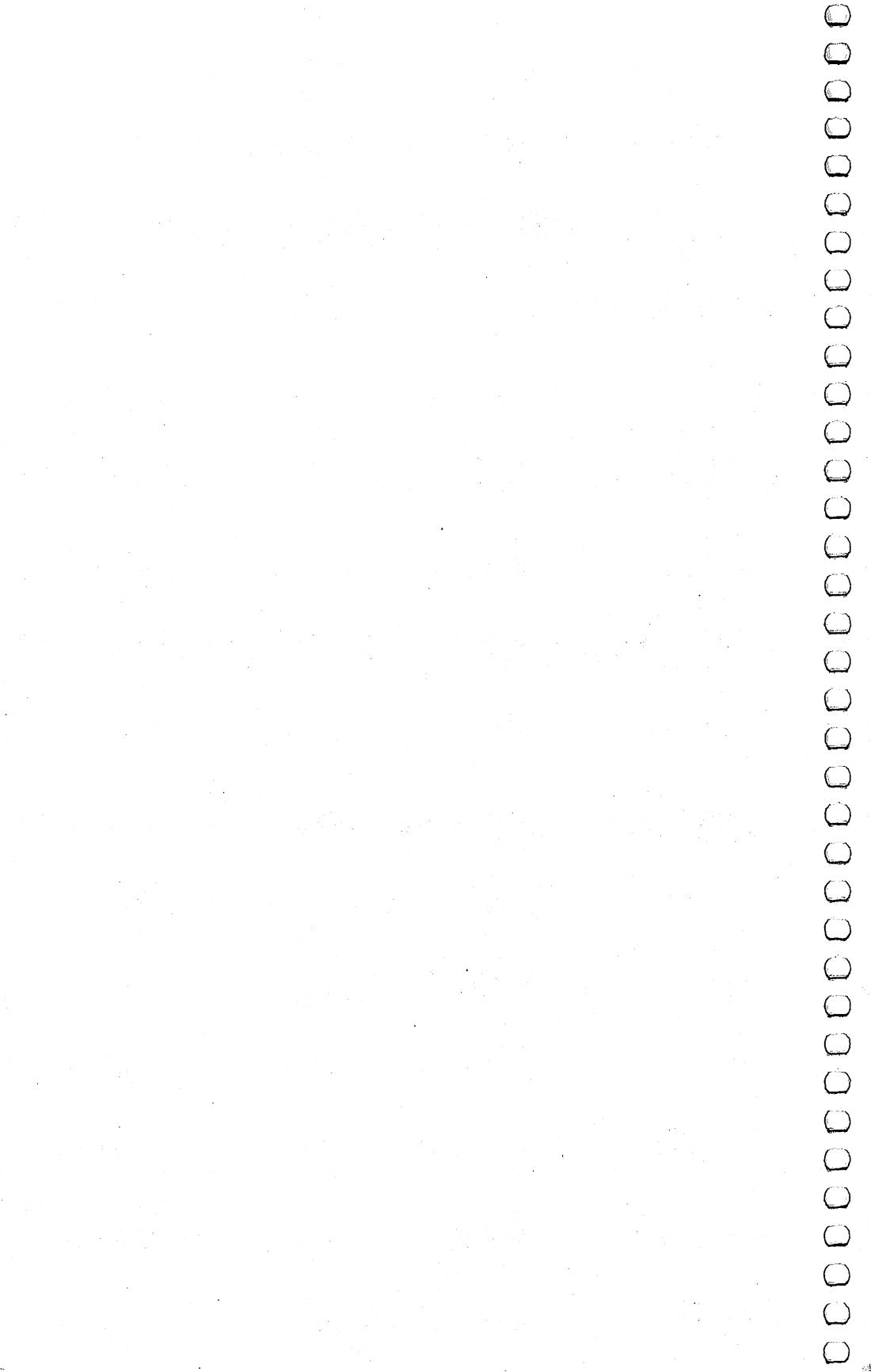
The previous chapter had a routine that provided coarse scrolling for text. The following program is essentially the same, but includes a short machine language routine that scrolls the text smoothly across the screen.

This is performed by incrementing the scroll register 53270.

```
1 DATA 169,60,141,136,2,169,21,141,24,208,169,200
     ,141,22,208,96
2 DATA 169,4,141,136,2,169,245,141,24,208,169,200
     ,141,22,208,96
3 FOR R=8000 TO 8031: READ A:POKE R,A: NEXT
10 POKE 648,4:PRINT"{CLR}":POKE648,60:PRINT"{CLR}"
"
30 A$(1)="THIS IS AN EXAMPLE OF SMOOTH
{ 12 {SPACES}}"
40 A$(2)="TEXT SCROLLING. BECAUSE OF THE DELAY
{ 4 {SPACES}}"
50 A$(3)="INVOLVED WITH SWITCHING FROM THE BEGIN-
"
60 A$(4)="NING TO THE END OF THE SCROLL SEQUENCE
{ 2 {SPACES}}"
70 A$(5)="A SHORT MACHINE LANGUAGE ROUTINE HAS
{ 4 {SPACES}}"
80 A$(6)="BEEN ADDED. IT HAS TWO ENTRY POINTS, ON
E"
90 A$(7)="IS AT LOCATION 8000 AND THE OTHER IS AT
"
92 A$(8)="LOCATION 8016.{ 26 {SPACES}}"
95 FOR M=0TO20
100 PRINT"{HOME}{ 6 {DOWN}}";:FOR R=1 TO 8
110 PRINTA$(R)
120 A$(R)=" "+LEFT$(A$(R),LEN(A$(R))-1)
125 POKE53270,(PEEK(53270)AND248)+(R-1)
130 NEXT
140 SYS 8000
160 PRINT"{HOME}{ 6 {DOWN}}";:FOR R=1 TO 8
165 PRINTA$(R)
170 A$(R)=" "+LEFT$(A$(R),LEN(A$(R))-1)
175 POKE53270,(PEEK(53270)AND248)+(R-1)
180 NEXT
190 SYS 8016
210 NEXT
220 POKE 53272,21
230 PRINT"{CLR}"
```

---

# Appendices



## Appendix A

# A Beginner's Guide to Typing In Programs

### What Is a Program?

A computer cannot perform any task by itself. Like a car without gas, a computer has *potential*, but without a program, it isn't going anywhere. Most of the programs published in this book are written in a computer language called BASIC. BASIC is easy to learn and is built into all Commodore 64s.

### BASIC Programs

Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, or number is significant. A common mistake is substituting a letter such as O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, you must enter all punctuation such as colons and commas just as they appear in the book. Spacing can be important. To be safe, type in the listings *exactly* as they appear.

### Braces and Special Characters

The exception to this typing rule is when you see the braces, such as {DOWN}. Anything within a set of braces is a special character or characters that cannot easily be listed on a printer. When you come across such a special statement, refer to Appendix B "How to Type In Programs."

### About DATA Statements

Some programs contain a section or sections of DATA statements. These lines provide information needed by the program. Some DATA statements contain actual programs (called machine language); others contain graphics codes. These lines are especially sensitive to errors.

If a single number in any one DATA statement is mistyped, your machine could lock up, or crash. The keyboard and STOP key may seem dead, and the screen may go blank. Don't panic — no damage is done. To regain control, you have to turn off your

# Appendix A

computer, then turn it back on. This will erase whatever program was in memory, *so always SAVE a copy of your program before you RUN it.* If your computer crashes, you can LOAD the program and look for your mistake.

Sometimes a mistyped DATA statement will cause an error message when the program is RUN. The error message may refer to the program line that READS the data. *The error is still in the DATA statements, though.*

## Get to Know Your Machine

You should familiarize yourself with your computer before attempting to type in a program. Learn the statements you use to store and retrieve programs from tape or disk. You'll want to save a copy of your program, so that you won't have to type it in every time you want to use it. Learn to use your machine's editing functions. How do you change a line if you made a mistake? You can always retype the line, but you at least need to know how to backspace. Do you know how to enter reverse video, lowercase, and control characters? It's all explained in your computer's manuals.

## A Quick Review

- 1) Type in the program a line at a time, in order. Press RETURN at the end of each line. Use backspace or the back arrow to correct mistakes.
- 2) Check the line you've typed against the line in the book. You can check the entire program again if you get an error when you RUN the program.

# How to Type In Programs

To make it easy to know exactly what to type when entering one of these programs into your computer, we have established the following listing conventions.

Generally, Commodore 64 program listings will contain words in braces which spell out any special characters: {DOWN} would mean to press the cursor down key. {5 {SPACES}} would mean to press the space bar five times.

To indicate that a key should be *shifted* (hold down the SHIFT key while pressing the other key), the key would be underlined in our listings. For example, S would mean to type the S key while holding the shift key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (e.g., {10 N}), you should type the key as many times as indicated (in our example, you would enter ten shifted N's).

