

# CHAPTER 3

## PROGRAMMING GRAPHICS ON THE COMMODORE 64

- Graphics Overview
- Graphics Locations
- Standard Character Mode
- Programmable Characters
- Multi-Color Mode Graphics
- Extended Background Color Mode
- Bit Mapped Graphics
- Multi-Color Bit Map Mode
- Smooth Scrolling
- Sprites
- Other Graphics Features
- Programming Sprites—Another Look

## GRAPHICS OVERVIEW

All of the graphics abilities of the Commodore 64 come from the 6567 Video Interface Chip (also known as the VIC-II chip). This chip gives a variety of graphics modes, including a 40 column by 25 line text display, a 320 by 200 dot high resolution display, and SPRITES, small movable objects which make writing games simple. And if this weren't enough, many of the graphics modes can be mixed on the same screen. It is possible, for example, to define the top half of the screen to be in high resolution mode, while the bottom half is in text mode. And SPRITES will combine with anything! More on sprites later. First the other graphics modes.

The VIC-II chip has the following graphics display modes:

### A) CHARACTER DISPLAY MODES

#### 1) Standard Character Mode

- a) ROM characters
- b) RAM programmable characters

#### 2) Multi-Color Character Mode

- a) ROM characters
- b) RAM programmable characters

#### 3) Extended Background Color Mode

- a) ROM characters
- b) RAM programmable characters

### B) BIT MAP MODES

#### 1) Standard Bit Map Mode

#### 2) Multi-Color Bit Map Mode

### C) SPRITES

#### 1) Standard Sprites

#### 2) Multi-Color Sprites

## GRAPHICS LOCATIONS

Some general information first. There are 1000 possible locations on the Commodore 64 screen. Normally, the screen starts at location 1024 (\$0400 in HEXadecimal notation) and goes to location 2023. Each of these locations is 8 bits wide. This means that it can hold any integer number from 0 to 255. Connected with screen memory is a group of 1000 locations called **COLOR MEMORY** or **COLOR RAM**. These start at location 55296 (\$D800 in HEX) and go up to 56295. Each of the color RAM locations is 4 bits wide, which means that it can hold any integer number from 0 to 15. Since there are 16 possible colors that the Commodore 64 can use, this works out well.

In addition, there are 256 different characters that can be displayed at any time. For normal screen display, each of the 1000 locations in screen memory contains a code number which tells the VIC-II chip which character to display at that screen location.

The various graphics modes are selected by the 47 **CONTROL** registers in the VIC-II chip. Many of the graphics functions can be controlled by **POKE**ing the correct value into one of the registers. The VIC-II chip is located starting at 53248 (\$D000 in HEX) through 53294 (\$D02E in HEX).

## VIDEO BANK SELECTION

The VIC-II chip can access ("see") 16K of memory at a time. Since there is 64K of memory in the Commodore 64, you want to be able to have the VIC-II chip see all of it. There is a way. There are 4 possible **BANKS** (or sections) of 16K of memory. All that is needed is some means of controlling which 16K bank the VIC-II chip looks at. In that way, the chip can "see" the entire 64K of memory. The **BANK SELECT** bits that allow you access to all the different sections of memory are located in the **6526 COMPLEX INTERFACE ADAPTER CHIP #2 (CIA #2)**. The **POKE** and **PEEK BASIC** statements (or their machine language versions) are used to select a bank by controlling bits 0 and 1 of **PORT A** of **CIA#2** (location 56576 (or \$DD00 HEX)). These 2 bits must be set to *outputs* by setting bits 0 and 1 of location 56578 (\$DD02, HEX) to change banks. The following example shows this:

```
POKE 56578,(PEEK(56578)OR 3):REM MAKE SURE BITS 0 AND 1 ARE  
SET TO OUTPUTS  
POKE 56576,(PEEK(56576)AND 252)OR A:REM CHANGE BANKS
```

"A" should have one of the following values:

VALUE OF A	BITS	BANK	STARTING LOCATION	VIC-II CHIP RANGE
0	00	3	49152	(\$C000-\$FFFF)*
1	01	2	32768	(\$8000-\$BFFF)
2	10	1	16384	(\$4000-\$7FFF)*
3	11	0	0	(\$0000-\$3FFF) (DEFAULT VALUE)

This 16K bank concept is part of everything that the VIC-II chip does. You should always be aware of which bank the VIC-II chip is pointing at, since this will affect where character data patterns come from, where the screen is, where sprites come from, etc. When you turn on the power of your Commodore 64, bits 0 and 1 of location 56576 are automatically set to BANK 0 (\$0000-\$3FFF) for all display information.

\*NOTE: The Commodore 64 character set is not available to the VIC-II chip in BANKS 1 and 3. (See character memory section.)

## SCREEN MEMORY

The location of screen memory can be changed easily by a POKE to control register 53272 (\$D018 HEX). However, this register is also used to control which character set is used, so be careful to avoid disturbing that part of the control register. The **UPPER 4** bits control the location of screen memory. To move the screen, the following statement should be used:

```
POKE53272,(PEEK(53272)AND15)ORA
```

Where "A" has one of the following values:

A	BITS	LOCATION*	
		DECIMAL	HEX
0	0000XXXX	0	\$0000
16	0001XXXX	1024	\$0400 (DEFAULT)
32	0010XXXX	2048	\$0800
48	0011XXXX	3072	\$0C00
64	0100XXXX	4096	\$1000
80	0101XXXX	5120	\$1400
96	0110XXXX	6144	\$1800
112	0111XXXX	7168	\$1C00
128	1000XXXX	8192	\$2000
144	1001XXXX	9216	\$2400
160	1010XXXX	10240	\$2800
176	1011XXXX	11264	\$2C00
192	1100XXXX	12288	\$3000
208	1101XXXX	13312	\$3400
224	1110XXXX	14336	\$3800
240	1111XXXX	15360	\$3C00

\*Remember that the BANK ADDRESS of the VIC-II chip must be added in.

You must also tell the KERNAL'S screen editor where the screen is as follows: POKE 648, page (where page = address/256, e.g., 1024/256 = 4, so POKE 648,4).

## COLOR MEMORY

Color memory can NOT move. It is always located at locations 55296 (\$D800) through 56295 (\$DBE7). Screen memory (the 1000 locations starting at 1024) and color memory are used differently in the different graphics modes. A picture created in one mode will often look completely different when displayed in another graphics mode.

## CHARACTER MEMORY

Exactly where the VIC-II gets its character information is important to graphic programming. Normally, the chip gets the shapes of the characters you want to be displayed from the **CHARACTER GENERATOR ROM**. In this chip are stored the *patterns* which make up the various letters, numbers, punctuation symbols, and the other things that you see

on the keyboard. One of the features of the Commodore 64 is the ability to use patterns located in RAM memory. These RAM patterns are created by you, and that means that you can have an almost infinite set of symbols for games, business applications, etc.

A normal character set contains 256 characters in which each character is defined by 8 bytes of data. Since each character takes up 8 bytes this means that a full character set is  $256 * 8 = 2K$  bytes of memory. Since the VIC-II chip looks at 16K of memory at a time, there are 8 possible locations for a complete character set. Naturally, you are free to use less than a full character set. However, it must still start at one of the 8 possible starting locations.

The location of character memory is controlled by 3 bits of the VIC-II control register located at 53272 (\$D018 in HEX notation). Bits 3, 2, and 1 control where the characters' set is located in 2K blocks. Bit 0 is ignored. Remember that this is the same register that determines where screen memory is located so avoid disturbing the screen memory bits. To change the location of character memory, the following BASIC statement can be used:

```
POKE 53272,(PEEK(53272)AND240)OR A
```

Where A is one of the following values:

VALUE of A	BITS	LOCATION OF CHARACTER MEMORY**	
		DECIMAL	HEX
0	XXXX000X	0	\$0000-\$07FF
2	XXXX001X	2048	\$0800-\$0FFF
4	XXXX010X	4096	\$1000-\$17FF ROM IMAGE in BANK 0 & 2 (default)
6	XXXX011X	6144	\$1800-\$1FFF ROM IMAGE in BANK 0 & 2
8	XXXX100X	8192	\$2000-\$27FF
10	XXXX101X	10240	\$2800-\$2FFF
12	XXXX110X	12288	\$3000-\$37FF
14	XXXX111X	14336	\$3800-\$3FFF

\*\*Remember to add in the BANK address.

The **ROM IMAGE** in the above table refers to the character generator ROM. It appears in place of RAM at the above locations in bank 0. It also appears in the corresponding RAM at locations 36864–40959 (\$9000–\$9FFF) in bank 2. Since the VIC-II chip can only access 16K of memory at a time, the ROM character patterns appear in the 16K block of memory the VIC-II chip looks at. Therefore, the system was designed to make the VIC-II chip think that the ROM characters are at 4096–8191 (\$1000–\$1FFF) when your data is in bank 0, and 36864–40959 (\$9000–\$9FFF) when your data is in bank 2, even though the character ROM is actually at location 53248–57343 (\$D000–\$DFFF). This imaging only applies to character data as seen by the VIC-II chip. It can be used for programs, other data, etc., just like any other RAM memory.

**NOTE:** If these ROM images get in the way of your own graphics, then set the **BANK SELECT BITS** to one of the **BANKS** without the images (BANKS 1 or 3). The ROM patterns won't be there.

The location and contents of the character set in ROM are as follows:

BLOCK	ADDRESS		VIC-II IMAGE	CONTENTS
	DECIMAL	HEX		
0	53248	D000–D1FF	1000–11FF	Upper case characters
	53760	D200–D3FF	1200–13FF	Graphics characters
	54272	D400–D5FF	1400–15FF	Reversed upper case characters
	54784	D600–D7FF	1600–17FF	Reversed graphics characters
1	55296	D800–D9FF	1800–19FF	Lower case characters
	55808	DA00–DBFF	1A00–1BFF	Upper case & graphics characters
	56320	DC00–DDFF	1C00–1DFF	Reversed lower case characters
	56832	DE00–DFFF	1E00–1FFF	Reversed upper case & graphics characters

Sharp-eyed readers will have just noticed something. The locations occupied by the character ROM are the same as the ones occupied by the VIC-II chip control registers. This is possible because they don't occupy the same locations at the same time. When the VIC-II chip needs to

access character data the ROM is switched in. It becomes an image in the 16K bank of memory that the VIC-II chip is looking at. Otherwise, the area is occupied by the I/O control registers, and the character ROM is only available to the VIC-II chip.

However, you may need to get to the character ROM if you are going to use programmable characters and want to copy some of the character ROM for some of your character definitions. In this case you must switch out the I/O register, switch in the character ROM, and do your copying. When you're finished, you must switch the I/O registers back in again. During the copying process (when I/O is switched out) no interrupts can be allowed to take place. This is because the I/O registers are needed to service the interrupts. If you forget and perform an interrupt, really strange things happen. The keyboard should not be read during the copying process. To turn off the keyboard and other normal interrupts that occur with your Commodore 64, the following POKE should be used:

**POKE 56334,PEEK(56334)AND254 (TURNS INTERRUPTS OFF)**

After you are finished getting characters from the character ROM, and are ready to continue with your program, you must turn the keyboard scan back on by the following POKE:

**POKE 56334,PEEK(56334)OR1 (TURNS INTERRUPTS ON)**

The following POKE will switch out I/O and switch the CHARACTER ROM in:

**POKE 1,PEEK(1)AND251**

The character ROM is now in the locations from 53248-57343 (\$D000-\$DFFF).

To switch I/O back into \$D000 for normal operation use the following POKE:

**POKE 1,PEEK(1)OR 4**

## STANDARD CHARACTER MODE

Standard character mode is the mode the Commodore 64 is in when you first turn it on. It is the mode you will generally program in.

Characters can be taken from ROM or from RAM, but normally they are taken from ROM. When you want special graphics characters for a program, all you have to do is define the new character shapes in RAM, and tell the VIC-II chip to get its character information from there instead of the character ROM. This is covered in more detail in the next section.

In order to display characters on the screen in color, the VIC-II chip accesses the screen memory to determine the character code for that location on the screen. At the same time, it accesses the color memory to determine what color you want for the character displayed. The character code is translated by the VIC-II into the starting address of the 8-byte block holding your character pattern. The 8-byte block is located in character memory.

The translation isn't too complicated, but a number of items are combined to generate the desired address. First the character code you use to POKE screen memory is multiplied by 8. Next add the start of character memory (see CHARACTER MEMORY section). Then the Bank Select Bits are taken into account by adding in the base address (see VIDEO BANK SELECTION section). Below is a simple formula to illustrate what happens:

$$\text{CHARACTER ADDRESS} = \text{SCREEN CODE} * 8 + (\text{CHARACTER SET} * 2048) + (\text{BANK} * 16384)$$

## CHARACTER DEFINITIONS

Each character is formed in an 8 by 8 grid of dots, where each dot may be either on or off. The Commodore 64 character images are stored in the Character Generator ROM chip. The characters are stored as a set of 8 bytes for each character, with each byte representing the dot pattern of a row in the character, and each bit representing a dot. A zero bit means that dot is off, and a one bit means the dot is on.

The character memory in ROM begins at location 53248 (when the I/O is switched off). The first 8 bytes from location 53248 (\$D000) to 53255 (\$D007) contain the pattern for the @ sign, which has a character code value of zero in the screen memory. The next 8 bytes, from location

53256 (\$D008) to 53263 (\$D00F), contain the information for forming the letter A.

IMAGE	BINARY	PEEK
**	00011000	24
****	00111100	60
** **	01100110	102
*****	01111110	126
** **	01100110	102
** **	01100110	102
** **	01100110	102
	00000000	0

Each complete character set takes up 2K (2048 bits) of memory, 8 bytes per character and 256 characters. Since there are two character sets, one for upper case and graphics and the other with upper and lower case, the character generator ROM takes up a total of 4K locations.

## PROGRAMMABLE CHARACTERS

Since the characters are stored in ROM, it would seem that there is no way to change them for customizing characters. However, the memory location that tells the VIC-II chip where to find the characters is a programmable register which can be changed to point to many sections of memory. By changing the character memory pointer to point to RAM, the character set may be programmed for any need.

If you want your character set to be located in RAM, there are a few **VERY IMPORTANT** things to take into account when you decide to actually program your own character sets. In addition, there are two other important points you must know to create your own special characters:

- 1) It is an all or nothing process. Generally, if you use your own character set by telling the VIC-II chip to get the character information from the area you have prepared in RAM, the standard Commodore 64 characters are unavailable to you. To solve this, you must copy any letters, numbers, or standard Commodore 64 graphics you intend to use into your own character memory in RAM. You can pick and choose, take only the ones you want, and don't even have to keep them in order!

- 2) Your character set takes memory space away from your BASIC program. Of course, with 38K available for a BASIC program, most applications won't have problems.

**WARNING:** You must be careful to protect the character set from being overwritten by your BASIC program, which also uses the RAM.

There are two locations in the Commodore 64 to start your character set that **should NOT be used with BASIC: location 0 and location 2048**. The first should not be used because the system stores important data on page 0. The second can't be used because that is where your BASIC program starts! However, there are 6 other starting positions for your custom character set.

The best place to put your character set for use with BASIC while experimenting is beginning at 12288 (\$3000 in HEX). This is done by POKEing the low 4 bits of location 53272 with 12. Try the POKE now, like this:

```
POKE 53272,(PEEK(53272)AND240)+12
```

Immediately, all the letters on the screen turn to garbage. This is because there are no characters set up at location 12288 right now . . . only random bytes. Set the Commodore 64 back to normal by hitting the **RUN/STOP** key and then the **RESTORE** key.

Now let's begin creating graphics characters. To protect your character set from BASIC, you should reduce the amount of memory BASIC thinks it has. The amount of memory in your computer stays the same. . . it's just that you've told BASIC not to use some of it. Type:

```
PRINT FRE(0)-(SGN(FRE(0))<0)*65535
```

The number displayed is the amount of memory space left unused. Now type the following:

```
POKE 52,48:POKE56,48:CLR
```

Now type:

```
PRINT FRE(0)-(SGN(FRE(0))<0)*65535
```

See the change? BASIC now thinks it has less memory to work with. The memory you just claimed from BASIC is where you are going to put your character set; safe from actions of BASIC.

The next step is to put your characters into RAM. When you begin, there is random data beginning at 12288 (\$3000 HEX). You must put character patterns in RAM (in the same style as the ones in ROM) for the VIC-II chip to use.

The following program moves 64 characters from ROM to your character set RAM:

```
5 PRINTCHR$(142)           :REM SWITCH TO
UPPER CASE
10 POKE52.48:POKE56.48:CLR   :REM RESERVE MEMORY
FOR CHARACTERS$
20 POKE56334,PEEK(56334)AND254 :REM TURN OFF
KEYSCAN INTERRUPT TIMER
30 POKE1,PEEK(1)AND251      :REM SWITCH IN
CHARACTER
40 FOR I=#0TO511:POKEI+12288,PEEK(I+53240):NEXT
50 POKE1,PEEK(1)OR4        :REM SWITCH IN I/O
60 POKE56334,PEEK(56334)OR1 :REM RESTART
KEYSCAN INTERRUPT TIMER
70 END
```

Now POKE location 53272 with  $(PEEK(53272)AND240)+12$ . Nothing happens, right? Well, almost nothing. The Commodore 64 is now getting its character information from your RAM, instead of from ROM. But since we copied the characters from ROM exactly, no difference can be seen. . . . yet.

You can easily change the characters now. Clear the screen and type an @ sign. Move the cursor down a couple of lines, then type:

```
FOR I = 12288 TO 12288+7:POKE I, 255 - PEEK(I) : NEXT
```

You just created a reversed @ sign!

**TIP:** Reversed characters are just characters with their bit patterns in character memory reversed.

Now move the cursor up to the program again and hit **RETURN** again to re-reverse the character (bring it back to normal). By looking at the table of screen display codes, you can figure out where in RAM each character is. Just remember that each character takes eight memory locations to store. Here's a few examples just to get you started:

CHARACTER	DISPLAY CODE	CURRENT STARTING LOCATION IN RAM
@	0	12288
A	1	12296
!	33	12552
>	62	12784

Remember that we only took the first 64 characters. Something else will have to be done if you want one of the other characters.

What if you wanted character number 154, a reversed Z? Well, you could make it yourself, by reversing a Z, or you could copy the set of reversed characters from the ROM, or just take the one character you want from ROM and replace one of the characters you have in RAM that you don't need.

Suppose you decide that you won't need the > sign. Let's replace the > sign with the reversed Z. Type this:

```
FOR I=0 TO 7: POKE 12784 + I, 255-PEEK(I+12496): NEXT
```

Now type a > sign. It comes up as a reversed Z. No matter how many times you type the >, it comes out as a reversed Z. (This change is really an illusion. Though the > sign looks like a reversed Z, it still acts like a > in a program. Try something that needs a > sign. It will still work fine, only it will look strange.)

**A quick review:** You can now copy characters from ROM into RAM. You can even pick and choose only the ones you want. There's only one step left in programmable characters (the best step!) . . . making your own characters.

Remember how characters are stored in ROM? Each character is stored as a group of eight bytes. The bit patterns of the bytes directly control the character. If you arrange 8 bytes, one on top of another, and write out each byte as eight binary digits, it forms an eight by eight matrix, looking like the characters. When a bit is a one, there is a dot at that location. When a bit is a zero, there is a space at that location.

When creating your own characters, you set up the same kind of table in memory. Type NEW and then type this program:

```
10 FOR I = 12448 TO 12455 : READ A: POKE I,A: NEXT
20 DATA 60, 66, 165, 129, 165, 153, 66, 60
```

Now type RUN. The program will replace the letter T with a smile face character. Type a few T's to see the face. Each of the numbers in the DATA statement in line 20 is a row in the smile face character. The matrix for the face looks like this:

	7	6	5	4	3	2	1	0	BINARY	DECIMAL
ROW 0			*	*	*	*			00111100	60
1		*						*	01000010	66
2	*		*			*		*	10100101	165
3	*							*	10000001	129
4	*		*		*		*	*	10100101	165
5	*			*	*			*	10011001	153
6		*						*	01000010	66
ROW 7			*	*	*	*			00111100	60

	7	6	5	4	3	2	1	0
0								
1								
2								
3								
4								
5								
6								
7								

Figure 3-1. Programmable Character Worksheet.

The Programmable Character Worksheet (Figure 3-1) will help you design your own characters. There is an 8 by 8 matrix on the sheet, with row numbers, and numbers at the top of each column. (If you view each row as a binary word, the numbers are the value of that bit position. Each is a power of 2. The leftmost bit is equal to 128 or 2 to the 7th power, the next is equal to 64 or 2 to the 6th, and so on, until you reach the rightmost bit (bit 0) which is equal to 1 or 2 to the 0 power.)

Place an X on the matrix at every location where you want a dot to be in your character. When your character is ready you can create the DATA statement for your character.

Begin with the first row. Wherever you placed an X, take the number at the top of the column (the power-of-2 number, as explained above) and write it down. When you have the numbers for every column of the first row, add them together. Write this number down, next to the row. This is the number that you will put into the DATA statement to draw this row.

Do the same thing with all of the other rows (1-7). When you are finished you should have 8 numbers between 0 and 255. If any of your numbers are not within range, recheck your addition. The numbers must be in this range to be correct! If you have less than 8 numbers, you missed a row. It's OK if some are 0. The 0 rows are just as important as the other numbers.

Replace the numbers in the DATA statement in line 20 with the numbers you just calculated, and RUN the program. Then type a T. Every time you type it, you'll see your own character!

If you don't like the way the character turned out, just change the numbers in the DATA statement and re-RUN the program until you are happy with your character.

That's all there is to it!

**HINT:** For best results, always make any vertical lines in your characters at least 2 dots (bits) wide. This helps prevent CHROMA noise (color distortion) on your characters when they are displayed on a TV screen.

Here is an example of a program using standard programmable characters:

```
10 REM * EXAMPLE 1 *
20 REM CREATING PROGRAMMABLE CHARACTERS
31 POKE56334,PEEK(56334)AND254:POKE1,PEEK(1)AND251:
REM TURN OFF KB AND I/O
35 FORI=0TO63:REM CHARACTER RANGE TO BE COPIED
FROM ROM
36 FORJ=0TO7:REM COPY ALL 8 BYTES PER CHARACTER
37 POKE12288+I*8+J,PEEK(53248+I*8+J):REM COPY A
BYTE
38 NEXTJ:NEXTI:REM GOTO NEXT BYTE OR CHARACTER
39 POKE1,PEEK(1)OR4:POKE56334,PEEK(56334)OR1:REM
TURN ON I/O AND KB
40 POKE53272,(PEEK(53272)AND240)+12:REM SET CHAR
POINTER TO MEM. 12288
60 FORCHAR=60TO63:REM PROGRAM CHARACTERS 60 THRU 63
80 FORBYTE=0TO7:REM DO ALL 8 BYTES OF A CHARACTER
100 READ NUMBER:REM READ IN 1/8TH OF CHARACTER DATA
120 POKE12288+(8*CHAR)+BYTE,NUMBER:REM STORE THE
DATA IN MEMORY
140 NEXTBYTE:NEXTCHAR:REM ALSO COULD BE NEXT BYTE,
CHAR
150 PRINTCHR$(147)TAB(255)CHR$(60);
155 PRINTCHR$(61)TAB(55)CHR$(62)CHR$(63)
160 REM LINE 150 PUTS THE NEWLY DEFINED CHARACTERS
ON THE SCREEN
170 GETA$:REM WAIT FOR USER TO PRESS A KEY
180 IFA$=""THENGOTO170:REM IF NO KEYS WERE PRESSED,
TRY AGAIN!
190 POKE53272,21:REM RETURN TO NORMAL CHARACTERS
200 DATA4,6,7,5,7,7,3,3:REM DATA FOR CHARACTER 60
210 DATA 32,96,224,160,224,224,192,192:REM DATA
FOR CHARACTER 61
220 DATA7,7,7,31,31,95,143,127:REM DATA FOR
CHARACTER 62
230 DATA 224,224,224,248,248,248,240,224:REM DATA
FOR CHARACTER 63
240 END
```

## MULTI-COLOR MODE GRAPHICS

Standard high-resolution graphics give you control of very small dots on the screen. Each dot in character memory can have 2 possible values, 1 for on and 0 for off. When a dot is off, the color of the screen is used in the space reserved for that dot. If the dot is on, the dot is colored with the character color you have chosen for that screen position. When you're using standard high-resolution graphics, all the dots within each 8×8 character can either have background color or foreground color. In some ways this limits the color resolution within that space. For example, problems may occur when two different colored lines cross.

Multi-color mode gives you a solution to this problem. Each dot in multi-color mode can be one of 4 colors: screen color (background color register #0), the color in background register #1, the color in background color register #2, or character color. The only sacrifice is in the horizontal resolution, because each multi-color mode dot is twice as wide as a high-resolution dot. This minimal loss of resolution is more than compensated for by the extra abilities of multi-color mode.

### MULTI-COLOR MODE BIT

To turn on multi-color character mode, set bit 4 of the VIC-II control register at 53270 (\$D016) to a 1 by using the following POKE:

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

To turn off multi-color character mode, set bit 4 of location 53270 to a 0 by the following POKE:

```
POKE 53270,PEEK(53270)AND 239
```

Multi-color mode is set on or off for each space on the screen, so that multi-color graphics can be mixed with high-resolution (hi-res) graphics. This is controlled by bit 3 in color memory. Color memory begins at location 55296 (\$D800 in HEX). If the number in color memory is less than 8 (0-7) the corresponding space on the video screen will be standard hi-res, in the color (0-7) you've chosen. If the number located in color memory is greater or equal to 8 (from 8 to 15), then that space will be displayed in multi-color mode.

By POKEing a number into color memory, you can change the color of the character in that position on the screen. POKEing a number from 0 to 7 gives the normal character colors. POKEing a number between 8 and 15 puts the space into multi-color mode. In other words, turning BIT 3 ON in color memory, sets MULTI-COLOR MODE. Turning BIT 3 OFF in color memory, sets the normal, HIGH-RESOLUTION mode.

Once multi-color mode is set in a space, the bits in the character determine which colors are displayed for the dots. For example, here is a picture of the letter A, and its bit pattern:

IMAGE	BIT PATTERN
**	00011000
*****	00111100
** **	01100110
*****	01111110
** **	01100110
** **	01100110
** **	01100110
	00000000

In normal or high-resolution mode, the screen color is displayed everywhere there is a 0 bit, and the character color is displayed where the bit is a 1. Multi-color mode uses the bits in pairs, like so:

IMAGE	BIT PATTERN
AABB	00 01 10 00
CCCC	00 11 11 00
AABBAABB	01 10 01 10
AACCCCB	01 11 11 10
AABBAABB	01 10 01 10
AABBAABB	01 10 01 10
AABBAABB	01 10 01 10
	00 00 00 00

In the image area above, the spaces marked AA are drawn in the background #1 color, the spaces marked BB use the background #2 color, and the spaces marked CC use the character color. The bit pairs determine this, according to the following chart:

BIT PAIR	COLOR REGISTER	LOCATION
00	Background #0 color (screen color)	53281 (\$D021)
01	Background #1 color	53282 (\$D022)
10	Background #2 color	53283 (\$D023)
11	Color specified by the lower 3 bits in color memory	color RAM

**NOTE:** The sprite foreground color is a 10. The character foreground color is a 11.

Type NEW and then type this demonstration program:

```

100 POKE53281,1:REM SET BACKGROUND COLOR #0 TO
WHITE
110 POKE53282,3:REM SET BACKGROUND COLOR #1 TO CYAN
120 POKE53283,8:REM SET BACKGROUND COLOR #2 TO
ORANGE
130 POKE53270,PEEK(53270)OR16:REM TURN ON
MULTICOLOR MODE
140 C=13*4096+8*255:REM SET C TO POINT TO COLOR
MEMORY
150 PRINTCHR$(147)"AAAAAAAAAA"
160 FORL=0TOD9
170 POKEC+L,8:REM USE MULTI BLACK
180 NEXT

```

The screen color is white, the character color is black, one color register is cyan (greenish blue), the other is orange.

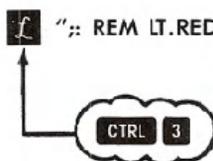
You're not really putting color codes in the space for character color, you're actually using references to the registers associated with those colors. This conserves memory, since 2 bits can be used to pick 16 colors (background) or 8 colors (character). This also makes some neat tricks possible. Simply changing one of the indirect registers will change every dot drawn in that color. Therefore everything drawn in the screen and

background colors can be changed on the whole screen instantly. Here is an example of changing background color register #1:

```
100 POKE53270,PEEK(53270)OR16:REM TURN ON
MULTICOLOR MODE
110 PRINTCHR$(147)CHR$(18);
120 PRINT"███";REM TYPE C= & 1 FOR ORANGE OR
MULTICOLOR BLACK BACKGROUND
130 FORL=1TO22:PRINTCHR$(65);:NEXT
135 FORT=1TO500:NEXT
140 PRINT"███";REM TYPE CTRL & 7 FOR BLUE COLOR
CHANGE
145 FORT=1TO500:NEXT
150 PRINT"███";REM TYPE CTRL & 1
160 BETA$ IFA$=""THEN160
170 X=INT(RND(1)*16)
180 POKE 53282,X
190 GOTO 160
```

By using the **C** key and the COLOR keys the characters can be changed to any color, including multi-color characters. For example, type this command:

```
POKE 53270,PEEK(53270)OR 16:PRINT "███"; REM LT.RED/
MULTI-COLOR RED
```



The word READY and anything else you type will be displayed in multi-color mode. Another color control can set you back to regular text.

Here is an example of a program using multi-color programmable characters:

```
10 REM * EXAMPLE 2 *
20 REM CREATING MULTI COLOR PROGRAMMABLE CHARACTERS
31 POKE56334,PEEK(56334)AND254:POKE1,PEEK(1)AND251
35 FORI=0T063:REM CHARACTER RANGE TO BE COPIED
FROM ROM
36 FORJ=0T07:REM COPY ALL 8 BYTES PER CHARACTER
37 POKE12288+I*8+J,PEEK(53248+I*8+J):REM COPY A
BYTE
38 NEXTJ,I:REM GOTO NEXT BYTE OR CHARACTER
39 POKE1,PEEK(1)OR4:POKE56334,PEEK(56334)OR1:REM
TURN ON I/O AND KB
40 POKE53272,(PEEK(53272)AND240)+12:REM SET CHAR
POINTER TO MEM. 12288
50 POKE53270,PEEK(53270)OR16
51 POKE53281,0:REM SET BACKGROUND COLOR #0 TO BLACK
52 POKE53282,2:REM SET BACKGROUND COLOR #1 TO RED
53 POKE53283,7:REM SET BACKGROUND COLOR #2 TO
YELLOW
60 FORCHAR=60T063:REM PROGRAM CHARACTERS 60 THRU 63
80 FORBYTE=0T07:REM DO ALL 8 BYTES OF A CHARACTER
100 READNUMBER:REM READ 1/8TH OF CHARACTER DATA
120 POKE12288+(8*CHAR)+BYTE,NUMBER:REM STORE THE
DATA IN MEMORY
140 NEXTBYTE,CHAR
150 SHIFT CLR/HOME
PRINT"0"TAB(255)CHR$(60)CHR$(61)TAB(55)CHR$(62)CHR$(63)
160 REM LINE 150 PUTS THE NEWLY DEFINED CHARACTERS
ON THE SCREEN
170 GETA$:REM WAIT FOR USER TO PRESS A KEY
180 IFA$=""THEN170:REM IF NO KEYS WERE PRESSED,
TRY AGAIN
190 POKE53272,21:POKE53270,PEEK(53270)AND239:REM
RETURN TO NORMAL CHARACTERS
200 DATA129,37,21,29,93,85,85,85:REM DATA FOR
CHARACTER 60
210 DATA66,72,84,116,117,85,85,85:REM DATA FOR
CHARACTER 61
220 DATA37,67,85,21,6,6,40,0:REM DATA FOR
CHARACTER 62
230 DATA213,213,85,64,32,32,40,0:REM DATA FOR
CHARACTER 63
240 END
```

## EXTENDED BACKGROUND COLOR MODE

Extended background color mode gives you control over the background color of each individual character, as well as over the foreground color. For example, in this mode you could display a blue character with a yellow background on a white screen.

There are 4 registers available for extended background color mode. Each of the registers can be set to any of the 16 colors.

Color memory is used to hold the foreground color in extended background mode. It is used the same as in standard character mode.

Extended character mode places a limit on the number of different characters you can display, however. When extended color mode is on, only the first 64 characters in the character ROM (or the first 64 characters in your programmable character set) can be used. This is because two of the bits of the character code are used to select the background color. It might work something like this:

The character code (the number you would POKE to the screen) of the letter "A" is a 1. When extended color mode is on, if you POKEd a 1 to the screen, an "A" would appear. If you POKEd a 65 to the screen normally, you would expect the character with character code (CHRS) 129 to appear, which is a reversed "A." This does NOT happen in extended color mode. Instead you get the same unreversed "A" as before, but on a different background color. The following chart gives the codes:

CHARACTER CODE			BACKGROUND COLOR REGISTER	
RANGE	BIT 7	BIT 6	NUMBER	ADDRESS
0-63	0	0	0	53281 (\$D021)
64-127	0	1	1	53282 (\$D022)
128-191	1	0	2	53283 (\$D023)
192-255	1	1	3	53284 (\$D024)

Extended color mode is turned ON by setting bit 6 of the VIC-II register to a 1 at location 53265 (\$D011 in HEX). The following POKE does it:

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

Extended color mode is turned OFF by setting bit 6 of the VIC-II register to a 0 at location 53265 (\$D011). The following statement will do this:

```
POKE 53265, PEEK(53265)AND 191
```

## BIT MAPPED GRAPHICS

When writing games, plotting charts for business applications, or other types of programs, sooner or later you get to the point where you want high-resolution displays.