If a key is enclosed in special brackets, { }, you should hold down the *Commodore key* while pressing the key inside the special brackets. (The Commodore key is the key in the lower-left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as necessary.

Rarely, you'll see a solitary letter of the alphabet enclosed in braces. These characters can be entered by holding down the CTRL key while typing the letter in the braces. For example, {A} would indicate that you should press CTRL-A.

About the *quote mode*: You know that you can move the cursor around the screen with the CRSR keys. Sometimes a programmer will want to move the cursor under program control. That's why you see all the {LEFT}'s, {HOME}'s, and {BLU}'s in our programs. The only way the computer can tell the difference between direct and programmed cursor control is the quote mode.

Once you press the quote (the double quote, SHIFT-2), you are in the quote mode. If you type something and then try to change it by moving the cursor left, you'll only get a bunch of reverse-video lines. These are the symbols for cursor left. The

# Appendix B

only editing key that isn't programmable is the DEL key; you can still use DEL to back up and edit the line. Once you type another quote, you are out of quote mode.

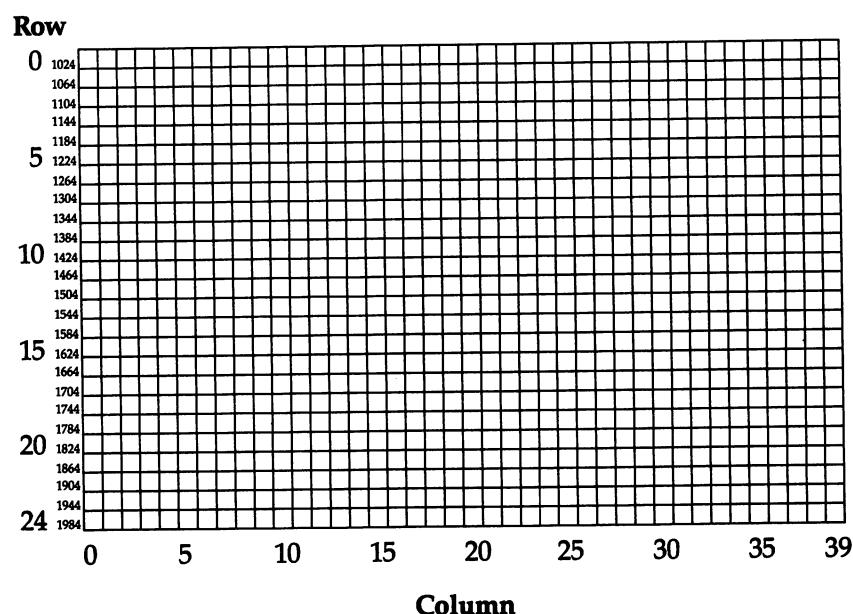
You also go into quote mode when you INSeRT spaces into a line. In any case, the easiest way to get out of quote mode is to just press RETURN. You'll then be out of quote mode and you can cursor up to the mistyped line and fix it.

Use the following table when entering cursor and color control keys:

When You Read:	Press:	See:	When You Read:	Press:	See:
{CLR}	SHIFT	CLR/HOME	{ORANGE}	COMMODORE	f1
{HOME}		CLR/HOME	{BROWN}	COMMODORE	f2
{UP}	SHIFT	CRSR	{LT/RED}	COMMODORE	f3
{DOWN}		CRSR	{DK/GREY}	COMMODORE	f4
{LEFT}	SHIFT	CRSR ←	{MED/GREY}	COMMODORE	f5
{RIGHT}		CRSR →	{LT/GREEN}	COMMODORE	f6
{RVS}	CTRL	9	{LT/BLUE}	COMMODORE	f7
{OFF}	CTRL	0	{LT/GREY}	COMMODORE	f8
{BLACK}	CTRL	1	{F-1}		f1
{WHITE}	CTRL	2	{F-2}		f2
{RED}	CTRL	3	{F-3}		f3
{CYAN}	CTRL	4	{F-4}		f4
{PURPLE}	CTRL	5	{F-5}		f5
{GREEN}	CTRL	6	{F-6}		f6
{BLUE}	CTRL	7	{F-7}		f7
{YELLOW}	CTRL	8	{F-8}		f8
			£		£

## **Appendix C**

# Screen Location Table



## **Appendix D**

# **Screen Color Memory Table**

**Row**

0	55296	55336	55376	55416	55456	55496	55536	55576	55616	55656	55696	55736	55776	55816	55856	55896	55936	55976	56016	56056	56096	56136	56176	56216	56256
5																									
10																									
15																									
20																									
24																									

**Column**

## Appendix E

# Screen Color Codes

### Value To POKE For Each Color

Color	Low nybble color value	High nybble color value	Select multicolor color value
Black	0	0	8
White	1	16	9
Red	2	32	10
Cyan	3	48	11
Purple	4	64	12
Green	5	80	13
Blue	6	96	14
Yellow	7	112	15
Orange	8	128	-
Brown	9	144	-
Light Red	10	160	-
Dark Grey	11	176	-
Medium Grey	12	192	-
Light Green	13	208	-
Light Blue	14	224	-
Light Grey	15	240	-

### Where To POKE Color Values For Each Mode

Mode*	Bit or bit-pair	Location	Color value
Regular text	0	53281	Low nybble
	1	Color memory	Low nybble
Multicolor text	00	53281	Low nybble
	01	53282	Low nybble
Extended color text #	10	53283	Low nybble
	11	Color memory	Select multicolor
Extended color text #	00	53281	Low nybble
	01	53282	Low nybble
Bitmapped	10	53283	Low nybble
	11	53284	Low nybble
Bitmapped	0	Screen memory	Low nybble ±
	1	Screen memory	High nybble ±
Multicolor bitmapped	00	53281	Low nybble
	01	Screen memory	High nybble ±
Multicolor bitmapped	10	Screen memory	Low nybble ±
	11	Color memory	Low nybble

\* For all modes, the screen border color is controlled by POKEing location 53280 with the low nybble color value.

# In extended color mode, bits 6 and 7 of each byte of screen memory serve as the bit-pair controlling background color. Because only bits 0-5 are available for character selection, only characters with screen codes 0-63 can be used in this mode.

+ In the bitmapped modes, the high and low nybble color values are ORed together and POKEd into the same location in screen memory to control the colors of the corresponding cell in the bitmap. For example, to control the colors of cell 0 of the bitmap, OR the high and low nybble values and POKE the result into location 0 of screen memory.

## Appendix F

# ASCII Codes

ASCII	CHARACTER	ASCII	CHARACTER
5	WHITE	50	2
8	DISABLE	51	3
	SHIFT COMMODORE	52	4
9	ENABLE	53	5
	SHIFT COMMODORE	54	6
13	RETURN	55	7
14	LOWERCASE	56	8
17	CURSOR DOWN	57	9
18	REVERSE VIDEO	58	:
19	HOME	59	;
20	DELETE	60	<
28	RED	61	=
29	CURSOR RIGHT	62	>
30	GREEN	63	?
31	BLUE	64	@
32	SPACE	65	A
33	!	66	B
34	"	67	C
35	#	68	D
36	\$	69	E
37	%	70	F
38	&	71	G
39	,	72	H
40	(	73	I
41	)	74	J
42	*	75	K
43	+	76	L
44	,	77	M
45	-	78	N
46	.	79	O
47	/	80	P
48	0	81	Q
49	1	82	R

## Appendix F

ASCII	CHARACTER	ASCII	CHARACTER
83	S	120	♣
84	T	121	♦
85	U	122	♥
86	V	123	♠
87	W	124	π
88	X	125	■
89	Y	126	ORANGE
90	Z	127	f1
91	—	129	f3
92	£	133	f5
93	—	134	f7
94	→	135	f2
95	↑	136	f4
96	█	137	f6
97	█	138	f8
98	█	140	SHIFTED RETURN
99	█	141	UPPERCASE
100	█	142	BLACK
101	█	144	CURSOR UP
102	█	145	REVERSE VIDEO OFF
103	█	146	CLEAR SCREEN
104	█	147	INSERT
105	█	148	BROWN
106	█	149	LIGHT RED
107	█	150	GRAY 1
108	█	151	GRAY 2
109	█	152	LIGHT GREEN
110	█	153	LIGHT BLUE
111	█	154	GRAY 3
112	█	155	PURPLE
113	█	156	CURSOR LEFT
114	█	157	YELLOW
115	█	158	CYAN
116	█	159	SPACE
117	█	160	█
118	█	161	█
119	█		

## **Appendix F**

ASCII	CHARACTER	ASCII	CHARACTER
162	█	200	██
163	□	201	███
164	□	202	███
165	□	203	███
166	█	204	███
167	□	205	███
168	█	206	███
169	█	207	███
170	█	208	███
171	█	209	███
172	█	210	███
173	█	211	███
174	█	212	███
175	█	213	███
176	█	214	███
177	█	215	███
178	█	216	███
179	█	217	███
180	█	218	███
181	█	219	███
182	█	220	███
183	█	221	███
184	█	222	███
185	█	223	███
186	█	224	███
187	█	225	███
188	█	226	███
189	█	227	███
190	█	228	███
191	█	229	███
192	█	230	███
193	█	231	███
194	█	232	███
195	█	233	███
196	█	234	███
197	█	235	███
198	█	236	███
199	█	237	███
		SPACE	

## **Appendix F**

<b>ASCII</b>	<b>CHARACTER</b>
238	□
239	□
240	□
241	□
242	□
243	□
244	□
245	□
246	□
247	□
248	□
249	□
250	□
251	□
252	□
253	□
254	□
255	π

0-4, 6, 7, 10-12, 15, 16, 21-27, 128,  
130-132, and 143 are not used.