The Commodore 64 has been designed to do just that: high resolution is available through bit mapping of the screen. Bit mapping is the method in which each possible dot (pixel) of resolution on the screen is assigned its own bit (location) in memory. If that memory bit is a one, the dot it is assigned to is on. If the bit is set to zero, the dot is off.

High-resolution graphic design has a couple of drawbacks, which is why it is not used all the time. First of all, it takes lots of memory to bit map the entire screen. This is because every pixel must have a memory bit to control it. You are going to need one bit of memory for each pixel (or one byte for 8 pixels). Since each character is 8 by 8, and there are 40 lines with 25 characters in each line, the resolution is 320 pixels (dots) by 200 pixels for the whole screen. That gives you 64000 separate dots, each of which requires a bit in memory. In other words, 8000 bytes of memory are needed to map the whole screen.

Generally, high-resolution operations are made of many short, simple, repetitive routines. Unfortunately, this kind of thing is usually rather slow if you are trying to write high-resolution routines in BASIC. However, short, simple, repetitive routines are exactly what machine language does best. The solution is to either write your programs entirely in machine language, or call machine language, high-resolution sub-routines from your BASIC program using the SYS command from BASIC. That way you get both the ease of writing in BASIC, and the speed of machine language for graphics. The VSP cartridge is also available to add high-resolution commands to COMMODORE 64 BASIC.

All of the examples given in this section will be in BASIC to make them clear. Now to the technical details.

**BIT MAPPING** is one of the most popular graphics techniques in the computer world. It is used to create highly detailed pictures. Basically, when the Commodore 64 goes into bit map mode, it directly displays an

8K section of memory on the TV screen. When in bit map mode, you can *directly* control whether an individual dot on the screen is on or off.

There are two types of bit mapping available on the Commodore 64. They are:

- 1) Standard (high-resolution) bit mapped mode (320-dot by 200-dot resolution)
- 2) Multi-color bit mapped mode (160-dot by 200-dot resolution)

Each is very similar to the character type it is named for: standard has greater resolution, but fewer color selections. On the other hand, multi-color bit mapping trades horizontal resolution for a greater number of colors in an 8-dot by 8-dot square.

## STANDARD HIGH-RESOLUTION BIT MAP MODE

Standard bit map mode gives you a 320 horizontal dot by 200 vertical dot resolution, with a choice of 2 colors in each 8-dot by 8-dot section. Bit map mode is selected (turned ON) by setting bit 5 of the VIC-II control register to a 1 at location 53265 (\$D011 in HEX). The following POKE will do this:

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

Bit map mode is turned OFF by setting bit 5 of the VIC-II control register to 0 at location 53265 (\$D011), like this:

```
POKE 53265,PEEK(53265)AND 223
```

Before we get into the details of the bit map mode, there is one more issue to tackle, and that is where to locate the bit map area.

## HOW IT WORKS

If you remember the PROGRAMMABLE CHARACTERS section you will recall that you were able to set the bit pattern of a character stored in RAM to almost anything you wanted. If at the same time you change the character that is displayed on the screen, you would be able to change a single dot, and watch it happen. This is the basis of bit-mapping. The

entire screen is filled with programmable characters, and you make your changes directly into the memory that the programmable characters get their patterns from.

Each of the locations in screen memory that were used to control what character was displayed, are now used for color information. For example, instead of POKEing a 1 in location 1024 to make an "A" appear in the top left hand corner of the screen, location 1024 now controls the colors of the bits in that top left space.

Colors of squares in bit map mode do not come from color memory, as they do in the character modes. Instead, colors are taken from screen memory. The upper 4 bits of screen memory become the color of any bit that is set to 1 in the 8 by 8 area controlled by that screen memory location. The lower 4 bits become the color of any bit that is set to a 0.

**EXAMPLE:** Type the following:

```
5 BASE=2*4096:POKE53272,PEEK(53272)OR0:REM PUT BIT  
MAP AT 8192  
10 POKE53265,PEEK(53265)OR32:REM ENTER BIT MAP MODE
```

Now RUN the program.

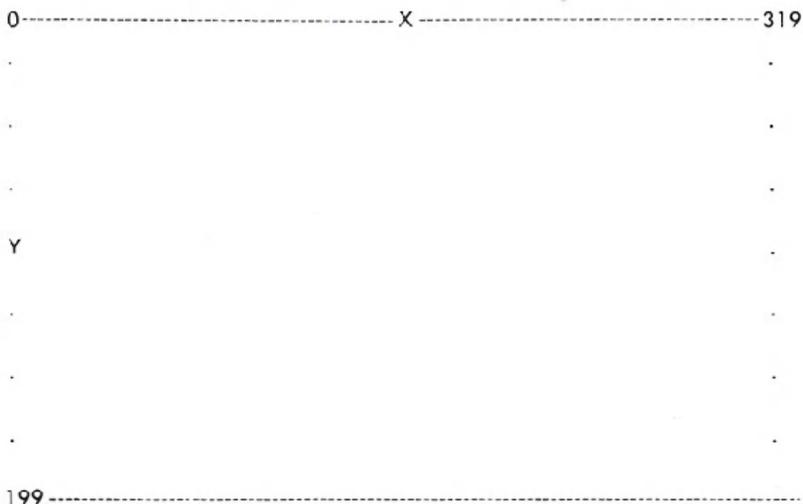
Garbage appears on the screen, right? Just like the normal screen mode, you have to clear the HIGH-RESOLUTION (HI-RES) screen before you use it. Unfortunately, printing a CLR won't work in this case. Instead you have to clear out the section of memory that you're using for your programmable characters. Hit the **RUN/STOP** and **RESTORE** keys, then add the following lines to your program to clear the HI-RES screen:

```
20 FORI=BASE+7999:POKEI,0:NEXT:REM CLEAR BIT  
MAP  
30 FORI=1024+02023:POKEI,3:NEXT:REM SET COLOR TO  
CYAN AND BLACK
```

Now RUN the program again. You should see the screen clearing, then the greenish blue color, cyan, should cover the whole screen. What we want to do now is to turn the dots on and off on the HI-RES screen.

To SET a dot (turn a dot ON) or UNSET a dot (turn a dot OFF) you must know how to find the correct bit in the character memory that you have to set to a 1. In other words, you have to find the character you need to change, the row of the character, and which bit of the row that you have to change. You need a formula to calculate this.

We will use X and Y to stand for the horizontal and vertical positions of a dot. The dot where  $X=0$  and  $Y=0$  is at the upper-left of the display. Dots to the right have higher X values, and the dots toward the bottom have higher Y values. The best way to use bit mapping is to arrange the bit map display something like this:



Each dot will have an X and a Y coordinate. With this format it is easy to control any dot on the screen.

However, what you actually have is something like this:

TOP LINE (ROW 0)	-----	BYTE 0	BYTE 8	BYTE 16	BYTE 24	.....	BYTE 312	
		BYTE 1	BYTE 9	.	.		BYTE 313	
		BYTE 2	BYTE 10	.	.		BYTE 314	
		BYTE 3	BYTE 11	.	.		BYTE 315	
		BYTE 4	BYTE 12	.	.		BYTE 316	
		BYTE 5	BYTE 13	.	.		BYTE 317	
		BYTE 6	BYTE 14	.	.		BYTE 318	
		-----	BYTE 7	BYTE 15	.	.		BYTE 319
SECOND LINE (ROW 1)	-----	BYTE 320	BYTE 328	BYTE 336	BYTE 344	.....	BYTE 632	
		BYTE 321	BYTE 329	.	.		BYTE 633	
		BYTE 322	BYTE 330	.	.		BYTE 634	
		BYTE 323	BYTE 331	.	.		BYTE 635	
		BYTE 324	BYTE 332	.	.		BYTE 636	
		BYTE 325	BYTE 333	.	.		BYTE 637	
		BYTE 326	BYTE 334	.	.		BYTE 638	
		-----	BYTE 327	BYTE 335	.	.		BYTE 639

The programmable characters which make up the bit map are arranged in 25 rows of 40 columns each. While this is a good method of organization for text, it makes bit mapping somewhat difficult. (There is a good reason for this method. See the section on MIXED MODES.)

The following formula will make it easier to control a dot on the bit map screen:

The start of the display memory area is known as the BASE. The row number (from 0 to 24) of your dot is:

$$\text{ROW} = \text{INT}(Y/8) \text{ (There are 320 bytes per line.)}$$

The character position on that line (from 0 to 39) is:

$$\text{CHAR} = \text{INT}(X/8) \text{ (There are 8 bytes per character.)}$$

The line of that character position (from 0 to 7) is:

$$\text{LINE} = Y \text{ AND } 7$$

The bit of that byte is:

$$\text{BIT} = 7 - (\text{X AND } 7)$$

Now we put these formulas together. The byte in which character memory dot (X,Y) is located is calculated by:

$$\text{BYTE} = \text{BASE} + \text{ROW} * 320 + \text{CHAR} * 8 + \text{LINE}$$

To turn on any bit on the grid with coordinates (X,Y), use this line:

$$\text{POKE BYTE, PEEK(BYTE) OR } 2^{\uparrow}\text{BIT}$$

Let's add these calculations to the program. In the following example, the COMMODORE 64 will plot a sine curve:

```
50 FORX=0TO319STEP.5:REM WAVE WILL FILL THE SCREEN
60 Y=INT(90+80*%SIN(X/10))
70 CH=INT(X/8)
80 RO=INT(Y/8)
85 LN=YAND7
90 BY=BASE+RO*320+8*CH+LN
100 BI=7-(XAND7)
110 POKEBY,PEEK(BY)OR(2^BI)
120 NEXTX
125 POKE1024,16
130 GOTO130
```

The calculation in line 60 will change the values for the sine function from a range of +1 to -1 to a range of 10 to 170. Lines 70 to 100 calculate the character, row, byte, and bit being affected, using the formulae as shown above. Line 125 signals the program is finished by changing the color of the top left corner of the screen. Line 130 freezes the program by putting it into an infinite loop. When you have finished looking at the display, just hold down **RUN/STOP** and hit **RESTORE**.

As a further example, you can modify the sine curve program to display a semicircle. Here are the lines to type to make the changes:

```
50 FORX=0TO160:REM DO HALF THE SCREEN
55 Y1=100+SQR(160*X-XX)
56 Y2=100-SQR(160*X-XX)
60 FORY=Y1TOY2STEPY1-Y2
70 CH=INT(X/8)
80 RO=INT(Y/8)
95 LN=YAND7
90 BY=BASE+RO*320-8*CH+LN
100 BI=7-(XAND7)
110 POKEBY,PEEK(BY)OR(21BI)
114 NEXT
```

This will create a semicircle in the HI-RES area of the screen.

**WARNING:** BASIC variables can overlay your high-resolution screen. If you need more memory space you must move the bottom of BASIC above the high-resolution screen area. Or, you must move your high-resolution screen area. This problem will NOT occur in machine language. It ONLY happens when you're writing programs in BASIC.

## MULTI-COLOR BIT MAP MODE

Like multi-color mode characters, multi-color bit map mode allows you to display up to four different colors in each 8 by 8 section of bit map. And as in multi-character mode, there is a sacrifice of horizontal resolution (from 320 dots to 160 dots).

Multi-color bit map mode uses an 8K section of memory for the bit map. You select your colors for multi-color bit map mode from (1) the background color register 0, (the screen background color), (2) the video matrix (the upper 4 bits give one possible color, the lower 4 bits another), and (3) color memory.

Multi-color bit mapped mode is turned ON by setting bit 5 of 53265 (\$D011) and bit 4 at location 53270 (\$D016) to a 1. The following POKE does this:

```
POKE 53265,PEEK(53625)OR 32: POKE 53270,PEEK(53270)OR 16
```

Multi-color bit mapped mode is turned OFF by setting bit 5 of 53265 (\$D011) and bit 4 at location 53270 (\$D016) to a 0. The following POKE does this:

```
POKE 53265,PEEK(53265)AND 223: POKE 53270,PEEK(53270)AND 239
```

As in standard (HI-RES) bit mapped mode, there is a one to one correspondence between the 8K section of memory being used for the display, and what is shown on the screen. However, the horizontal dots are two bits wide. Each 2 bits in the display memory area form a dot, which can have one of 4 colors.

BITS	COLOR INFORMATION COMES FROM
00	Background color #0 (screen color)
01	Upper 4 bits of screen memory
10	Lower 4 bits of screen memory
11	Color nybble (nybble = 1/2 byte = 4 bits)

## SMOOTH SCROLLING

The VIC-II chip supports smooth scrolling in both the horizontal and vertical directions. Smooth scrolling is a one pixel movement of the entire screen in one direction. It can move either up, or down, or left, or right. It is used to move new information smoothly onto the screen, while smoothly removing characters from the other side.

While the VIC-II chip does much of the task for you, the actual scrolling must be done by a machine language program. The VIC-II chip features the ability to place the video screen in any of 8 horizontal positions, and 8 vertical positions. Positioning is controlled by the VIC-II scrolling registers. The VIC-II chip also has a 38 column mode, and a 24 row mode. The smaller screen sizes are used to give you a place for your new data to scroll on from.

The following are the steps for SMOOTH SCROLLING:

- 1) Shrink the screen (the border will expand).
- 2) Set the scrolling register to maximum (or minimum value depending upon the direction of your scroll).
- 3) Place the new data on the proper (covered) portion of the screen.
- 4) Increment (or decrement) the scrolling register until it reaches the maximum (or minimum) value.
- 5) At this point, use your machine language routine to shift the entire screen one entire character in the direction of the scroll.
- 6) Go back to step 2.

To go into 38 column mode, bit 3 of location 53270 (\$D016) must be set to a 0. The following POKE does this:

```
POKE 53270,PEEK(53270)AND 247
```

To return to 40 column mode, set bit 3 of location 53270 (\$D016) to a 1. The following POKE does this:

```
POKE 53270,PEEK(53270)OR 8
```

To go into 24 row mode, bit 3 of location 53265 (\$D011) must be set to a 0. The following POKE will do this:

```
POKE 53265,PEEK(53265)AND 247
```

To return to 25 row mode, set bit 3 of location 53265 (\$D011) to a 1. The following POKE does this:

```
POKE 53265,PEEK(53265)OR 8
```

When scrolling in the X direction, it is necessary to place the VIC-II chip into 38 column mode. This gives new data a place to scroll from. When scrolling LEFT, the new data should be placed on the right. When scrolling RIGHT the new data should be placed on the left. Please note that there are still 40 columns to screen memory, but only 38 are visible.

When scrolling in the Y direction, it is necessary to place the VIC-II chip into 24 row mode. When scrolling UP, place the new data in the LAST row. When scrolling DOWN, place the new data on the FIRST row. Unlike X scrolling, where there are covered areas on each side of the screen, there is only one covered area in Y scrolling. When the Y scroll-

ing register is set to 0, the first line is covered, ready for new data. When the Y scrolling register is set to 7 the last row is covered.

For scrolling in the X direction, the scroll register is located in bits 2 to 0 of the VIC-II control register at location 53270 (\$D016 in HEX). As always, it is important to affect only those bits. The following POKE does this:

```
POKE 53270, (PEEK(53270)AND 248)+X
```

where X is the X position of the screen from 0 to 7.

For scrolling in the Y direction, the scroll register is located in bits 2 to 0 of the VIC-II control register at location 53265 (\$D011 in HEX). As always, it is important to affect only those bits. The following POKE does this:

```
POKE 53265, (PEEK(53265)AND 248)+Y
```

where Y is the Y position of the screen from 0 to 7.

To scroll text onto the screen from the bottom, you would step the low-order 3 bits of location 53265 from 0-7, put more data on the covered line at the bottom of the screen, and then repeat the process. To scroll characters onto the screen from left to right, you would step the low-order 3 bits of location 53270 from 0 to 7, print or POKE another column of new data into column 0 of the screen, then repeat the process.

If you step the scroll bits by -1, your text will move in the opposite direction.

#### EXAMPLE: Text scrolling onto the bottom of the screen:

```
10 POKE53265,PEEK(53265)AND247          :REM GO
INTO 24 ROW MODE
20 PRINTCHR$(147)                          :REM
CLEAR THE SCREEN
30 FORX=1TO24:PRINTCHR$(17):NEXT          :REM MOVE
THE CURSOR TO THE BOTTOM
40 POKE53265,(PEEK(53265)AND248)+7 PRINT :REM
POSITION FOR 1ST SCROLL
50 PRINT"      HELLO";
60 FORP=6TO0STEP-1
70 POKE53265,(PEEK(53265)AND248)+P
80 FORX=1TO50:NEXT                          :REM
DELAY LOOP
90 NEXT:GOTO40
```

## SPRITES

A **SPRITE** is a special type of user definable character which can be displayed anywhere on the screen. Sprites are maintained directly by the VIC-II chip. And all you have to do is tell a sprite "what to look like," "what color to be," and "where to appear." The VIC-II chip will do the rest! Sprites can be any of the 16 colors available.