# Appendix G

## Screen Codes

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
0	@	@	31	←	←
1	A	a	32	-space-	!
2	B	b	33	"	"
3	C	c	34	#	#
4	D	d	35	\$	\$
5	E	e	36	%	%
6	F	f	37	&	&
7	G	g	38	,	,
8	H	h	39	(	(
9	I	i	40	)	)
10	J	j	41	*	*
11	K	k	42	+	+
12	L	l	43	,	,
13	M	m	44	-	-
14	N	n	45	.	.
15	O	o	46	/	/
16	P	p	47	0	0
17	Q	q	48	1	1
18	R	r	49	2	2
19	S	s	50	3	3
20	T	t	51	4	4
21	U	u	52	5	5
22	V	v	53	6	6
23	W	w	54	7	7
24	X	x	55	8	8
25	Y	y	56	9	9
26	Z	z	57	:	:
27	[	[	58	;	;
28	]	]	59	<	<
29	↑	↑	60	=	=
30			61		

# Appendix G

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
62	>		99	□	
63	?		100	□	
64	▀		101	□	
65	♠	A	102	□	
66	▀	B	103	□	
67	▀	C	104	□	
68	▀	D	105	□	
69	▀	E	106	□	
70	▀	F	107	□	
71	▀	G	108	□	
72	▀	H	109	□	
73	▀	I	110	□	
74	▀	J	111	□	
75	▀	K	112	□	
76	▀	L	113	□	
77	▀	M	114	□	
78	▀	N	115	□	
79	▀	O	116	□	
80	▀	P	117	□	
81	▀	Q	118	□	
82	▀	R	119	□	
83	▀	S	120	□	
84	▀	T	121	□	
85	▀	U	122	□	
86	▀	V	123	□	
87	▀	W	124	□	
88	▀	X	125	□	
89	▀	Y	126	□	
90	▀	Z	127	▀	
91					
92					
93					
94					
95					
96					
97					
98					
		-space-			
		▀			
		▀			

## Appendix H

# Commodore 64 Keycodes

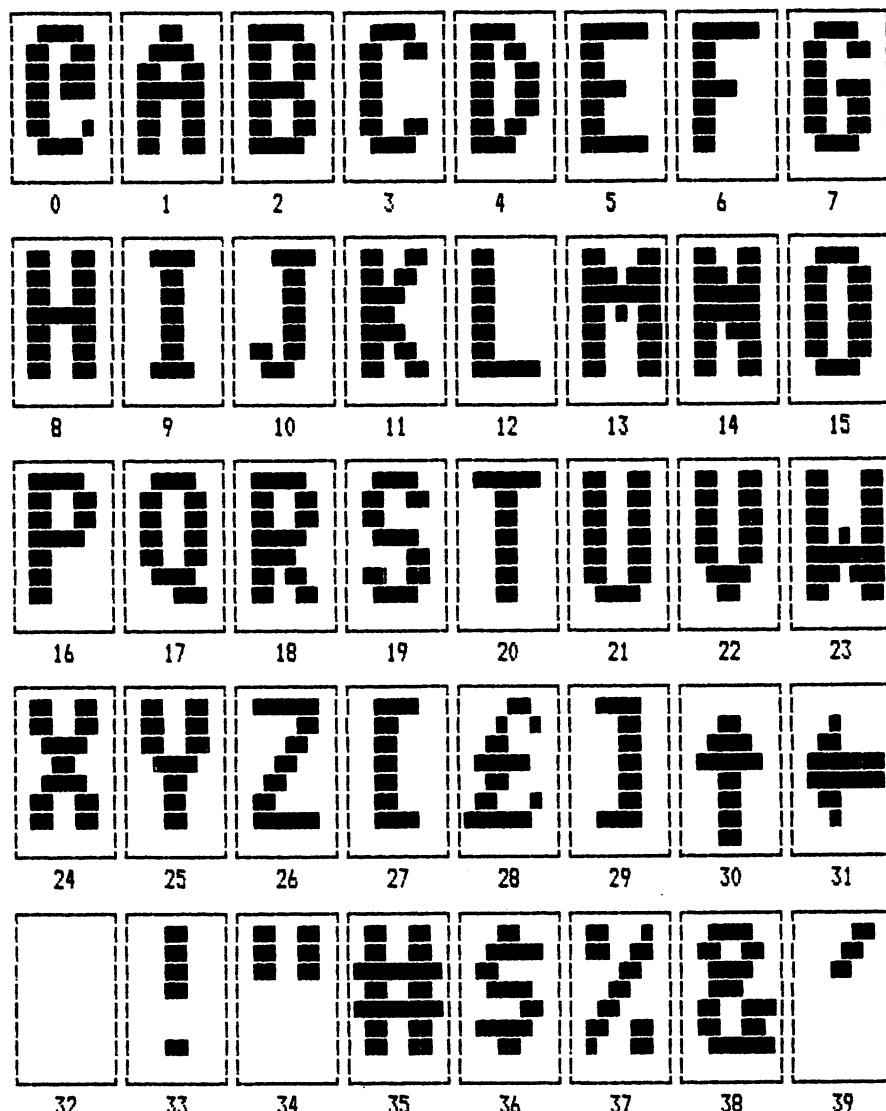
Key	Keycode	Key	Keycode
A	10	6	19
B	28	7	24
C	20	8	27
D	18	9	32
E	14	0	35
F	21	+	40
G	26	-	43
H	29	£	48
I	33	CLR/HOME	51
J	34	INST/DEL	0
K	37	←	57
L	42	@	46
M	36	*	49
N	39	↑	54
O	38	:	45
P	41	;	50
Q	62	=	53
R	17	RETURN	1
S	13	,	47
T	22	.	44
U	30	/	55
V	31	CRSR↑↓	7
W	9	CRSR↔	2
X	23	f1	4
Y	25	f3	5
Z	12	f5	6
1	56	f7	3
2	59	SPACE	60
3	8	RUN/STOP	63
4	11	NO KEY	
5	16	PRESSED	64

The keycode is the number found at location 197 for the current key being pressed. Try this one-line program:

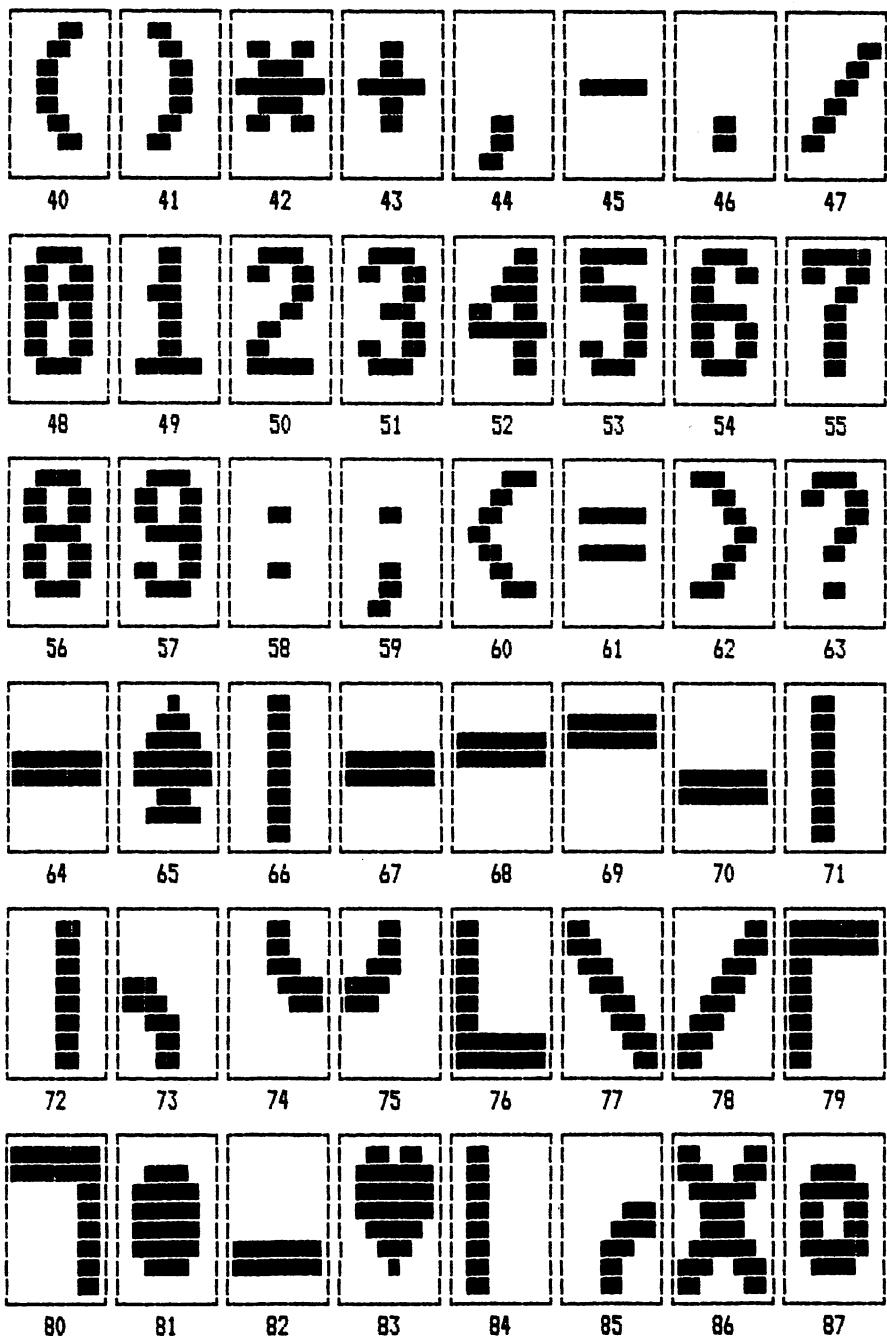
**10 PRINT PEEK (197): GOTO 10**

## Appendix I

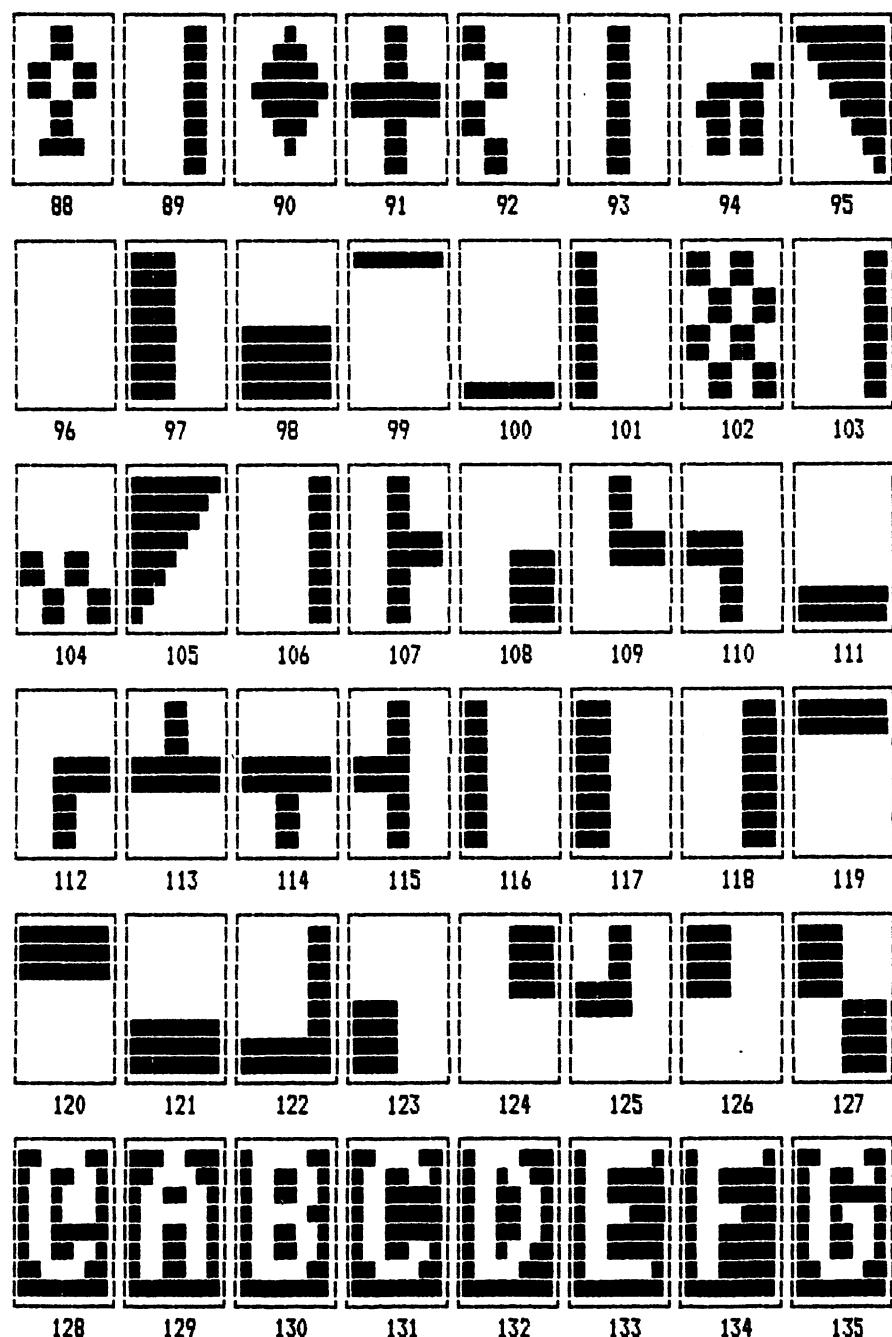
# Character Patterns and Screen Codes, Set 1