Sprites can be used with ANY of the other graphics modes, bit mapped, character, multi-color, etc., and they'll keep their shape in all of them. The sprite carries its own color definition, its own mode (HI-RES or multi-colored), and its own shape.

Up to 8 sprites at a time can be maintained by the VIC-II chip automatically. More sprites can be displayed using RASTER INTERRUPT techniques.

The features of **SPRITES** include:

- 1) 24 horizontal dot by 21 vertical dot size.
- 2) Individual color control for each sprite.
- 3) Sprite multi-color mode.
- 4) Magnification (2X) in horizontal, vertical, or both directions.
- 5) Selectable sprite to background priority.
- 6) Fixed sprite to sprite priorities.
- 7) Sprite to sprite collision detection.
- 8) Sprite to background collision detection.

These special sprite abilities make it simple to program many arcade style games. Because the sprites are maintained by hardware, it is even possible to write a good quality game in BASIC!

There are 8 sprites supported directly by the VIC-II chip. They are numbered from 0 to 7. Each of the sprites has its own definition location, position registers and color register, and has its own bits for enable and collision detection.

## DEFINING A SPRITE

Sprites are defined like programmable characters are defined. However, since the size of the sprite is larger, more bytes are needed. A sprite is 24 by 21 dots, or 504 dots. This works out to 63 bytes (504/8

COLUMN NUMBER	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23
BIT	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
BIT DATA VALUES (ON = 1 x VAL)	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
ROW 0																								
ROW 1																								
ROW 2																								
ROW 3																								
ROW 4																								
ROW 5																								
ROW 6																								
ROW 7																								
ROW 8																								
ROW 9																								
ROW 10																								
ROW 11																								
ROW 12																								
ROW 13																								
ROW 14																								
ROW 15																								
ROW 16																								
ROW 17																								
ROW 18																								
ROW 19																								
ROW 20																								

Figure 3-2. Sprite Definition Block.

bits) needed to define a sprite. The 63 bytes are arranged in 21 rows of 3 bytes each. A sprite definition looks like this:

BYTE 0	BYTE 1	BYTE 2
BYTE 3	BYTE 4	BYTE 5
BYTE 6	BYTE 7	BYTE 8
..	..	..
..	..	..
..	..	..
BYTE 60	BYTE 61	BYTE 62

Another way to view how a sprite is created is to take a look at the sprite definition block on the bit level. It would look something like Figure 3-2.

In a standard (HI-RES) sprite, each bit set to 1 is displayed in that sprite's foreground color. Each bit set to 0 is transparent and will display whatever data is behind it. This is similar to a standard character.

Multi-color sprites are similar to multi-color characters. Horizontal resolution is traded for extra color resolution. The resolution of the sprite becomes 12 horizontal dots by 21 vertical dots. Each dot in the sprite becomes twice as wide, but the number of colors displayable in the sprite is increased to 4.

## SPRITE POINTERS

Even though each sprite takes only 63 bytes to define, one more byte is needed as a place holder at the end of each sprite. Each sprite, then, takes up 64 bytes. This makes it easy to calculate where in memory your sprite definition is, since 64 bytes is an even number and in binary it's an even power.

Each of the 8 sprites has a byte associated with it called the **SPRITE POINTER**. The sprite pointers control where each sprite definition is located in memory. These 8 bytes are always located as the last 8 bytes of the 1K chunk of screen memory. Normally, on the Commodore 64, this means they begin at location 2040 (\$07F8 in HEX). However, if you move the screen, the location of your sprite pointers will also move.

Each sprite pointer can hold a number from 0 to 255. This number points to the definition for that sprite. Since each sprite definition takes 64 bytes, that means that the pointer can "see" anywhere in the 16K block of memory that the VIC-II chip can access (since  $256 * 64 = 16K$ ).

If sprite pointer #0, at location 2040, contains the number 14, for example, this means that sprite 0 will be displayed using the 64 bytes beginning at location  $14 * 64 = 896$  which is in the cassette buffer. The following formula makes this clear:

$$\text{LOCATION} = (\text{BANK} * 16384) + (\text{SPRITE POINTER VALUE} * 64)$$

Where BANK is the 16K segment of memory that the VIC-II chip is looking at and is from 0 to 3.

The above formula gives the start of the 64 bytes of the sprite definition block.

When the VIC-II chip is looking at BANK 0 or BANK 2, there is a ROM IMAGE of the character set present in certain locations, as mentioned before. Sprite definitions can NOT be placed there. If for some reason you need more than 128 different sprite definitions, you should use one of the banks without the ROM IMAGE, 1 or 3.

## TURNING SPRITES ON

The VIC-II control register at location 53269 (\$D015 in HEX) is known as the **SPRITE ENABLE** register. Each of the sprites has a bit in this register which controls whether that sprite is ON or OFF. The register looks like this:

\$D015 7 6 5 4 3 2 1 0

To turn on sprite 1, for example, it is necessary to turn that bit to a 1. The following POKE does this:

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

A more general statement would be the following:

```
POKE 53269,PEEK(53269)OR (2↑SN)
```

where SN is the sprite number, from 0 to 7.

**NOTE:** A sprite must be turned ON before it can be seen.

## TURNING SPRITES OFF

A sprite is turned off by setting its bit in the VIC-II control register at 53269 (\$D015 in HEX) to a 0. The following POKE will do this:

```
POKE 53269, PEEK(53269)AND (255-2↑SN)
```

where SN is the sprite number from 0 to 7.

## COLORS

A sprite can be any of the 16 colors generated by the VIC-II chip. Each of the sprites has its own sprite color register. These are the memory locations of the color registers:

ADDRESS		DESCRIPTION
53287	(\$D027)	SPRITE 0 COLOR REGISTER
53288	(\$D028)	SPRITE 1 COLOR REGISTER
53289	(\$D029)	SPRITE 2 COLOR REGISTER
53290	(\$D02A)	SPRITE 3 COLOR REGISTER
53291	(\$D02B)	SPRITE 4 COLOR REGISTER
53292	(\$D02C)	SPRITE 5 COLOR REGISTER
53293	(\$D02D)	SPRITE 6 COLOR REGISTER
53294	(\$D02E)	SPRITE 7 COLOR REGISTER

All dots in the sprite will be displayed in the color contained in the sprite color register. The rest of the sprite will be transparent, and will show whatever is behind the sprite.

## MULTI-COLOR MODE

Multi-color mode allows you to have up to 4 different colors in each sprite. However, just like other multi-color modes, horizontal resolution is cut in half. In other words, when you're working with sprite multi-color mode (like in multi-color character mode), instead of 24 dots across the sprite, there are 12 pairs of dots. Each pair of dots is called a BIT PAIR. Think of each bit pair (pair of dots) as a single dot in your overall sprite when it comes to choosing colors for the dots in your sprites. The table

below gives you the bit pair values needed to turn ON each of the four colors you've chosen for your sprite:

BIT PAIR	DESCRIPTION
00	TRANSPARENT, SCREEN COLOR
01	SPRITE MULTI-COLOR REGISTER #0 (53285) (\$D025)
10	SPRITE COLOR REGISTER
11	SPRITE MULTI-COLOR REGISTER #1 (53286) (\$D026)

**NOTE:** The sprite foreground color is a 10. The character foreground is a 11.

## SETTING A SPRITE TO MULTI-COLOR MODE

To switch a sprite into multi-color mode you must turn ON the VIC-II control register at location 53276 (\$D01C). The following POKE does this:

```
POKE 53276,PEEK(53276) OR (2↑SN)
```

where SN is the sprite number (0 to 7).

To switch a sprite out of multi-color mode you must turn OFF the VIC-II control register at location 53276 (\$D01C). The following POKE does this:

```
POKE 53276,PEEK(53276) AND (255-2↑SN)
```

where SN is the sprite number (0 to 7).

## EXPANDED SPRITES

The VIC-II chip has the ability to expand a sprite in the vertical direction, the horizontal direction, or both at once. When expanded, each dot in the sprite is twice as wide or twice as tall. Resolution doesn't actually increase . . . the sprite just gets bigger.

To expand a sprite in the horizontal direction, the corresponding bit in the VIC-II control register at location 53277 (\$D01D in HEX) must be turned ON (set to a 1). The following POKE expands a sprite in the X direction:

```
POKE 53277,PEEK(53277)OR (2↑SN)
```

where SN is the sprite number from 0 to 7.

To unexpand a sprite in the horizontal direction, the corresponding bit in the VIC-II control register at location 53277 (\$D01D in HEX) must be turned OFF (set to a 0). The following POKE "unexpands" a sprite in the X direction:

**POKE 53277,PEEK(53277)AND (255-2↑SN)**

where SN is the sprite number from 0 to 7.

To expand a sprite in the vertical direction, the corresponding bit in the VIC-II control register at location 53271 (\$D017 in HEX) must be turned ON (set to a 1). The following POKE expands a sprite in the Y direction:

**POKE 53271,PEEK(53271)OR (2↑SN)**

where SN is the sprite number from 0 to 7.

To unexpand a sprite in the vertical direction, the corresponding bit in the VIC-II control register at location 53271 (\$D017 in HEX) must be turned OFF (set to a 0). The following POKE "unexpands" a sprite in the Y direction:

**POKE 53271,PEEK(53271)AND (255-2↑SN)**

where SN is the sprite number from 0 to 7.

## SPRITE POSITIONING

Once you've made a sprite you want to be able to move it around the screen. To do this, your Commodore 64 uses three positioning registers:

- 1) SPRITE X POSITION REGISTER
- 2) SPRITE Y POSITION REGISTER
- 3) MOST SIGNIFICANT BIT X POSITION REGISTER

Each sprite has an X position register, a Y position register, and a bit in the X most significant bit register. This lets you position your sprites very accurately. You can place your sprite in 512 possible X positions and 256 possible Y positions.

The X and Y position registers work together, in pairs, as a team. The locations of the X and Y registers appear in the memory map as follows: First is the X register for sprite 0, then the Y register for sprite 0. Next

comes the X register for sprite 1, the Y register for sprite 1, and so on. After all 16 X and Y registers comes the most significant bit in the X position (X MSB) located in its own register.

The chart below lists the locations of each sprite position register. You use the locations at their appropriate time through POKE statements:

LOCATION		DESCRIPTION
DECIMAL	HEX	
53248	(\$D000)	SPRITE 0 X POSITION REGISTER
53249	(\$D001)	SPRITE 0 Y POSITION REGISTER
53250	(\$D002)	SPRITE 1 X POSITION REGISTER
53251	(\$D003)	SPRITE 1 Y POSITION REGISTER
53252	(\$D004)	SPRITE 2 X POSITION REGISTER
53253	(\$D005)	SPRITE 2 Y POSITION REGISTER
53254	(\$D006)	SPRITE 3 X POSITION REGISTER
53255	(\$D007)	SPRITE 3 Y POSITION REGISTER
53256	(\$D008)	SPRITE 4 X POSITION REGISTER
53257	(\$D009)	SPRITE 4 Y POSITION REGISTER
53258	(\$D00A)	SPRITE 5 X POSITION REGISTER
53259	(\$D00B)	SPRITE 5 Y POSITION REGISTER
53260	(\$D00C)	SPRITE 6 X POSITION REGISTER
53261	(\$D00D)	SPRITE 6 Y POSITION REGISTER
53262	(\$D00E)	SPRITE 7 X POSITION REGISTER
53263	(\$D00F)	SPRITE 7 Y POSITION REGISTER
53264	(\$D010)	SPRITE X MSB REGISTER

The position of a sprite is calculated from the TOP LEFT corner of the 24 dot by 21 dot area that your sprite can be designed in. It does NOT matter how many or how few dots you use to make up a sprite. Even if only one dot is used as a sprite, and you happen to want it in the middle of the screen, you must still calculate the exact positioning by starting at the top left corner location.

## VERTICAL POSITIONING

Setting up positions in the horizontal direction is a little more difficult than vertical positioning, so we'll discuss vertical (Y) positioning first.

There are 200 different dot positions that can be individually programmed onto your TV screen in the Y direction. The sprite Y position registers can handle numbers up to 255. This means that you have more

than enough register locations to handle moving a sprite up and down. You also want to be able to smoothly move a sprite on and off the screen. More than 200 values are needed for this.

The first on-screen value from the top of the screen, and in the Y direction for an unexpanded sprite is 30. For a sprite expanded in the Y direction it would be 9. (Since each dot is twice as tall, this makes a certain amount of sense, as the initial position is STILL calculated from the top left corner of the sprite.)

The first Y value in which a sprite (expanded or not) is fully on the screen (all 21 possible lines displayed) is 50.

The last Y value in which an unexpanded sprite is fully on the screen is 229. The last Y value in which an expanded sprite is fully on the screen is 208.

The first Y value in which a sprite is fully off the screen is 250.

### EXAMPLE:

```
10 PRINT" " SHIFT CLR/HOME          :REM CLEAR SCREEN
20 POKE2040,13          :REM GET SPRITE 0
DATA FROM BLOCK 13
30 FORI=0TO62:POKE832+I,129:NEXT:REM POKE SPRITE
DATA INTO BLOCK 13 (13*64=832)
40 Y=53248              :REM SET BEGINNING
OF VIDEO CHIP
50 POKEV+21,1          :REM ENABLE SPRITE
1
60 POKEV+39,1          :REM SET SPRITE 0
COLOR
70 POKEV+1,100         :REM SET SPRITE 0
Y POSITION
80 POKEV+16,0:POKEV,100 :REM SET SPRITE 0
X POSITION
```

### HORIZONTAL POSITIONING

Positioning in the horizontal direction is more complicated because there are more than 256 positions. This means that an extra bit, or 9th bit is used to control the X position. By adding the extra bit when necessary a sprite now has 512 possible positions in the left/right, X, direction. This makes more possible combinations than can be seen on the visible part of the screen. Each sprite can have a position from 0 to 511. However, only those values between 24 and 343 are visible on the screen. If the X position of a sprite is greater than 255 (on the right side of the screen), the bit in the X MOST SIGNIFICANT BIT POSITION register must be set to a 1 (turned ON). If the X position of a sprite is less than

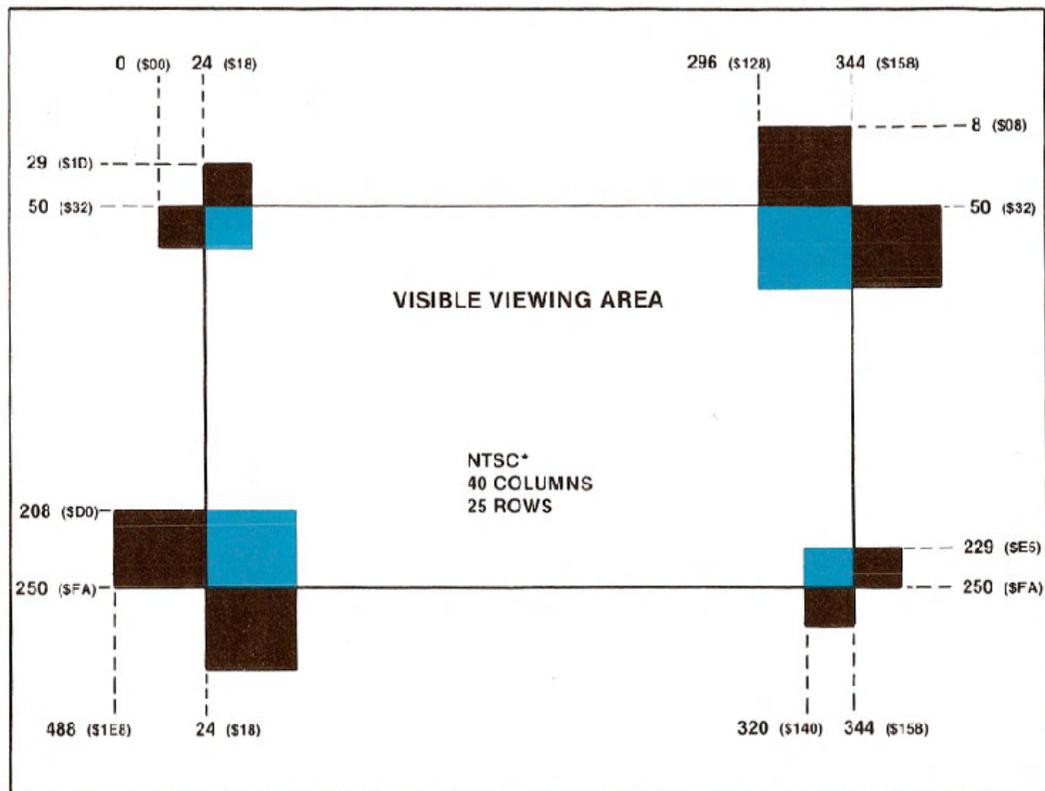
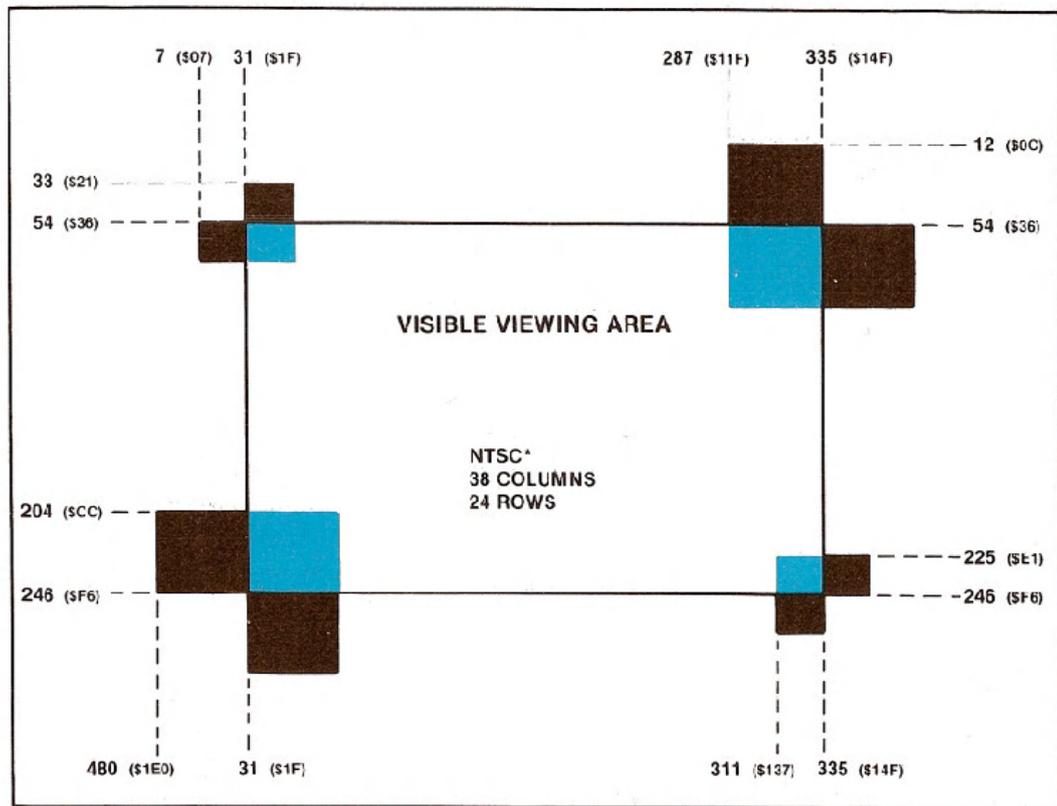


Figure 3-3. Sprite

\*North American television transmission standards for your home TV.

Positioning Charts.



\*North American television transmission standards for your home TV.

256 (on the left side of the screen), then the X MSB of that sprite must be 0 (turned OFF). Bits 0 to 7 of the X MSB register correspond to sprites 0 to 7, respectively.

The following program moves a sprite across the screen:

#### EXAMPLE:

```
10 PRINT "0"
20 POKE2040,13
30 FORI=0TO62:POKE832+I,129:NEXT
40 V=53248
50 POKEV+21,1
60 POKEV+39,1
70 POKEV+1,100
80 FORJ=0TO347
90 HX=INT(J/256):LX=J-256*HX
100 POKEV,LX:POKEV+16,HX:NEXT
```

When moving expanded sprites onto the left side of the screen in the X direction, you have to start the sprite OFF SCREEN on the RIGHT SIDE. This is because an expanded sprite is larger than the amount of space available on the left side of the screen.

#### EXAMPLE:

```
10 PRINT "0"
20 POKE2040,13
30 FORI=0TO62:POKE832+I,129:NEXT
40 V=53248
50 POKEV+21,1
60 POKEV+39,1:POKEV+23,1:POKEV+29,1
70 POKEV+1,100
80 J=488
90 HX=INT(J/256):LX=J-256*HX
100 POKEV,LX:POKEV+16,HX
110 J=J+1:IFJ>511THENJ=0
120 IFJ>488ORJ<348GOTO90
```

The charts in Figure 3-3 explain sprite positioning.

By using these values, you can position each sprite anywhere. By moving the sprite a single dot position at a time, very smooth movement is easy to achieve.

## SPRITE POSITIONING SUMMARY

Unexpanded sprites are at least partially visible in the 40 column, by 25 row mode within the following parameters:

$$1 < = X < = 343$$

$$30 < = Y < = 249$$

In the 38 column mode, the X parameters change to the following:

$$8 < = X < = 334$$

In the 24 row mode, the Y parameters change to the following:

$$34 < = Y < = 245$$

Expanded sprites are at least partially visible in the 40 column, by 25 row mode within the following parameters:

$$489 > = X < = 343$$

$$9 > = Y < = 249$$

In the 38 column mode, the X parameters change to the following:

$$496 > = X < = 334$$

In the 24 row mode, the Y parameters change to the following:

$$13 < = Y < = 245$$

## SPRITE DISPLAY PRIORITIES

Sprites have the ability to cross each other's paths, as well as cross in front of, or behind other objects on the screen. This can give you a truly three dimensional effect for games.

Sprite to sprite priority is fixed. That means that sprite 0 has the highest priority, sprite 1 has the next priority, and so on, until we get to sprite 7, which has the lowest priority. In other words, if sprite 1 and sprite 6 are positioned so that they cross each other, sprite 1 will be in front of sprite 6.

So when you're planning which sprites will appear to be in the foreground of the picture, they must be assigned lower sprite numbers than those sprites you want to put towards the back of the scene. Those sprites will be given higher sprite numbers.

**NOTE:** A "window" effect is possible. If a sprite with higher priority has "holes" in it (areas where the dots are not set to 1 and thus turned ON), the sprite with the lower priority will show through. This also happens with sprite and background data.

Sprite to background priority is controllable by the SPRITE-BACKGROUND priority register located at 53275 (\$D01B). Each sprite has a bit in this register. If that bit is 0, that sprite has a higher priority than the background on the screen. In other words, the sprite appears in front of background data. If that bit is a 1, that sprite has a lower priority than the background. Then the sprite appears behind the background data.

## COLLISION DETECTS

One of the more interesting aspects of the VIC-II chip is its collision detection abilities. Collisions can be detected between sprites, or between sprites and background data. A collision occurs when a non-zero part of a sprite overlaps a non-zero portion of another sprite or characters on the screen.

## SPRITE TO SPRITE COLLISIONS

Sprite to sprite collisions are recognized by the computer, or flagged, in the sprite to sprite collision register at location 53278 (\$D01E in HEX) in the VIC-II chip control register. Each sprite has a bit in this register. If that bit is a 1, then that sprite is involved in a collision. The bits in this register will remain set until read (PEEKed). Once read, the register is automatically cleared, so it is a good idea to save the value in a variable until you are finished with it.

**NOTE:** Collisions can take place even when the sprites are off screen.

## SPRITE TO DATA COLLISIONS

Sprite to data collisions are detected in the sprite to data collision register at location 53279 (\$D01F in HEX) of the VIC-II chip control register.

Each sprite has a bit in this register. If that bit is a 1, then that sprite is involved in a collision. The bits in this register remain set until read (PEEKed). Once read, the register is automatically cleared, so it is a good idea to save the value in a variable until you are finished with it.

**NOTE:** MULTI-COLOR data 01 is considered transparent for collisions, even though it shows up on the screen. When setting up a background screen, it is a good idea to make everything that should not cause a collision 01 in multi color mode.

```

10 REM SPRITE EXAMPLE 1...
20 REM THE HOT AIR BALLOON
30 VIC=13*4096:REM THIS IS WHERE THE VIC REGISTERS
BEGIN
35 POKEVIC+21,1:REM ENABLE SPRITE 0
36 POKEVIC+33,14:REM SET BACKGROUND COLOR TO LIGHT
BLUE
37 POKEVIC+23,1:REM EXPAND SPRITE 0 IN Y
38 POKEVIC+29,1:REM EXPAND SPRITE 0 IN X
40 POKE2040,192:REM SET SPRITE 0'S POINTER
180 POKEVIC+0,100:REM SET SPRITE 0'S X POSITION
190 POKEVIC+1,100:REM SET SPRITE 0'S Y POSITION
220 POKEVIC+39,1:REM SET SPRITE 0'S COLOR
250 FORY=0TO63:REM BYTE COUNTER WITH SPRITE LOOP
300 READA:REM READ IN A BYTE
310 POKE192*64+Y,A:REM STORE THE DATA IN SPRITE
AREA
320 NEXTY:REM CLOSE LOOP
330 DX=1:DY=1
340 X=PEEK(VIC):REM LOOK AT SPRITE 0'S X POSITION
350 Y=PEEK(VIC+1):REM LOOK AT SPRITE 0'S Y POSITION
360 IFY=50ORY=208THENDY=-DY:REM IF Y IS ON THE
EDGE OF THE...
370 REM SCREEN, THEN REVERSE DELTA Y
380 IFX=24AND(PEEK(VIC+16)AND1)=0THENDX=-DX:REM IF
SPRITE IS...
390 REM TOUCHING THE LEFT EDGE (X=24 AND THE MSB
FOR SPRITE 0 IS 0), REVERSE IT
400 IFX=40AND(PEEK(VIC+16)AND1)=1THENDX=-DX:REM IF
SPRITE IS...
410 REM TOUCHING THE RIGHT EDGE (X=40 AND THE MSB
FOR SPRITE 0 IS 1), REVERSE IT
420 IFX=255ANDDX=1THENX=-1:SIDE=1
430 REM SWITCH TO OTHER SIDE OF THE SCREEN
440 IFX=0ANDDX=-1THENX=256:SIDE=0
450 REM SWITCH TO OTHER SIDE OF THE SCREEN
460 X=X+DX:REM ADD DELTA X TO X
470 X=XAND255:REM MAKE SURE X IS IN ALLOWED RANGE
480 Y=Y+DY:REM ADD DELTA Y TO Y
485 POKEVIC+16,SIDE
490 POKEVIC,X:REM PUT NEW X VALUE INTO SPRITE 0'S
X POSITION
510 POKEVIC+1,Y:REM PUT NEW Y VALUE INTO SPRITE
0'S Y POSITION
530 GOTO340
600 REM ***** SPRITE DATA *****
610 DATA0,127,0,1,255,192,3,255,224,3,231,224
620 DATA7,217,240,7,223,240,7,217,240,3,231,224
630 DATA3,255,224,3,255,224,2,255,160,1,127,64
640 DATA1,62,64,0,156,128,0,156,128,0,73,0,0,73,0
650 DATA0,62,0,0,62,0,0,62,0,0,28,0,0

```

```

10 REM SPRITE EXAMPLE 2...
20 REM THE HOT AIR BALLOON AGAIN
30 VIC=13#4096:REM THIS IS WHERE THE VIC REGISTERS
  BEGIN
35 POKEVIC+21,63:REM ENABLE SPRITES 0 THRU 5
36 POKEVIC+33,14:REM SET BACKGROUND COLOR TO LIGHT
  BLUE
37 POKEVIC+23,3:REM EXPAND SPRITES 0 AND 1 IN Y
38 POKEVIC+29,3:REM EXPAND SPRITES 0 AND 1 IN X
40 POKE2040,192:REM SET SPRITE 0'S POINTER
50 POKE2041,193:REM SET SPRITE 1'S POINTER
60 POKE2042,192:REM SET SPRITE 2'S POINTER
70 POKE2043,193:REM SET SPRITE 3'S POINTER
80 POKE2044,192:REM SET SPRITE 4'S POINTER
90 POKE2045,193:REM SET SPRITE 5'S POINTER
100 POKEVIC+4,30:REM SET SPRITE 2'S X POSITION
110 POKEVIC+5,58:REM SET SPRITE 2'S Y POSITION
120 POKEVIC+6,65:REM SET SPRITE 3'S X POSITION
130 POKEVIC+7,58:REM SET SPRITE 3'S Y POSITION
140 POKEVIC+8,100:REM SET SPRITE 4'S X POSITION
150 POKEVIC+9,58:REM SET SPRITE 4'S Y POSITION
160 POKEVIC+10,100:REM SET SPRITE 5'S X POSITION
170 POKEVIC+11,58:REM SET SPRITE 5'S Y POSITION

175 PRINT"␣"TAB(15)"THIS IS TWO HIRES SPRITES".
  CTR: 2
  SHIFT CLR/RCME
176 PRINTTAB(55)"ON TOP OF EACH OTHER"
180 POKEVIC+0,100:REM SET SPRITE 0'S X POSITION
190 POKEVIC+1,100:REM SET SPRITE 0'S Y POSITION
200 POKEVIC+2,100:REM SET SPRITE 1'S X POSITION
210 POKEVIC+3,100:REM SET SPRITE 1'S Y POSITION
220 POKEVIC+39,1:REM SET SPRITE 0'S COLOR
230 POKEVIC+41,1:REM SET SPRITE 2'S COLOR
240 POKEVIC+43,1:REM SET SPRITE 4'S COLOR
250 POKEVIC+40,6:REM SET SPRITE 1'S COLOR
260 POKEVIC+42,6:REM SET SPRITE 3'S COLOR
270 POKEVIC+44,6:REM SET SPRITE 5'S COLOR
280 FORX=192TO193:REM THE START OF THE LOOP THAT
  DEFINES THE SPRITES
290 FORY=0TO63:REM BYTE COUNTER WITH SPRITE LOOP
300 READA:REM READ IN A BYTE
310 POKEX#64+Y,A:REM STORE THE DATA IN SPRITE AREA
320 NEXTY,X:REM CLOSE LOOPS
330 DX=1:DY=1
340 X=PEEK(VIC):REM LOOK AT SPRITE 0'S X POSITION
350 Y=PEEK(VIC+1):REM LOOK AT SPRITE 0'S Y POSITION
360 IFY=50ORY=208THENDY=-DY:REM IF Y IS ON THE
  EDGE OF THE...
370 REM SCREEN, THEN REVERSE DELTA Y
380 IFX=24AND(PEEK(VIC+16)AND1)=0THENDX=-DX:REM IF
  SPRITE IS...
390 REM TOUCHING THE LEFT EDGE, THEN REVERSE IT

```

```

400 IFX=40AND(PEEK(VIC+16)AND1)=1THENDX=-DX:REM IF
  SPRITE IS...
410 REM TOUCHING THE RIGHT EDGE, THEN REVERSE IT
420 IFX=255ANDDX=1THENX=-1:SIDE=3
430 REM SWITCH TO OTHER SIDE OF THE SCREEN
440 IFX=0ANDDX=-1THENX=256:SIDE=0
450 REM SWITCH TO OTHER SIDE OF THE SCREEN
460 X=X+DX:REM ADD DELTA X TO X
470 X=XAND255:REM MAKE SURE X IS IN ALLOWED RANGE
480 Y=Y+DY:REM ADD DELTA Y TO Y
485 POKEVIC+16,SIDE
490 POKEVIC,X:REM PUT NEW X VALUE INTO SPRITE 0'S
  X POSITION
500 POKEVIC+2,X:REM PUT NEW X VALUE INTO SPRITE
  1'S X POSITION
510 POKEVIC+1,Y:REM PUT NEW Y VALUE INTO SPRITE
  0'S Y POSITION
520 POKEVIC+3,Y:REM PUT NEW Y VALUE INTO SPRITE
  1'S Y POSITION
530 GOTO340
540 REM ***** SPRITE DATA *****
550 DATA0,255,0,3,153,192,7,24,224,7,56,224,14,126,
  112,14,126,112,14,126,112
560 DATA0,126,96,7,56,224,7,56,224,1,56,128,0,153,
  0,0,90,0,0,56,0
570 DATA0,56,0,0,0,0,0,0,0,126,0,0,42,0,0,34,0,0,
  40,0,0
580 DATA0,0,0,0,102,0,0,231,0,0,195,0,1,129,128,1,
  129,128,1,129,128
590 DATA1,129,128,0,195,0,0,195,0,4,195,32,2,102,
  64,2,36,64,1,0,128
600 DATA1,0,128,0,153,0,0,153,0,0,0,0,0,94,0,0,42,
  0,0,20,0,0

```

```

10 REM SPRITE EXAMPLE 3...
20 REM THE HOT AIR GOLF
30 VIC=53248:REM THIS IS WHERE THE VIC REGISTERS
  BEGIN
35 POKEVIC+21,1:REM ENABLE SPRITE 0