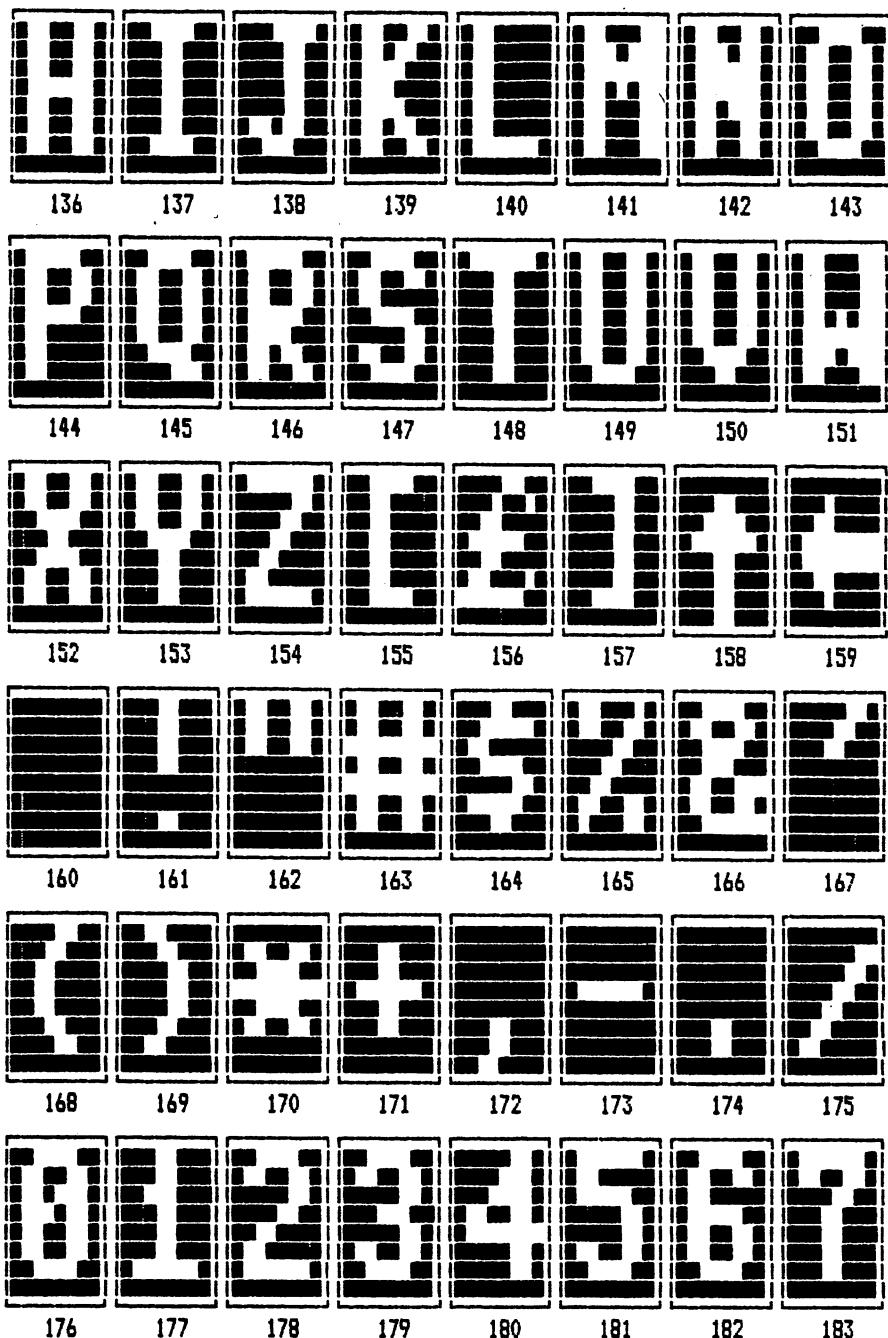
## Appendix I



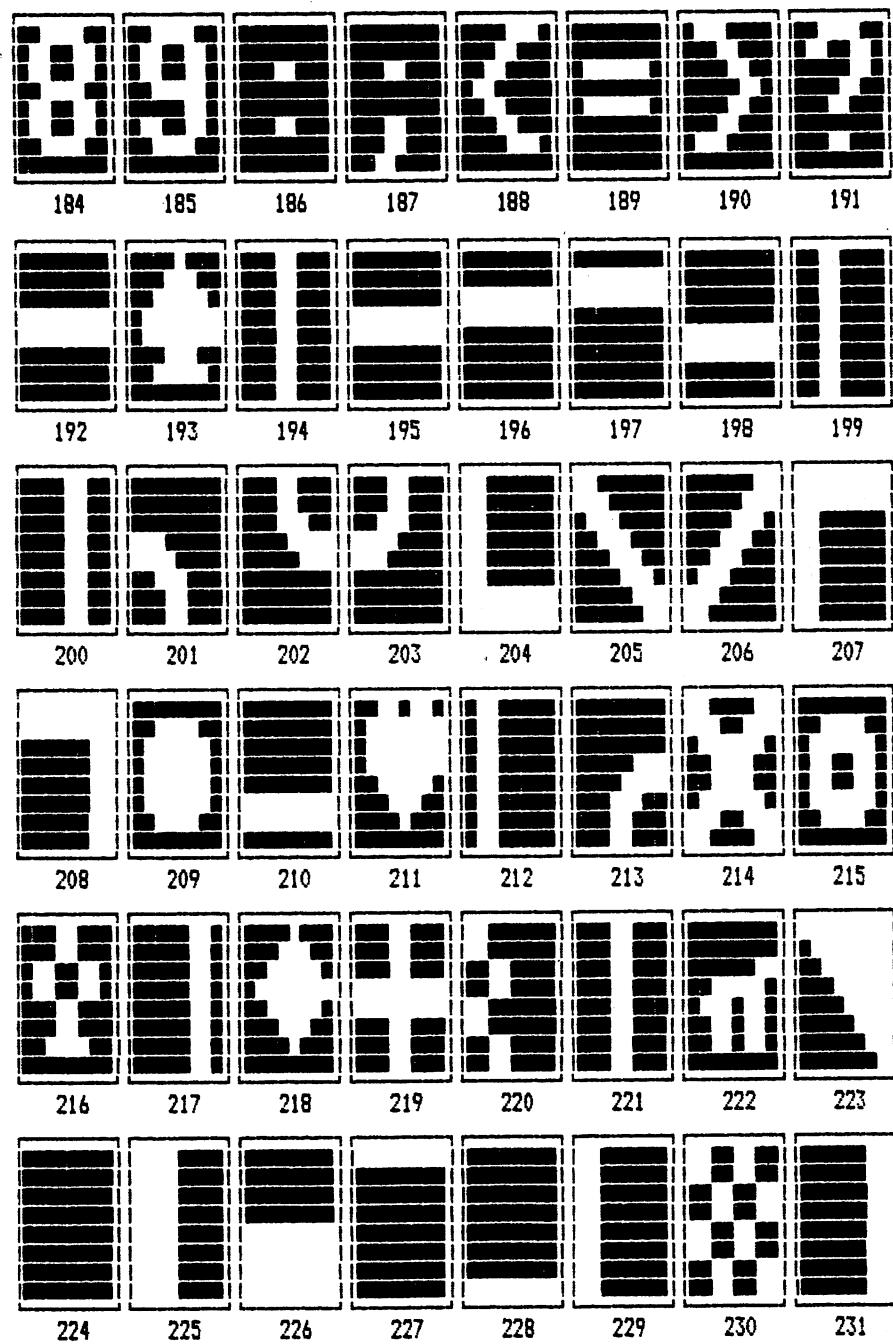
# Appendix I



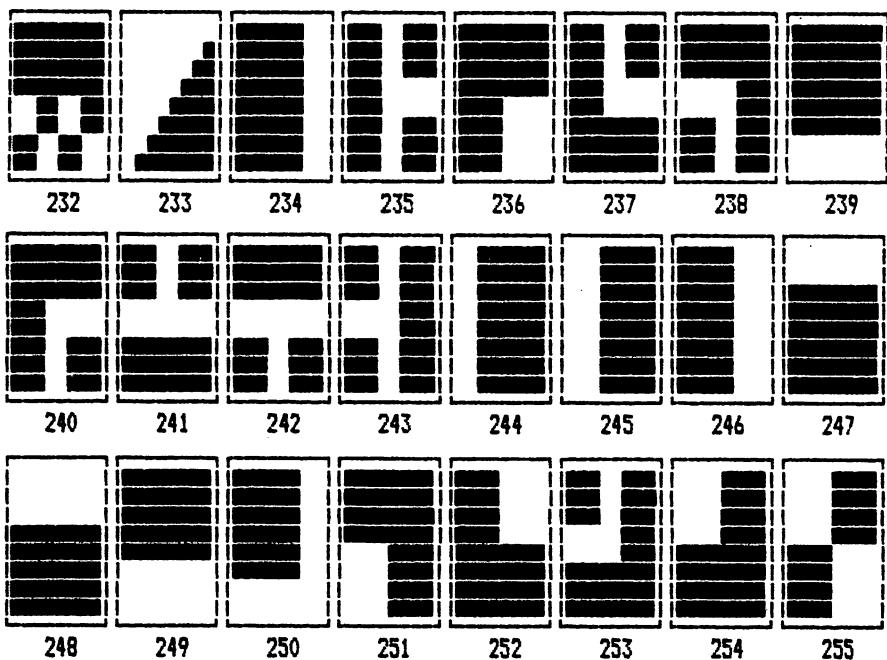
# Appendix I



## Appendix I

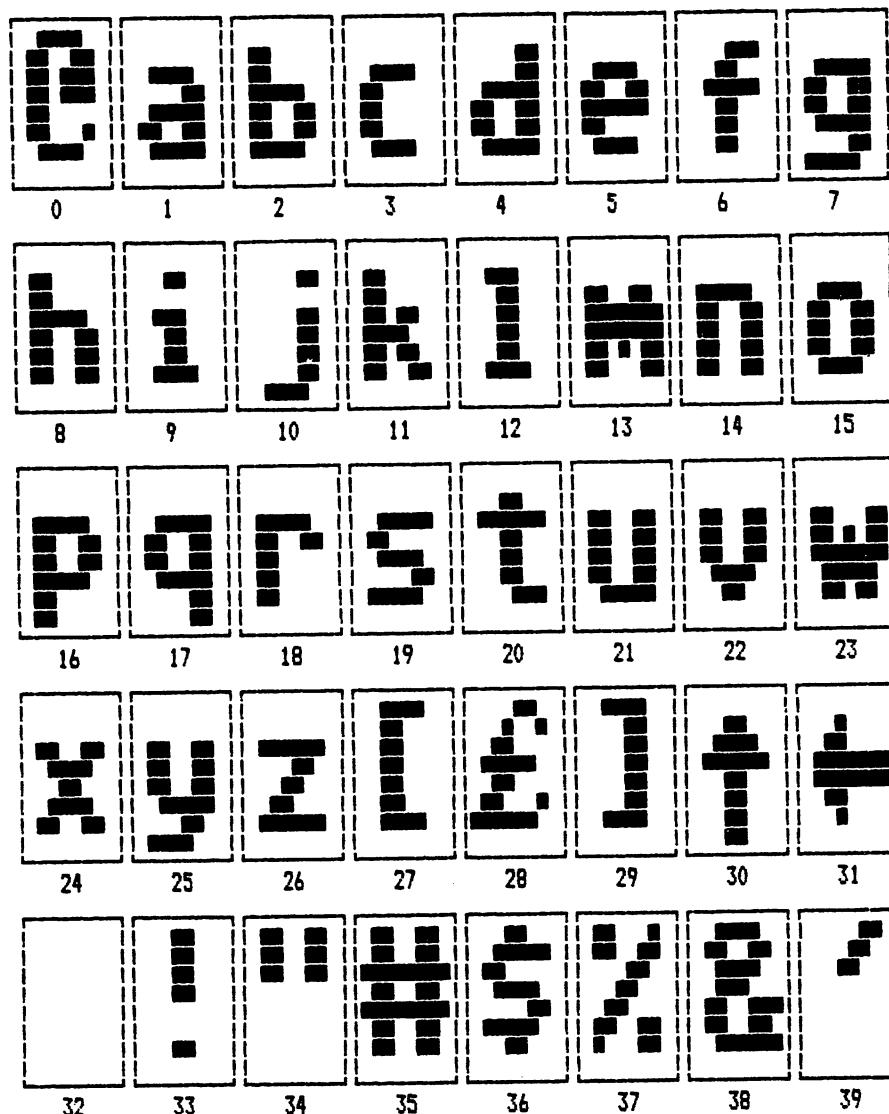


# **Appendix I**

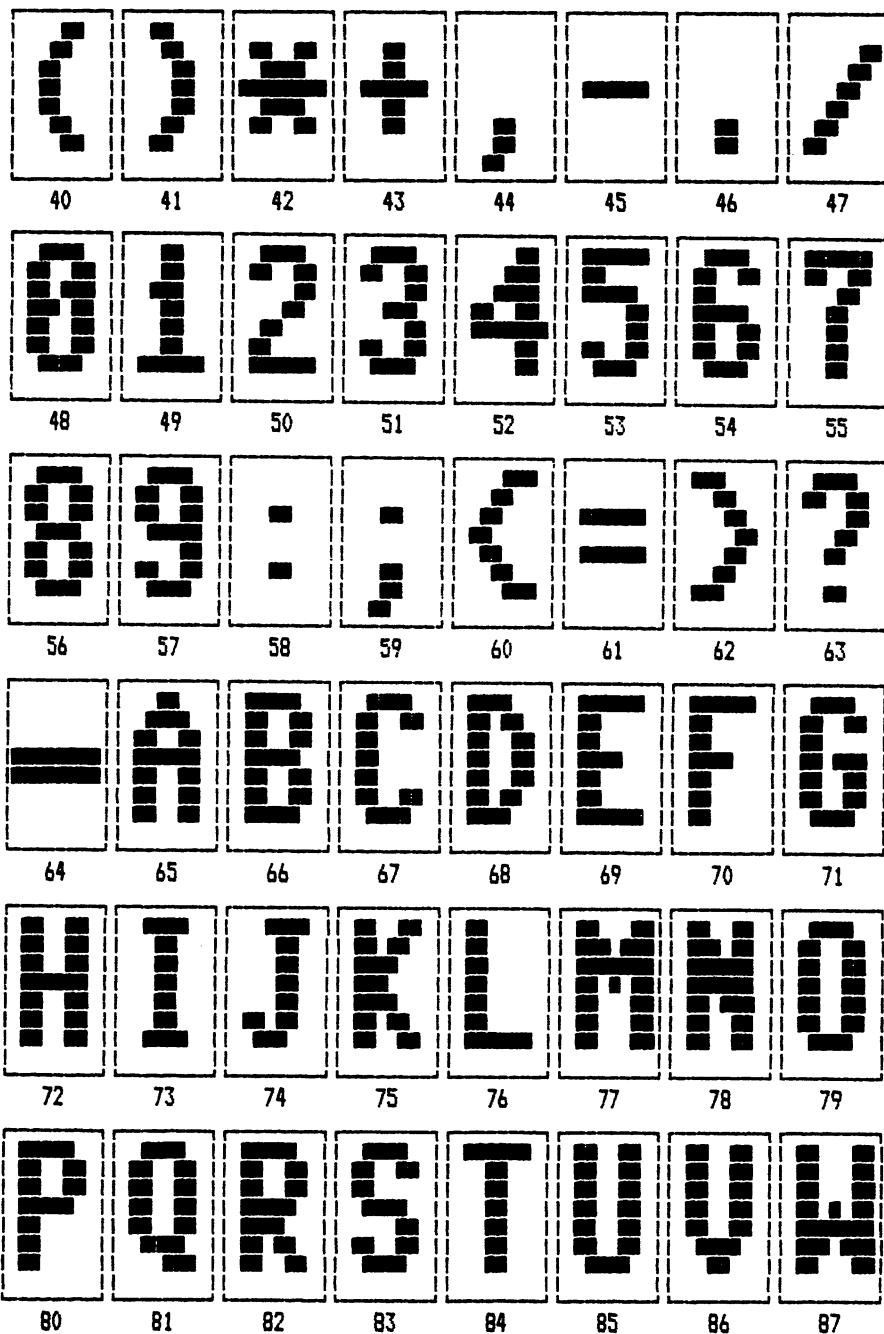


## Appendix J

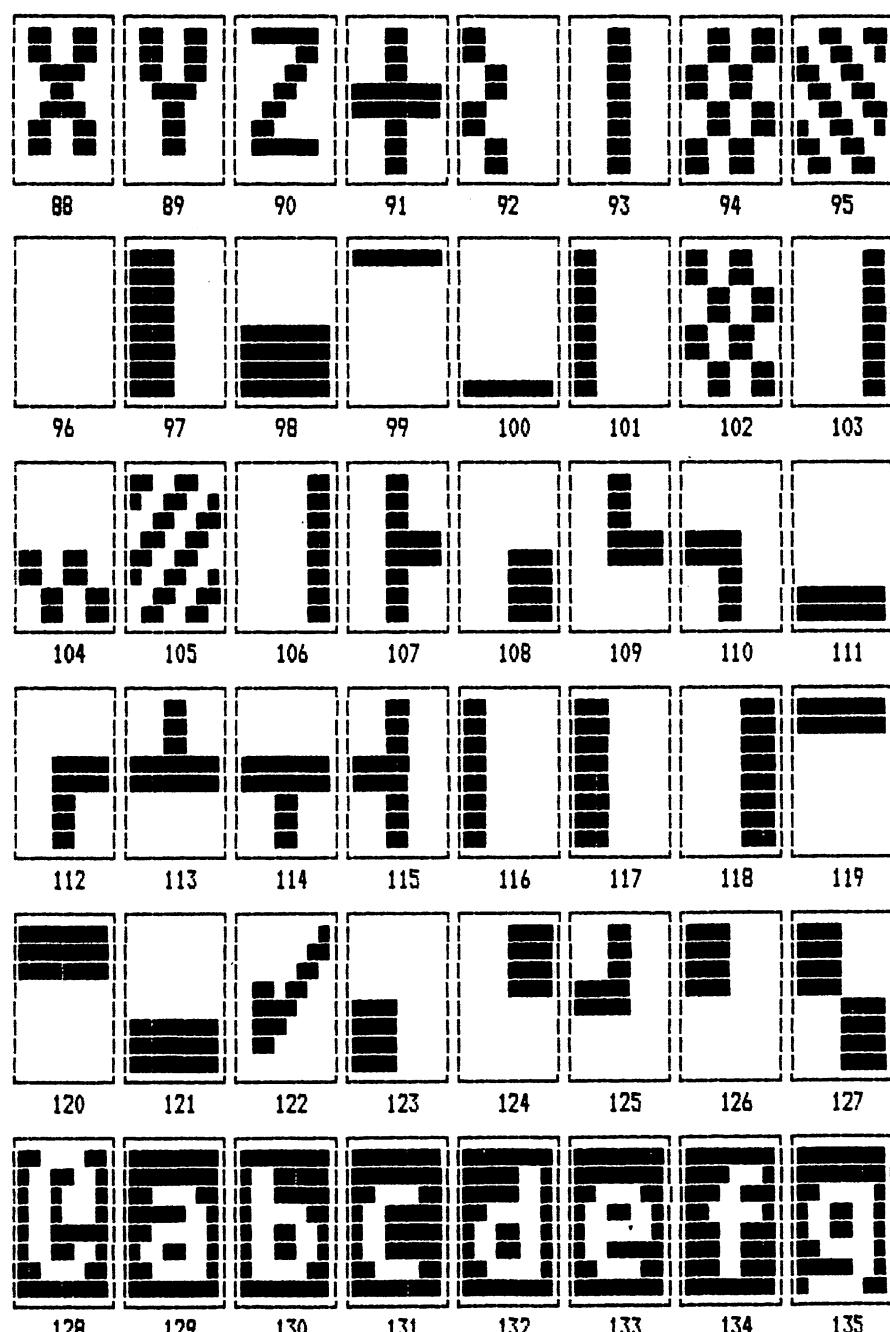
# Character Patterns and Screen Codes, Set 2