```

```

36 POKEVIC+33,14:REM SET BACKGROUND COLOR TO LIGHT
BLUE
37 POKEVIC+23,1:REM EXPAND SPRITE 0 IN Y
38 POKEVIC+29,1:REM EXPAND SPRITE 0 IN X
40 POKE2040,192:REM SET SPRITE 0'S POINTER
50 POKEVIC+28,1:REM TURN ON MULTICOLOR
60 POKEVIC+37,7:REM SET MULTICOLOR 0
70 POKEVIC+38,4:REM SET MULTICOLOR 1
180 POKEVIC+0,100:REM SET SPRITE 0'S X POSITION
190 POKEVIC+1,100:REM SET SPRITE 0'S Y POSITION
220 POKEVIC+39,2:REM SET SPRITE 0'S COLOR
290 FOR Y=0 TO 63:REM BYTE COUNTER WITH SPRITE LOOP
300 READA:REM READ IN A BYTE
310 POKE12288+Y,A:REM STORE THE DATA IN SPRITE AREA
320 NEXT Y:REM CLOSE LOOP
330 DX=1:DY=1
340 X=PEEK(VIC):REM LOOK AT SPRITE 0'S X POSITION
350 Y=PEEK(VIC+1):REM LOOK AT SPRITE 0'S Y POSITION
360 IF Y=50 OR Y=20 THEN DY=-DY:REM IF Y IS ON THE
EDGE OF THE...
370 REM SCREEN, THEN REVERSE DELTA Y
380 IF X=24 AND (PEEK(VIC+16) AND 1)=0 THEN DX=-DX:REM
IF SPRITE IS...
390 REM TOUCHING THE LEFT EDGE, THEN REVERSE IT
400 IF X=40 AND (PEEK(VIC+16) AND 1)=1 THEN DX=-DX:REM IF
SPRITE IS...
410 REM TOUCHING THE RIGHT EDGE, THEN REVERSE IT
420 IF X=255 AND DX=1 THEN X=-1:SIDE=1
430 REM SWITCH TO OTHER SIDE OF THE SCREEN
440 IF X=0 AND DX=-1 THEN X=255:SIDE=0
450 REM SWITCH TO OTHER SIDE OF THE SCREEN
460 X=X+DX:REM ADD DELTA X TO X
470 X=X AND 255:REM MAKE SURE X IS IN ALLOWED RANGE
480 Y=Y+DY:REM ADD DELTA Y TO Y
485 POKEVIC+16,SIDE
490 POKEVIC,X:REM PUT NEW X VALUE INTO SPRITE 0'S
X POSITION
510 POKEVIC+1,Y:REM PUT NEW Y VALUE INTO SPRITE
0'S Y POSITION
520 GETA#:REM GET A KEY FROM THE KEYBOARD
521 IF A#="M" THEN POKEVIC+28,1:REM USER SELECTED
MULTICOLOR
522 IF A#="H" THEN POKEVIC+28,0:REM USER SELECTED
HIGH RESOLUTION
530 GOTO 340
600 REM ***** SPRITE DATA *****
610 DATA 54,0,1,16,170,4,6,170,144,10,170,160,42,
170,168,41,105,104,169,235,105
620 DATA 169,235,106,169,235,106,170,170,170,170,
170,170,170,170,170,170,170,170
630 DATA 166,170,154,169,85,106,170,85,170,42,170,
168,10,170,160,1,0,64,1,0,64
640 DATA 5,0,80,0

```

## OTHER GRAPHICS FEATURES

### SCREEN BLANKING

Bit 4 of the VIC-II control register controls the screen blanking function. It is found in the control register at location 53265 (\$D011). When it is turned ON (in other words, set to a 1) the screen is normal. When bit 4 is set to 0 (turned OFF), the entire screen changes to border color.

The following POKE blanks the screen. No data is lost, it just isn't displayed.

```
POKE 53265,PEEK(53265)AND 239
```

To bring back the screen, use the POKE shown below:

```
POKE 53265,PEEK(53265)OR 16
```

**NOTE:** Turning off the screen will speed up the processor slightly. This means that program RUNNING is also sped up.

### RASTER REGISTER

The raster register is found in the VIC-II chip at location 53266 (\$D012). The raster register is a dual purpose register. When you read this register it returns the lower 8 bits of the current raster position. The raster position of the most significant bit is in register location 53265 (\$D011). You use the raster register to set up timing changes in your display so that you can get rid of screen flicker. The changes on your screen should be made when the raster is not in the visible display area, which is when your dot positions fall between 51 and 251.

When the raster register is written to (including the MSB) the number written to is saved for use with the raster compare function. When the actual raster value becomes the same as the number written to the raster register, a bit in the VIC-II chip interrupt register 53273 (\$D019) is turned ON by setting it to 1.

**NOTE:** If the proper interrupt bit is enabled (turned on), an interrupt (IRQ) will occur.

## INTERRUPT STATUS REGISTER

The interrupt status register shows the current status of any interrupt source. The current status of bit 2 of the interrupt register will be a 1 when two sprites hit each other. The same is true, in a corresponding 1 to 1 relationship, for bits 0–3 listed in the chart below. Bit 7 is also set with a 1, whenever an interrupt occurs.

The interrupt status register is located at 53273 (\$D019) and is as follows:

LATCH	BIT #	DESCRIPTION
IRST	0	Set when current raster count = stored raster count
IMDC	1	Set by SPRITE-DATA collision (1st one only, until reset)
IMMC	2	Set by SPRITE-SPRITE collision (1st one only, until reset)
ILP	3	Set by negative transition of light pen (1 per frame)
IRQ	7	Set by latch set and enabled

Once an interrupt bit has been set, it's "latched" in and must be cleared by writing a 1 to that bit in the interrupt register when you're ready to handle it. This allows selective interrupt handling, without having to store the other interrupt bits.

The **INTERRUPT ENABLE REGISTER** is located at 53274 (\$D01A). It has the same format as the interrupt status register. Unless the corresponding bit in the interrupt enable register is set to a 1, no interrupt from that source will take place. The interrupt status register can still be polled for information, but no interrupts will be generated.

To enable an interrupt request the corresponding interrupt enable bit (as shown in the chart above) must be set to a 1.

This powerful interrupt structure lets you use split screen modes. For instance you can have half of the screen bit mapped, half text, more than 8 sprites at a time, etc. The secret is to use interrupts properly. For example, if you want the top half of the screen to be bit mapped and the bottom to be text, just set the raster compare register (as explained previously) for halfway down the screen. When the interrupt occurs, tell the VIC-II chip to get characters from ROM, then set the raster compare register to interrupt at the top of the screen. When the interrupt occurs at the top of the screen, tell the VIC-II chip to get characters from RAM (bit map mode).

You can also display more than 8 sprites in the same way. Unfortunately BASIC isn't fast enough to do this very well. So if you want to start using display interrupts, you should work in machine language.

## SUGGESTED SCREEN AND CHARACTER COLOR COMBINATIONS

Color TV sets are limited in their ability to place certain colors next to each other on the same line. Certain combinations of screen and character colors produce blurred images. This chart shows which color combinations to avoid, and which work especially well together.

		CHARACTER COLOR															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SCREEN COLOR	0	X	●	X	●	●	■	X	●	●	X	●	●	●	●	●	●
	1	●	X	●	X	●	●	●	X	●	●	●	●	●	X	●	●
	2	X	●	X	X	●	X	X	●	●	X	●	X	X	X	X	●
	3	●	X	X	X	X	●	●	X	X	X	X	●	X	X	●	X
	4	●	●	X	X	X	X	X	X	X	X	X	X	X	X	X	●
	5	●	●	X	●	X	X	X	X	X	X	X	●	X	●	X	●
	6	●	●	X	●	X	X	X	X	X	X	X	X	X	●	●	●
	7	●	X	●	X	X	X	●	X	●	●	●	●	●	X	X	X
	8	●	●	●	X	X	X	X	●	X	●	X	X	X	X	X	●
	9	X	●	X	X	X	X	X	●	●	X	●	X	X	X	X	●
	10	●	●	●	X	X	X	X	●	X	●	X	X	X	X	X	●
	11	●	●	X	●	X	X	X	●	X	X	X	X	●	●	●	●
	12	●	●	●	X	X	X	●	X	X	●	X	●	X	X	X	●
	13	●	X	X	X	X	●	●	X	X	X	X	●	X	X	X	X
	14	●	●	X	●	X	X	●	X	X	X	X	●	X	X	X	●
	15	●	●	●	X	●	●	●	X	X	●	●	●	●	X	●	X

● = EXCELLENT  
 ● = FAIR  
 X = POOR

## PROGRAMMING SPRITES—ANOTHER LOOK

For those of you having trouble with graphics, this section has been designed as a more elementary tutorial approach to sprites.

### MAKING SPRITES IN BASIC—A SHORT PROGRAM

There are at least three different BASIC programming techniques which let you create graphic images and cartoon animations on the Commodore 64. You can use the computer's built-in graphics character set (see Page 376). You can program your own characters (see Page 108) or . . . best of all . . . you can use the computer's built-in "sprite graphics." To illustrate how easy it is, here's one of the shortest spritemaking programs you can write in BASIC:

```
10 PRINT " "
20 POKE2040,13
30 FOR S=832TO892+62:POKE S,255:NEXT
40 V=53249
50 POKEV+21,1
60 POKEV+39,1
70 POKEV,24
80 POKEV+1,100
```

This program includes the key "ingredients" you need to create any sprite. The POKE numbers come from the SPRITEMAKING CHART on Page 176. This program defines the first sprite . . . sprite 0 . . . as a solid white square on the screen. Here's a line-by-line explanation of the program:

**LINE 10** clears the screen.

**LINE 20** sets the "sprite pointer" to where the Commodore 64 will read its sprite data from. Sprite 0 is set at 2040, sprite 1 at 2041, sprite 2 at 2042, and so on up to sprite 7 at 2047. You can set all 8 sprite pointers to 13 by using this line in place of line 20:

```
20 FOR SP=2040TO2047:POKE SP,13:NEXT SP
```

**LINE 30** puts the first sprite (sprite 0) into 63 bytes of the Commodore 64's RAM memory starting at location 832 (each sprite requires 63 bytes of memory). The first sprite (sprite 0) is "addressed" at memory locations 832 to 894.

**LINE 40** sets the variable "V" equal to 53248, the starting address of the VIDEO CHIP. This entry lets us use the form (V+number) for sprite settings. We're using the form (V+number) when POKEing sprite settings because this format conserves memory and lets us work with smaller numbers. For example, in line 50 we typed POKE V+21. This is the same as typing POKE 53248+21 or POKE 53269 . . . but V+21 requires less space than 53269, and is easier to remember.

**LINE 50** enables or "turns on" sprite 0. There are 8 sprites, numbered from 0 to 7. To turn on an individual sprite, or a combination of sprites, all you have to do is POKE V+21 followed by a number from 0 (turn all sprites off) to 255 (turn all 8 sprites on). You can turn on one or more sprites by POKEing the following numbers:

ALL ON	SPRITE0	SPRITE1	SPRITE2	SPRITE3	SPRITE4	SPRITE5	SPRITE6	SPRITE7	ALL OFF
V+21,255	V+21,1	V+21,2	V+21,4	V+21,8	V+21,16	V+21,32	V+21,64	V+21,128	V+21,0

POKE V+21,1 turns on sprite 0. POKE V+21,128 turns on sprite 7. You can also turn on combinations of sprites. For example, POKE V+21,129 turns on *both* sprite 0 and sprite 7 by adding the two "turn on" numbers (1+128) together. (See SPRITEMAKING CHART, Page 176.)

**LINE 60** sets the COLOR of sprite 0. There are 16 possible sprite colors, numbered from 0 (black) to 15 (grey). Each sprite requires a different POKE to set its color, from V+39 to V+46. POKE V+39,1 colors sprite 0 white. POKE V+46,15 colors sprite 7 grey. (See the SPRITEMAKING CHART for more information.)

When you create a sprite, as you just did, the sprite will STAY IN MEMORY until you POKE it off, redefine it, or turn off your computer. This lets you change the color, position and even shape of the sprite in DIRECT or IMMEDIATE mode, which is useful for editing purposes. As an example, RUN the program above, then type this line in DIRECT mode (without a line number) and hit the **RETURN** key:

### **POKE V+39,8**

The sprite on the screen is now ORANGE. Try POKEing some other numbers from 0 to 15 to see the other sprite colors. Because you did this in DIRECT mode, if you RUN your program the sprite will return to its original color (white).

**LINE 70** determines the HORIZONTAL or "X" POSITION of the sprite on the screen. This number represents the location of the UPPER LEFT CORNER of the sprite. The farthest left horizontal (X) position which you can see on your television screen is position number 24, although you can move the sprite OFF THE SCREEN to position number 0.

**LINE 80** determines the VERTICAL or "Y" POSITION of the sprite. In this program, we placed the sprite at X (horizontal) position 24, and Y (vertical) position 100. To try another location, type this POKE in DIRECT mode and hit **RETURN**:

**POKE V,24:POKE V+1,50**

This places the sprite at the upper left corner of the screen. To move the sprite to the lower left corner, type this:

**POKE V,24:POKE V+1,229**

Each number from 832 to 895 in our sprite 0 address represents one block of 8 pixels, with three 8-pixel blocks in each horizontal row of the sprite. The loop in line 80 tells the computer to POKE 832,255 which makes the first 8 pixels solid . . . then POKE 833,255 to make the second 8 pixels solid, and so on to location 894 which is the last group of 8 pixels in the bottom right corner of the sprite. To better see how this works, try typing the following in DIRECT mode, and notice that the second group of 8 pixels is erased:

**POKE 833,0** (to put it back type POKE 833,255 or RUN your program)

The following line, which you can add to your program, erases the blocks in the MIDDLE of the sprite you created:

**90 FOR A=836 TO 891 STEP 3:POKE A,0:NEXT A**

Remember, the pixels that make up the sprite are grouped in blocks of eight. This line erases the 5th group of eight pixels (block 836) and every third block up to block 890. Try POKEing any of the other numbers from 832 to 894 with either a 255 to make them solid or 0 to make them blank.

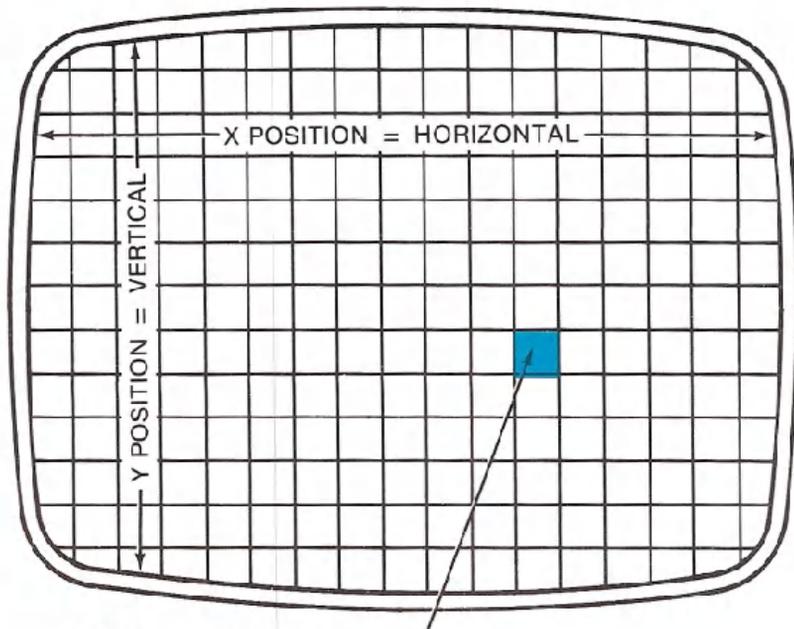
## CRUNCHING YOUR SPRITE PROGRAMS

Here's a helpful "crunching" tip: The program described above is already short, but it can be made even shorter by "crunching" it smaller. In our example we list the key sprite settings on separate program lines so you can see what's happening in the program. In actual practice, a good programmer would probably write this program as a TWO LINE PROGRAM . . . by "crunching" it as follows:

```
10PRINTCHR$(147):V=53248:POKEV+21,1:POKE2040,13:  
   POKEV+39,1  
20FORS=832TO894:POKES,255:NEXT:POKEV,24:POKEV+1,100
```

For more tips on how to crunch your programs so they fit in less memory and run more efficiently, see the "crunching guide" on Page 24.

### TV SCREEN



A Sprite located here must have both its X-position (horizontal) and Y-position (vertical) set so it can be displayed on the screen.

**Figure 3-4.** The display screen is divided into a grid of X and Y coordinates.

## POSITIONING SPRITES ON THE SCREEN

The entire display screen is divided into a grid of X and Y coordinates, like a graph. The X COORDINATE is the HORIZONTAL position across the screen and the Y COORDINATE is the VERTICAL position up and down (see Figure 3-4).

To position any sprite on the screen, you must POKE TWO SETTINGS . . . the X position and the Y position . . . these tell the computer where to display the UPPER LEFT HAND CORNER of the sprite. Remember that a sprite consists of 504 individual pixels, 24 across by 21 down . . . so if you POKE a sprite onto the upper left corner of your screen, the sprite will be displayed as a graphic image 24 pixels ACROSS and 21 pixels DOWN starting at the X-Y position you defined. The sprite will be displayed based on the upper left corner of the entire sprite, even if you define the sprite using only a small part of the 24x21-pixel sprite area.

To understand how X-Y positioning works, study the following diagram (Figure 3-5), which shows the X and Y numbers in relation to your display screen. Note that the GREY AREA in the diagram shows your television viewing area . . . the white area represents positions which are OFF your viewing screen . . .

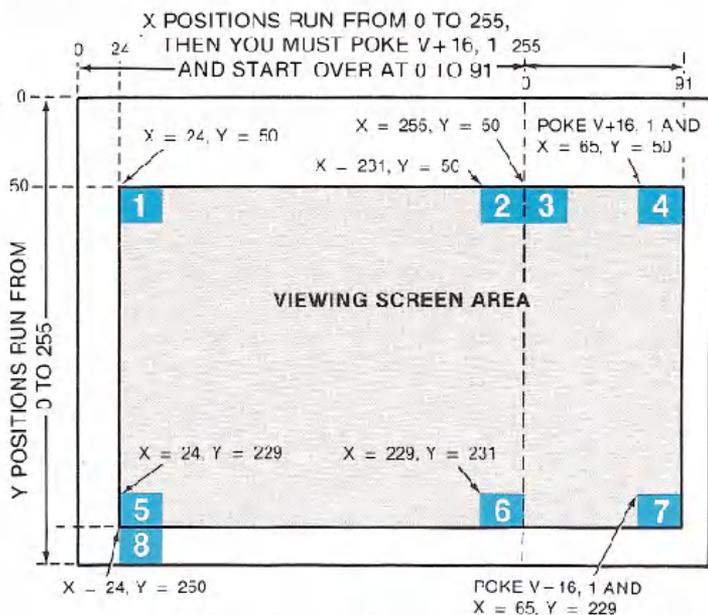


Figure 3-5. Determining X-Y sprite positions.

To display a sprite in a given location, you must POKE the X and Y settings for each SPRITE . . . remembering that every sprite has its own unique X POKE and Y POKE. The X and Y settings for all 8 sprites are shown here:

### POKE THESE VALUES TO SET X-Y SPRITE POSITIONS

	SPRITE0	SPRITE1	SPRITE2	SPRITE3	SPRITE4	SPRITE5	SPRITE6	SPRITE7
SET X	V,X	V+2,X	V+4,X	V+6,X	V+8,X	V+10,X	V+12,X	V+14,X
SET Y	V+1,Y	V+3,Y	V+5,Y	V+7,Y	V+9,Y	V+11,Y	V+13,Y	V+15,Y
RIGHTX	V+16,1	V+16,2	V+16,4	V+16,8	V+16,16	V+16,32	V+16,64	V+16,128

**POKEING AN X POSITION:** The possible values of X are 0 to 255, counting from left to right. Values 0 to 23 place all or part of the sprite OUT OF THE VIEWING AREA off the left side of the screen . . . values 24 to 255 place the sprite IN THE VIEWING AREA up to the 255th position (see next paragraph for settings beyond the 255th X position). To place the sprite at one of these positions, just type the X-POSITION POKE for the sprite you're using. For example, to POKE sprite 1 at the farthest left X position IN THE VIEWING AREA, type: POKE V+2,24.

**X VALUES BEYOND THE 255TH POSITION:** To get beyond the 255th position across the screen, you need to make a SECOND POKE using the numbers in the "RIGHT X" row of the chart (Figure 3-5). Normally, the horizontal (X) numbering would continue past the 255th position to 256, 257, etc., but because registers only contain 8 bits we must use a "second register" to access the RIGHT SIDE of the screen and start our X numbering over again at 0. So to get beyond X position 255, you must POKE V+16 and a number (depending on the sprite). This gives you 65 additional X positions (renumbered from 0 to 65) in the viewing area on the RIGHT side of the viewing screen. (You can actually POKE the right side X value as high as 255, which takes you off the right edge of the viewing screen.)

**POKEING A Y POSITION:** The possible values of Y are 0 to 255, counting from top to bottom. Values 0 to 49 place all or part of the sprite OUT OF THE VIEWING AREA off the TOP of the screen. Values 50 to 229 place the sprite IN THE VIEWING AREA. Values 230 to 255 place all or part of the sprite OUT OF THE VIEWING AREA off the BOTTOM of the screen.

Let's see how this X-Y positioning works, using sprite 1. Type this program:

```
SHIFT CLR/HOME
10 PRINT"0":V=53248:POKEV+21,2:POKE2041,13:
FOR$=832TO895:POKE$,255:NEXT
20 POKEV+40,7
30 POKEV+2,24
40 POKEV+3,50
```

This simple program establishes sprite 1 as a solid box and positions it at the upper left corner of the screen. Now change line 40 to read:

```
40 POKE V+3,229
```

This moves the sprite to the bottom left corner of the screen. Now let's test the RIGHT X LIMIT of the sprite. Change line 30 as shown:

```
30 POKE V+2,255
```

This moves the sprite to the RIGHT but reaches the RIGHT X LIMIT, which is 255. At this point, the "most significant bit" in register 16 must be SET. In other words, you must type POKE V+16 and the number shown in the "RIGHT X" column in the X-Y POKE CHART above to RESTART the X position counter at the 256th pixel/position on the screen. Change line 30 as follows:

```
30 POKE V+16, PEEK(V+16)OR 2:POKE V+2,0
```

**POKE V+16,2** sets the most significant bit of the X position for sprite 1 and restarts it at the 256th pixel/position on the screen. **POKE V+2,0** displays the sprite at the NEW POSITION ZERO, which is now reset to the 256th pixel.

*To get back to the left side of the screen, you must reset the most significant bit of the X position counter to 0 by typing (for sprite 1):*

```
POKE V+16, PEEK(V+16)AND 253
```

**TO SUMMARIZE** how the X positioning works . . . POKE the X POSITION for any sprite with a number from 0 to 255. To access a position beyond the 255th position/pixel across the screen, you must use an additional POKE (V+16) which sets the most significant bit of the X position and start counting from 0 again at the 256th pixel across the screen.

This POKE starts the X numbering over again from 0 at the 256th position (**Example: POKE V+16, PEEK(V+16)OR 1** and **POKE V,1** must be included to place sprite 0 at the 257th pixel across the screen.) To get back to the left side X positions you have to **TURN OFF** the control setting by typing **POKE V+16, PEEK(V+16)AND 254**.

## POSITIONING MULTIPLE SPRITES ON THE SCREEN

Here's a program which defines **THREE DIFFERENT SPRITES (0, 1, and 2)** in different colors and places them in different positions on the screen:

```
SHIFT CLR/HOME
10 PRINT "0":V=52248:FORM=002T0095:POKE V,255:NEXT
20 FORM=2040T02642:POKE V,13:NEXT
30 POKE V+21,7
40 POKE V+39,1:POKE V+40,7:POKE V+41,8
50 POKE V,24:POKE V+1,50
60 POKE V+2,12:POKE V+3,229
70 POKE V+4,255:POKE V+5,50
```

For convenience, all 3 sprites have been defined as solid squares, getting their data from the same place. The important lesson here is how the 3 sprites are positioned. The white sprite 0 is at the top lefthand corner. The yellow sprite 1 is at the bottom lefthand corner but **HALF** the sprite is **OFF THE SCREEN** (remember, 24 is the leftmost X position in the viewing area . . . an X position less than 24 puts all or part of the sprite off the screen and we used an X position 12 here which put the sprite halfway off the screen). Finally, the orange sprite 2 is at the **RIGHT X LIMIT** (position 255) . . . but what if you want to display a sprite in the area to the **RIGHT** of X position 255?

## DISPLAYING A SPRITE BEYOND THE 255TH X-POSITION

Displaying a sprite beyond the 255th X position requires a special POKE which **SETS** the most significant bit of the X position and starts over at the 256th pixel position across the screen. Here's how it works . . .

First, you **POKE V+16** with the number for the sprite you're using (check the "RIGHT X" row in the X-Y chart . . . we'll use sprite 0). Now we assign an X position, keeping in mind that the X counter starts over from 0 at the 256th position on the screen. Change line 50 to read as follows:

```
50 POKE V+16,1:POKE V,24:POKE V+1,75
```

This line POKes V+16 with the number required to "open up" the right side of the screen. . .the new X position 24 for sprite 0 now begins 24 pixels to the RIGHT of position 255. To check the right edge of the screen, change line 60 to:

```
60 POKE V+16,1:POKE V,65:POKE V+1,75
```

Some experimentation with the settings in the sprite chart will give you the settings you need to position and move sprites on the left and right sides of the screen. The section on "moving sprites" will also increase your understanding of how sprite positioning works.

## SPRITE PRIORITIES

You can actually make different sprites seem to move IN FRONT OF or BEHIND each other on the screen. This incredible three dimensional illusion is achieved by the built-in SPRITE PRIORITIES which determine which sprites have priority over the others when 2 or more sprites OVERLAP on the screen.

The rule is "first come, first served" which means lower-numbered sprites AUTOMATICALLY have priority over higher-numbered sprites. For example, if you display sprite 0 and sprite 1 so they overlap on the screen, sprite 0 will appear to be IN FRONT OF sprite 1. Actually, sprite 0 always supersedes all the other sprites because it's the lowest numbered sprite. In comparison, sprite 1 has priority over sprites 2-7; sprite 2 has priority over sprites 3-7, etc. Sprite 7 (the last sprite) has LESS PRIORITY than any of the other sprites, and will always appear to be displayed "BEHIND" any other sprites which overlap its position.

To illustrate how priorities work, change lines 50, 60, and 70 in the program above to the following:

```
10 PRINT "S":V=53249:FORS=332TO895:POKES,255:NEXT S
20 FORM=2040TO2042:POKEM,13:NEXT M
30 POKEV+21,7
40 POKEV+39,1:POKEV+40,7:POKEV+41,8
50 POKEV,24:POKEV+1,50:POKEV+16,0
60 POKEV+2,34:POKEV+3,60
70 POKEV+4,44:POKEV+5,70
```

You should see a white sprite on top of a yellow sprite on top of an orange sprite. Of course, now that you see how priorities work, you can also MOVE SPRITES and take advantage of these priorities in your animation.





puters as a special way of counting. Here's a close-up view of the first eight pixels in the top left hand corner of the sprite:

128	64	32	16	8	4	2	1

**STEP 4:**

Add up the numbers of the SOLID pixels. This first group of eight pixels is completely solid, so the total number is 255.

**STEP 5:**

Enter that number as the FIRST DATA STATEMENT in line 100 of the Spritemaking Program below. Enter 255 for the second and third groups of eight.

**STEP 6:**

Look at the FIRST EIGHT PIXELS IN THE SECOND ROW of the sprite. Add up the values of the solid pixels. Since only one of these pixels is solid, the total value is 128. Enter this as the first DATA number in line 101.

128	64	32	16	8	4	2	1

**STEP 7:**

Add up the values of the next group of eight pixels (which is 0 because they're all BLANK) and enter in line 101. Now move to the next group of pixels and repeat the process for each GROUP OF EIGHT PIXELS (there are 3 groups across each row, and 21 rows). This will give you a total of 63 numbers. Each number represents ONE group of 8 pixels, and 63 groups of eight equals 504 total individual pixels. Perhaps a better way of looking at the program is like this . . . each line in the program represents ONE ROW in the sprite. Each of the 3 numbers in each row represents ONE GROUP OF EIGHT PIXELS. And each number tells the computer which pixels to make SOLID and which pixels to leave blank.

## STEP 8:

CRUNCH YOUR PROGRAM INTO A SMALLER SPACE BY RUNNING TOGETHER ALL THE DATA STATEMENTS, AS SHOWN IN THE SAMPLE PROGRAM BELOW. Note that we asked you to write your sprite program on a piece of paper. We did this for a good reason. The DATA STATEMENT LINES 100–120 in the program in STEP 1 are only there to help you see which numbers relate to which groups of pixels in your sprite. Your final program should be “crunched” like this:

```
SHIFT CLR/HOME
10 PRINT "D":POKE53280,5:POKE53281,6
20 V=53248:POKEV+34,3
30 POKE53269,4:POKE2042,10
40 FORN=0TO62:READQ:POKE932+N,Q:NEXT
100 DATA255,255,255,128,0,1,128,0,1,128,0,1,144,0,
1,144,0,1,144,0,1,144,0,1
101 DATA144,0,1,144,0,1,144,0,1,144,0,1,144,0,1,
144,0,1,128,0,1,128,0,1
102 DATA128,0,1,128,0,1,128,0,1,128,0,1,255,255,255
200 X=200:Y=100:POKE53252,X:POKE53253,Y
```

## MOVING YOUR SPRITE ON THE SCREEN

Now that you've created your sprite, let's do some interesting things with it. To move your sprite smoothly across the screen, add these two lines to your program:

```
50 POKE V+5,100:FOR X=24TO255:POKE V+4,X:NEXT:POKE
V+16,4
55 FOR X=0TO65:POKE V+4,X:NEXT X:POKE V+16,0:GOTO 50
```

**LINE 50** POKEs the Y POSITION at 100 (try 50 or 229 instead for variety). Then it sets up a FOR . . . NEXT loop which POKEs the sprite into X position 0 to X position 255, in order. When it reaches the 255th position, it POKEs the RIGHT X POSITION (POKE V+16,4) which is required to cross to the right side of the screen.

**LINE 55** has a FOR . . . NEXT loop which continues to POKE the sprite in the last 65 positions on the screen. Note that the X value was reset to zero but because you used the RIGHT X setting (POKE V+16,2) X starts over on the right side of the screen.

This line keeps going back to itself (GOTO 50). If you just want the sprite to move ONCE across the screen and disappear, then take out GOTO50.

Here's a line which moves the sprite BACK AND FORTH:

```
50 POKE V+5,100:FOR X=24TO255:POKE V+4,X:NEXT: POKE
   V+16,4:FOR X=0TO65: POKE V+4,X: NEXT X
55 FOR X=65TO0 STEP-1:POKE V+4,X:NEXT:POKE V+16,0: FOR
   X=255TO24 STEP-1: POKE V+4,X:NEXT
60 GOTO 50
```

Do you see how these programs work? This program is the same as the previous one, except when it reaches the end of the right side of the screen, it REVERSES ITSELF and goes back in the other direction. That is what the STEP-1 accomplishes . . . it tells the program to POKE the sprite into X values from 65 to 0 on the right side of the screen, then from 255 to 0 on the left side of the screen, STEPPing backwards minus-1 position at a time.

## VERTICAL SCROLLING

This type of sprite movement is called "scrolling." To scroll your sprite up or down in the Y position, you only have to use ONE LINE. ERASE LINES 50 and 55 by typing the line numbers by themselves and hitting **RETURN** like this:

```
50 ( RETURN )
55 ( RETURN )
```

Now enter LINE 50 again as follows:

```
50 POKE V+4,24:FOR Y=0TO255:POKE V+5,Y:NEXT
```

## THE DANCING MOUSE—A SPRITE PROGRAM EXAMPLE

Sometimes the techniques described in a programmer's reference manual are difficult to understand, so we've put together a fun sprite program called "Michael's Dancing Mouse." This program uses three different sprites in a cute animation with sound effects—and to help you understand how it works we've included an explanation of EACH COMMAND so you can see exactly how the program is constructed:

```
5 S=54272:POKES+24,15:POKES,220:POKES+1,68:POKES+5,
15:POKES+C,215
10 POKES+7,120:POKES+8,100:POKES+12,15:POKES+13,215
```

SHIFT CLR/HOME

```
15 PRINT"O":V=53248:POKEV+2,1
20 FORS1=12288TO12350:READQ1:POKES1,Q1:NEXT
25 FORS2=12352TO12414:READQ2:POKES2,Q2:NEXT
30 FORS3=12416TO12478:READQ3:POKES3,Q3:NEXT
35 POKEV+3S,15:POKEV-1,68
```

CTRL 2

G 7

```
40 PRINTTAB(160)" I AM THE DANCING MOUSE!"
45 P=192
50 FORX=0TO347STEP3
55 RX=INT(X/256):LX=X-RX*256
60 POKEV,LX:POKEV+16,RX
70 IFF=192THENGOSUB200
75 IFF=193THENGOSUB300
80 POKE2040,P:FORI=1TO60:NEXT
85 P=P+1:IFF>194THENP=192
90 NEXT
95 END
100 DATA0,0,120,63,0,252,127,129,254,127,129,254,
127,189,254,127,255,254
101 DATA63,255,252,31,187,248,3,187,192,1,255,128,
3,189,192,1,231,128,1,255,0
102 DATA31,255,0,0,124,0,0,254,0,1,199,32,3,131,
224,7,1,192,1,192,0,3,192,0
103 DATA0,0,120,63,0,252,127,129,254,127,129,254,
127,189,254,127,255,254
104 DATA63,255,252,31,221,248,3,221,192,1,255,128,
3,255,192,1,195,128,1,231,3
105 DATA31,255,255,0,124,0,0,254,0,1,199,0,7,1,128,
7,0,204,1,128,124,7,128,56
106 DATA0,0,120,63,0,252,127,129,254,127,129,254,
127,189,254,127,255,254
107 DATA63,255,252,31,221,248,3,221,192,1,255,134,
3,139,204,1,199,152,1,255,48
108 DATA1,255,224,1,252,0,3,254,0
109 DATA7,14,0,204,14,0,248,56,0,112,112,0,0,0,0,0,
-1
200 POKES+4,129:POKES+4,128:RETURN
300 POKES+11,129:POKES+11,128:RETURN
```

#### LINE 5:

- S=54272** Sets the variable S equal to 54272, which is the beginning memory location of the SOUND CHIP. From now on, instead of poking a direct memory location, we will POKE S plus a value.
- POKES+24,15** Same as POKE 54296,15 which sets VOLUME to highest level.
- POKES,220** Same as POKE 54272,220 which sets Low Frequency in Voice 1 for a note which approximates high C in Octave 6.
- POKES+1,68** Same as POKE 54273,68 which sets High Frequency in Voice 1 for a note which approximates high C in Octave 6.
- POKES+5,15** Same as POKE 54277,15 which sets Attack/Decay for Voice 1 and: in this case consists of the maximum DECAY level with no attack, which produces the "echo" effect.
- POKES+6,215** Same as POKE 54278,215 which sets Sustain/ Release for Voice 1 (215 represents a combination of sustain and release values).