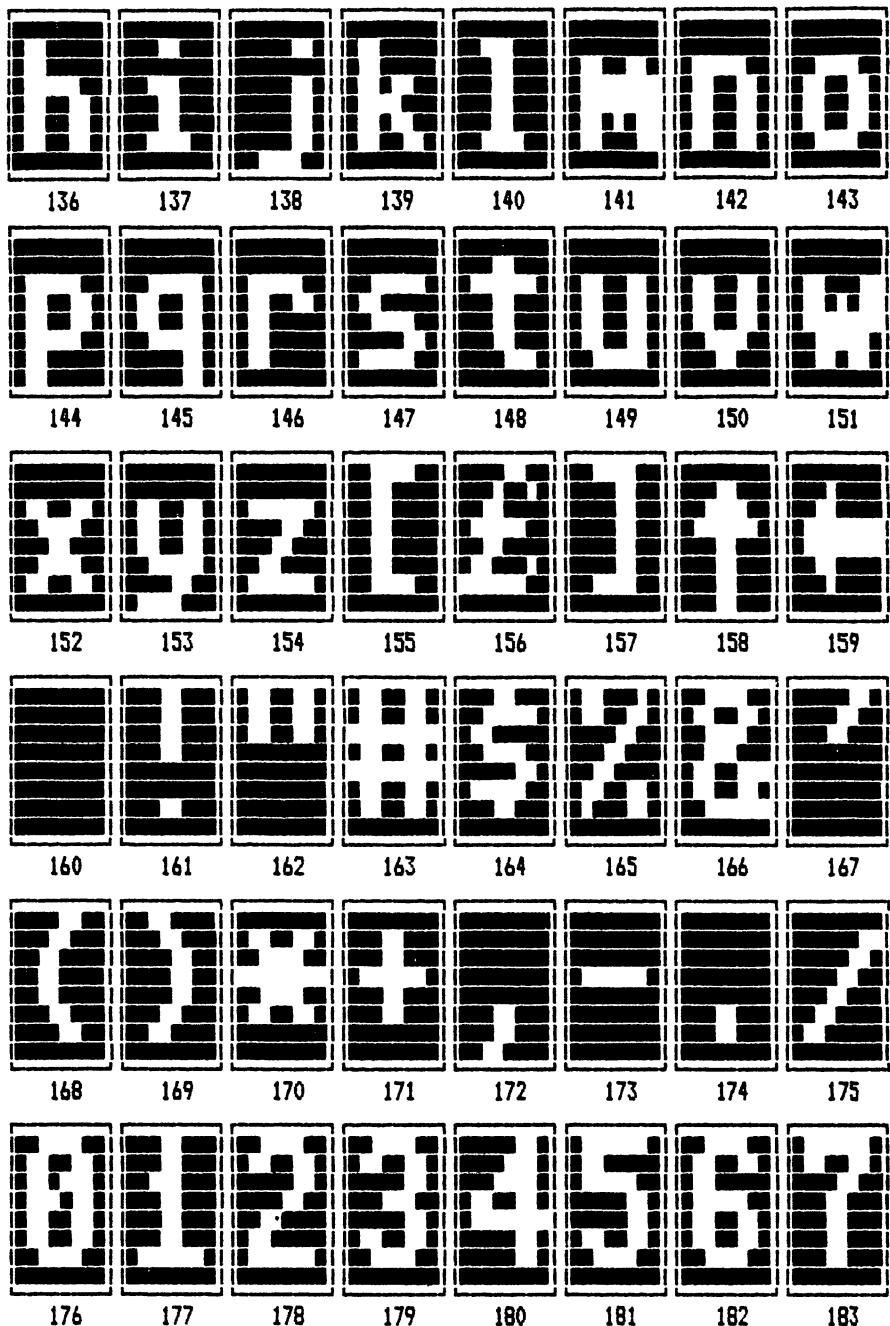
## Appendix J



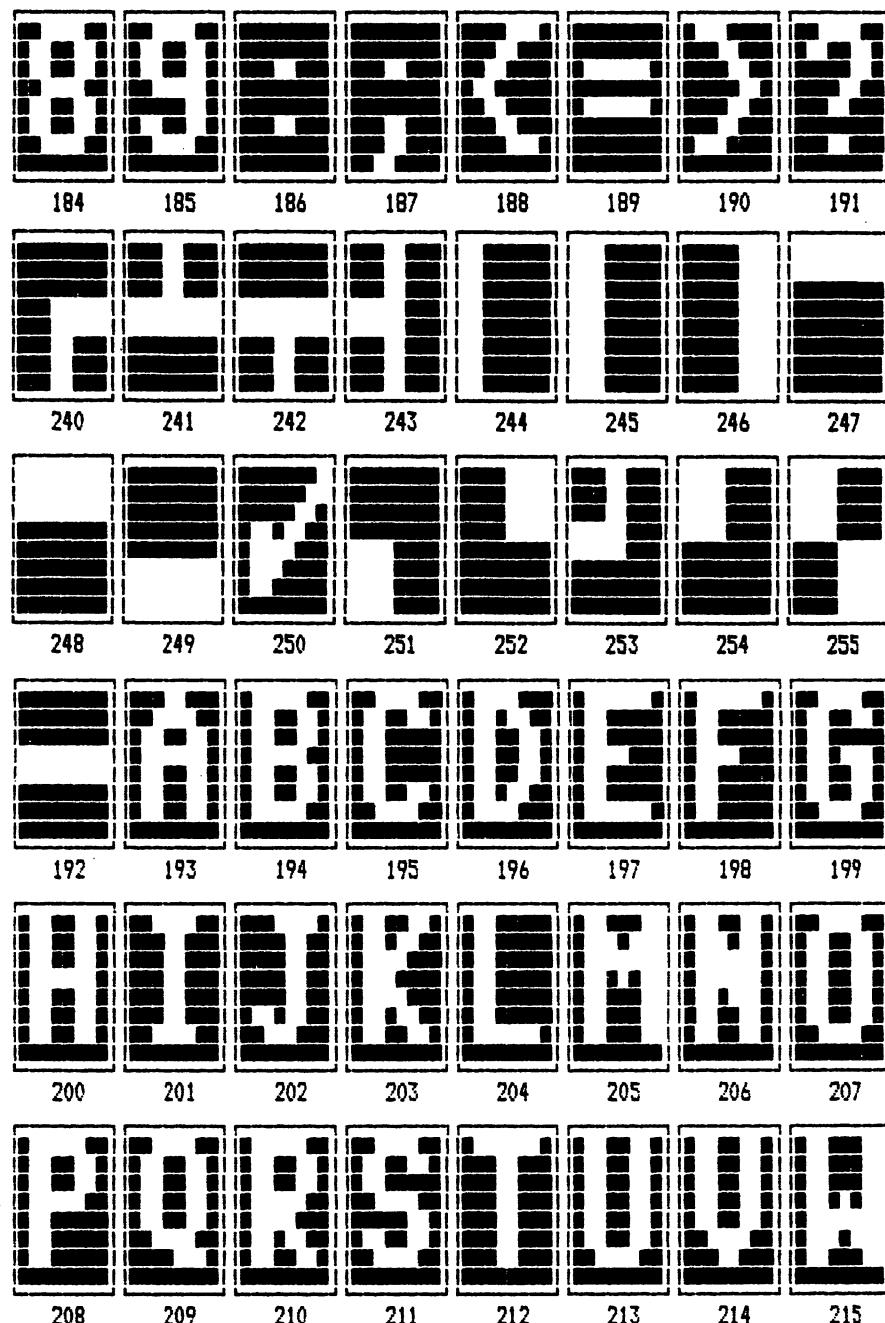
## Appendix J



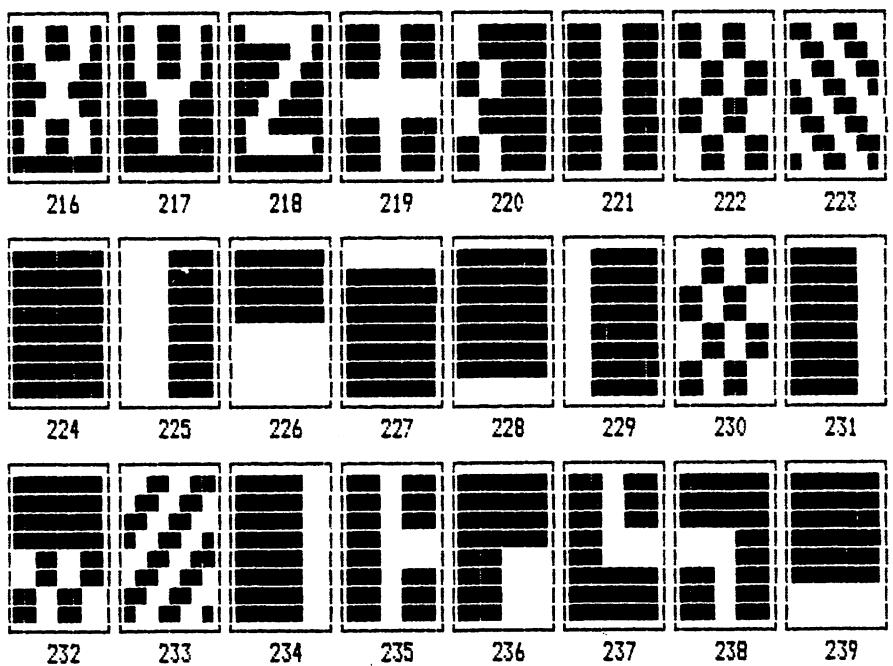
## Appendix J



## Appendix J



## Appendix J

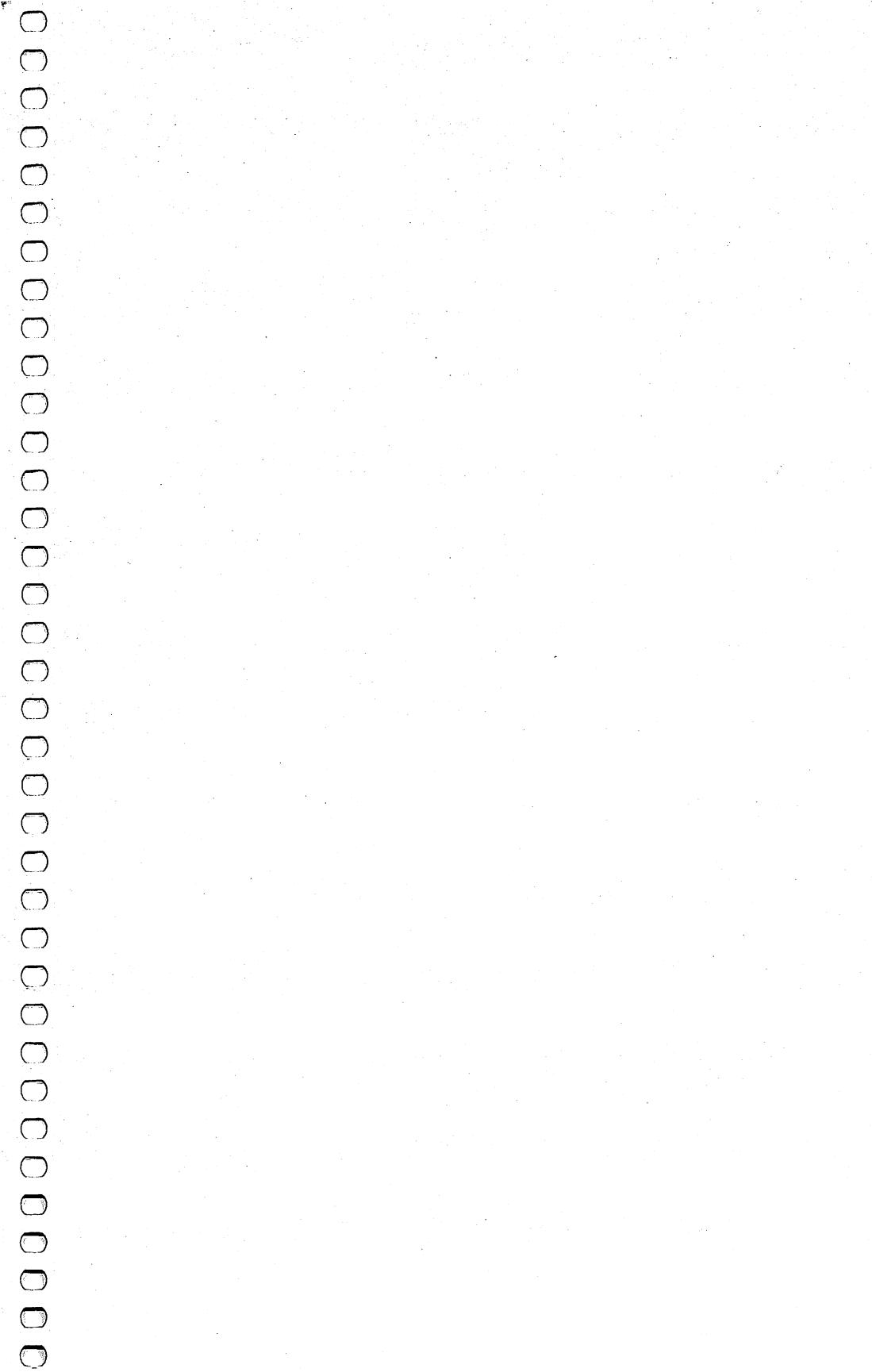


# Index

- alternate character set 6
- animation 153-76
- AND in bit masking 18, 88
- ASCII codes 144
- background bit-pair 141
- background colors (multiple) 141-47
- background registers 141-42
- bank switching 11
  - video memory banks 17-18
- bit masking 1
- bit-pairs 107
- bitmap 69, 101
  - with characters 181-84
- bitmapped mode 61, 68
- blended colors 97-98
- cells, in bitmap 64, 101-03
- character animation 158-59
- character colors 89-90
- character matrix 12-13
- character memory 16
- character ROM 11-12, 16, 68
- color 2, 101
  - from screen memory in bitmap mode 69
  - in multicolor bitmapped mode 147-49
- color control keys 84-86
- color memory 8, 68, 87
- color memory codes 89
- color overlays 90
- Complex Interface Adapter *See CIA*
- compound characters 2
- cursor addressing 10
- cursor position 88
- custom characters 15-34
- "Custom Character Editor" program 22-34
- CHR\$ 83
  - with color keys 86
- CIA register 17
- display priority *See sprites priority*
- DATA statements 43, 180
- expanded mode of sprites 34
- extended color mode 141-47
  - values 147
- graphics shapes 8-9
- graphics utilities 179-81
- high-resolution graphics 61-79, 84
- high-resolution mode *See "bitmapped mode"*
- high-resolution screen 61-63
  - moving 70
- keyboard 4, 83
- machine language 185-86
- masks *See bit masking*
- movement, philosophy of 156
- movement, coherent 154
- movement, incoherent 154
- movement tables 168
- multicolor bitmapped mode 147
- multicolor sprite editor program 126-41
- MSB (most significant byte) register 170
- nybble 101
- pointers 40-42 *See sprite pointers*
- POKE to screen 6-7
  - to screen & color memory 8
  - for animation 157
- PRINT command 3, 5-6, 8-9
  - color windows 144-47
  - faster than POKE 144
- Read Only Memory *See ROM*
- redefining characters 159-60
- reverse color 98
- RAM 11
- ROM 11
- screen animation 174-76
- screen cells 64-68
- screen codes 6, 16
- screen editor 87
  - moving 18-20
- "Screen editor" program (hi-res) 72-79
  - color feature 103-07
  - multicolor mode 107
  - multicolor text mode 108-11
  - mixing modes 111-12
  - custom multicolor characters 113-14
  - operation 114-15
  - multicolor feature 114-25
  - multicolor sprites 125-26
- screen flipping 175
- screen layout — character graphics 7
- screen memory 6, 8, 68
- scrolling
  - avoidance 8
  - horizontal 175-76
  - smooth 188
- shading 98-101
- shape table — sprite 40
  - addressing 40-42
  - programming 53
- sketching routine (hi-res) 71-72
- sprite animation 164-72
- sprite colors 93-97

sprite control registers 35-39, 94  
sprite editor program 45-61  
sprite graphics 84  
sprite patterns (multiple) 171  
sprite pointers 166  
sprite position registers 167  
sprite priority 34, 95-97, 172-74  
sprite shapes *See* shape tables  
sprites 34  
    moving 42

enabling 42, 164-66  
disabling 42  
expanding 172  
strings (limit on size) 9  
subcharacters 163  
text graphics 3  
text mode 3-10, 62  
VIC II chip 4, 16, 20, 69  
windows 141  
    program 142-44



COMPUTE!'s Reference Guide to  
Commodore 64 Graphics

## A Picture Is Worth A Thousand Bytes

Here is the book that teaches you how to use every feature of the Commodore 64's powerful graphics. Beginning and advanced programmers alike will find clear, complete explanations of how to use every graphics mode. And to help you in designing your own visuals for the 64, we've included complete sprite, character, and screen editor programs that will turn your TV screen into a canvas and your 64 into a palette of color.

- Learn how to create screen displays in character and bit-mapped modes.
- Make the screen come alive with standard color, multicolor, and extended background color modes.
- Design sprites and characters painlessly with standard (and multicolor) sprite and character editors — complete programs ready for you to type in and run.
- All programs are in ~~BASIC~~, so you can study them and see how each technique is carried out.
- Animate your pictures with movement and shape-changing techniques.
- Dozens of example programs illustrate every graphics technique.

Noted Commodore author John Heilborn uses clear writing and full explanations to help you understand and master graphics programming. Whether you want to program graphics for business applications, games, educational programs, or just for fun, everything you need is right here.