#### LINE 10:

- POKES+7,120** Same as POKE 54279,120 which sets the Low Frequency for Voice 2.
- POKES+8,100** Same as POKE 54280,100 which sets the High Frequency for Voice 2.
- POKES+12,15** Same as POKE 54284,15 which sets Attack/Decay for Voice 2 to same level as Voice 1 above.
- POKES+13,215** Same as POKE 54285,215 which sets Sustain/ Release for Voice 2 to same level as Voice 1 above.

#### LINE 15:

- PRINT" SHIFT**  
**CLR / HOME "**  
**V=53248** Clears the screen when the program begins. Defines the variable "V" as the starting location of the VIC chip which controls sprites. From now on we will define sprite locations as V plus a value.
- POKEV+21,1** Turns on (enables) sprite number 1.

## LINE 20:

FOR S1=12288  
TO 12350

We are going to use ONE SPRITE (sprite 0) in this animation, but we are going to use THREE sets of sprite data to define three separate shapes. To get our animation, we will switch the POINTERS for sprite 0 to the three places in memory where we have stored the data which defines our three different shapes. The same sprite will be redefined rapidly over and over again as 3 different shapes to produce the dancing mouse animation. You can define dozens of sprite shapes in DATA STATEMENTS, and rotate those shapes through one or more sprites. So you see, you don't have to limit one sprite to one shape or vice-versa. One sprite can have many different shapes, simply by changing the POINTER SETTING FOR THAT SPRITE to different places in memory where the sprite data for different shapes is stored. This line means we have put the DATA for "sprite shape 1" at memory locations 12288 to 12350.

READ Q1

Reads 63 numbers in order from the DATA statements which begin at line 100. Q1 is an arbitrary variable name. It could just as easily be A, Z1 or another numeric variable.

POKE S1,Q1

Pokes the first number from the DATA statements (the first "Q1" is 30) into the first memory location (the first memory location is 12288). This is the same as POKE12288,30.

NEXT

This tells the computer to look BETWEEN the FOR and NEXT parts of the loop and perform those in-between commands (READQ1 and POKES1,Q1 using the NEXT numbers in order). In other words, the NEXT statement makes the computer READ the NEXT Q1 from the DATA STATEMENTS, which is 0, and also increments S1 by 1 to the next value, which is 12289. The result is POKE12289,0 . . . the NEXT command makes the loop keep going back until the last values in the series, which are POKE 12350,0.

### LINE 25:

FORS2=12352 TO 12414	The second shape of sprite zero is defined by the DATA which is located at locations 12352 to 12414. NOTE that location 12351 is SKIPPED . . . this is the 64th location which is used in the definition of the first sprite group but does not contain any of the sprite data numbers. Just remember when defining sprites in consecutive locations that you will use 64 locations, but only POKE sprite data into the first 63 locations.
READQ2	Reads the 63 numbers which follow the numbers we used for the first sprite shape. This READ simply looks for the very next number in the DATA area and starts reading 63 numbers, one at a time.
POKES2,Q2	Pokes the data (Q2) into the memory locations (S2) for our second sprite shape, which begins at location 12352.
NEXT	Same use as line 20 above.

### LINE 30:

FORS3=12416 TO 12478	The third shape of sprite zero is defined by the DATA to be located at locations 12416 to 12478.
READQ3	Reads last 63 numbers in order as Q3.
POKES3,Q3	Pokes those numbers into locations 12416 to 12478.
NEXT	Same as lines 20 and 25.

### LINE 35:

POKEV+39,15	Sets color for sprite 0 to light grey.
POKEV+1,68	Sets the upper right hand corner of the sprite square to vertical (Y) position 68. For the sake of comparison, position 50 is the top lefthand corner Y position on the viewing screen.

#### LINE 40:

PRINTTAB(160)

Tabs 160 spaces from the top lefthand CHARACTER SPACE on the screen, which is the same as 4 rows beneath the clear command . . . this starts your PRINT message on the 6th line down on the screen.

" CTRL WHT

Hold down the CTRL key and press the key marked WHT at the same time. If you do this inside quotation marks, a "reversed E" will appear. This sets the color to everything PRINTed from then on to WHITE.

I AM THE  
DANCING  
MOUSE!

⌘ 7"

This is a simple PRINT statement.

This sets the color back to light blue when the PRINT statement ends. Holding down ⌘ and 7 at the same time inside quotation marks causes a "reversed diamond symbol" to appear.

#### LINE 45:

P=192

Sets the variable P equal to 192. This number 192 is the pointer you must use, in this case to "point" sprite 0 to the memory locations that begin at location 12288. Changing this pointer to the locations of the other two sprite shapes is the secret of using one sprite to create an animation that is actually three different shapes.

#### LINE 50:

FORX=0TO347  
STEP3

Steps the movement of your sprite 3 X positions at a time (to provide fast movement) from position 0 to position 347.

#### LINE 55:

`RX=INT(X/256)`

`RX` is the integer of  $X/256$  which means that `RX` is rounded off to 0 when `X` is less than 256, and `RX` becomes 1 when `X` reaches position 256. We will use `RX` in a moment to `POKE V+16` with a 0 or 1 to turn on the "RIGHT SIDE" of the screen.

`LX=X-RX*256`

When the sprite is at `X` position 0, the formula looks like this:  $LX = 0 - (0 \text{ times } 256)$  or 0. When the sprite is at `X` position 1 the formula looks like this:  $LX = 1 - (0 \text{ times } 256)$  or 1. When the sprite is at `X` position 256 the formula looks like this:  $LX = 256 - (1 \text{ times } 256)$  or 0 which resets `X` back to 0 which must be done when you start over on the RIGHT SIDE of the screen (`POKEV+16,1`).

#### LINE 60:

`POKEV,LX`

You `POKE V` by itself with a value to set the Horizontal (`X`) Position of sprite 0 on the screen. (See `SPRITE MAKING CHART` on Page 176). As shown above, the value of `LX`, which is the horizontal position of the sprite, changes from 0 to 255 and when it reaches 255 it automatically resets back to zero because of the `LX` equation set up in line 55.

`POKEV+16,RX`

`POKEV+16` always turns on the "right side" of the screen beyond position 256, and resets the horizontal positioning coordinates to zero. `RX` is either a 0 or a 1 based on the position of the sprite as determined by the `RX` formula in line 55.

#### LINE 70:

`IFP=192THEN  
GOSUB200`

If the sprite pointer is set to 192 (the first sprite shape) the waveform control for the first sound effect is set to 129 and 128 per line 200.

#### LINE 75:

```
IFP=193THEN  
GOSUB300
```

If the sprite pointer is set to 193 (the second sprite shape) the waveform control for the second sound effect (Voice 2) is set to 129 and 128 per line 300.

#### LINE 80:

```
POKE2040,P
```

Sets the SPRITE POINTER to location 192 (remember  $P=192$  in line 45? Here's where we use the P).

```
FORT=1TO60:  
NEXT
```

A simple time delay loop which sets the speed at which the mouse dances. (Try a faster or slower speed by increasing/decreasing the number 60.)

#### LINE 85:

```
P=P+1
```

Now we increase the value of the pointer by adding 1 to the original value of P.

```
IFP>194THEN  
P=192
```

We only want to point the sprite to 3 memory locations. 192 points to locations 12288 to 12350, 193 points to locations 12352 to 12414, and 194 points to locations 12416 to 12478. This line tells the computer to reset P back to 192 as soon as P becomes 195 so P never really becomes 195. P is 192, 193, 194 and then resets back to 192 and the pointer winds up pointing consecutively to the three sprite shapes in the three 64-byte groups of memory locations containing the DATA.

#### LINE 90:

NEXTX

After the sprite has become one of the 3 different shapes defined by the DATA, only then is it allowed to move across the screen. It will jump 3 X positions at a time (instead of scrolling smoothly one position at a time, which is also possible). STEPPing 3 positions at a time makes the mouse "dance" faster across the screen. NEXT X matches the FOR. . . X position loop in line 50.

#### LINE 95

END

ENDs the program, which occurs when the sprite moves off the screen.

#### LINES 100-109

DATA

The sprite shapes are read from the data numbers, in order. First the 63 numbers which comprise sprite shape 1 are read, then the 63 numbers for sprite shape 2, and then sprite shape 3. This data is permanently read into the 3 memory locations and after it is read into these locations, all the program has to do is point sprite 0 at the 3 memory locations and the sprite automatically takes the shape of the data in those locations. We are pointing the sprite at 3 locations one at a time which produces the "animation" effect. If you want to see how these numbers affect each sprite, try changing the first 3 numbers in LINE 100 to 255, 255, 255. See the section on defining sprite shapes for more information.

#### LINE 200:

- POKES+4,129      Waveform control set to 129 turns on the sound effect.
- POKES+4,128      Waveform control set to 128 turns off the sound effect.
- RETURN            Sends program back to end of line 70 after waveform control settings are changed, to resume program.

#### LINE 300:

- POKES+11,129     Waveform control set to 129 turns on the sound effect.
- POKES+11,128     Waveform control set to 128 turns off the sound effect.
- RETURN            Sends program back to end of line 75 to resume.

## EASY SPRITEMAKING CHART

	SPRITE 0	SPRITE 1	SPRITE 2	SPRITE 3	SPRITE 4	SPRITE 5	SPRITE 6	SPRITE 7
Turn on Sprite	V+21,1	V+21,2	V+21,4	V+21,8	V+21,16	V+21,32	V+21,64	V+21,128
Put in Memory (Set Pointers)	2040, 192	2041, 193	2042, 194	2043, 195	2044, 196	2045, 197	2046, 198	2047, 199
Locations for Sprite Pixel (12288-12798)	12288 to 12350	12352 to 12414	12416 to 12478	12480 to 12542	12544 to 12606	12508 to 12670	12672 to 12734	12736 to 12798
Sprite Color	V+39,C	V+40,C	V+41,C	V+42,C	V+43,C	V+44,C	V+45,C	V+46,C
Set LEFT X Position (0-255)	V+0,X	V+2,X	V+4,X	V+6,X	V+8,X	V+10,X	V+12,X	V+14,X
Set RIGHT X Position (0-255)	V+16,1 V+0,X	V+16,2 V+2,X	V+16,4 V+4,X	V+16,8 V+6,X	V+16,16 V+8,X	V+16,32 V+10,X	V+16,64 V+12,X	V+16,128 V+14,X
Set Y Position	V+1,Y	V+3,Y	V+5,Y	V+7,Y	V+9,Y	V+11,Y	V+13,Y	V+15,Y
Expand Sprite Horizontally/X	V+29,1	V+29,2	V+29,4	V+29,8	V+29,16	V+29,32	V+29,64	V+29,128
Expand Sprite Vertically/Y	V+23,1	V+23,2	V+23,4	V+23,8	V+23,16	V+23,32	V+23,64	V+23,128
Turn On (Set) Multi-Color Mode	V+28,1	V+28,2	V+28,4	V+28,8	V+28,16	V+28,32	V+28,64	V+28,128
Multi-Color 1 (First Color)	V+37,C	V+37,C	V+37,C	V+37,C	V+37,C	V+37,C	V+37,C	V+37,C
Multi-Color 2 (Second Color)	V+38,C	V+38,C	V+38,C	V+38,C	V+38,C	V+38,C	V+38,C	V+38,C
Set Priority of Sprites	The rule is that lower numbered sprites always have display priority over higher numbered sprites. For example, sprite 0 has priority over ALL other sprites, sprite 7 has last priority. This means lower numbered sprites always appear to move IN FRONT OF or ON TOP OF higher numbered sprites.							
Collision (Sprite to Sprite)	V+30 IF PEEK(V+30)ANDX=X THEN [action]							
Collision (Sprite to Background)	V+31 IF PEEK(V+31)ANDX=X THEN [action]							

## SPRITEMAKING NOTES

### Alternative Sprite Memory Pointers and Memory Locations Using Cassette Buffer

Put in Memory (Set pointers)	SPRITE 0 2040,13	SPRITE 1 2041,14	SPRITE 2 2042,15	If you're using 1 to 3 sprites you can use these memory locations in the cassette buffer (832 to 1023) but for more than 3 sprites we suggest using locations from 12288 to 12796 (see chart).
Sprite Pixel Locations for Blocks 13-15	832 to 894	896 to 958	960 to 1022	

### TURNING ON SPRITES:

You can turn on any individual sprite by using POKE V+21 and the number from the chart . . . BUT . . . turning on just ONE sprite will turn OFF any others. To turn on TWO OR MORE sprites, ADD TOGETHER the numbers of the sprites you want to turn on (Example: POKE V+21, 6 turns on sprites 1 and 2). Here is a method you can use to turn one sprite off and on without affecting any of the others (useful for animation).

### EXAMPLE:

To turn off just sprite 0 type: POKE V+21,PEEK V+21AND(255-1). Change the number 1 in (255-1) to 1,2,4,8,16,32,64, or 128 (for sprites 0-7). To re-enable the sprite and not affect the other sprites currently turned on, POKE V+21, PEEK(V+21)OR 1 and change the OR 1 to OR 2 (sprite 2), OR 4 (sprite 3), etc.

### X POSITION VALUES BEYOND 255:

X positions run from 0 to 255 . . . and then START OVER from 0 to 255. To put a sprite beyond X position 255 on the far right side of the screen, you must first POKE V+16 as shown, THEN POKE a new X value from 0 to 63, which will place the sprite in one of the X positions at the right side of the screen. To get back to positions 0-255, POKE V+16,0 and POKE in an X value from 0 to 255.

### Y POSITION VALUES:

Y positions run from 0 to 255, including 0 to 49 off the TOP of the viewing area, 50 to 229 IN the viewing area, and 230 to 255 off the BOTTOM of the viewing area.

## SPRITE COLORS:

To make sprite 0 WHITE, type: POKE V+39,1 (use COLOR POKE SETTING shown in chart, and INDIVIDUAL COLOR CODES shown below):

0—BLACK	4—PURPLE	8—ORANGE	12—MED. GREY
1—WHITE	5—GREEN	9—BROWN	13—LT. GREEN
2—RED	6—BLUE	10—LT. RED	14—LT. BLUE
3—CYAN	7—YELLOW	11—DARK GREY	15—LT. GREY

## MEMORY LOCATION:

You must "reserve" a separate 64-BYTE BLOCK of numbers in the computer's memory for each sprite of which 63 BYTES will be used for sprite data. The memory settings shown below are recommended for the "sprite pointer" settings in the chart above. Each sprite will be unique and you'll have to define it as you wish. To make all sprites exactly the same, point the sprites you want to look the same to the same register for sprites.

## DIFFERENT SPRITE POINTER SETTINGS:

These sprite pointer settings are RECOMMENDATIONS ONLY.

**Caution:** you can set your sprite pointers anywhere in RAM memory but if you set them too "low" in memory a long BASIC program may overwrite your sprite data, or vice versa. To protect an especially LONG BASIC PROGRAM from overwriting sprite data, you may want to set the sprites at a higher area of memory (for example, 2040,192 for sprite 0 at locations 12288 to 12350 . . . 2041,193 at locations 12352 to 12414 for sprite 1 and so on . . . by adjusting the memory locations from which sprites get their "data," you can define as many as 64 different sprites plus a sizable BASIC program. To do this, define several sprite "shapes" in your DATA statements and then redefine a particular sprite by changing the "pointer" so the sprite you are using is "pointed" at different areas of memory containing different sprite picture data. See the "Dancing Mouse" to see how this works. If you want two or more sprites to have THE SAME SHAPE (you can still change position and color of each sprite), use the same sprite pointer and memory location for the sprites you want to match (for example, you can point sprites 0 and 1 to the same location by using POKE 2040,192 and POKE 2041, 192).

## PRIORITY:

Priority means one sprite will appear to move "in front of" or "behind" another sprite on the display screen. Sprites with more priority always appear to move "in front of" or "on top of" sprites with less priority. The rule is that lower numbered sprites have priority over higher numbered sprites. Sprite 0 has priority over all other sprites. Sprite 7 has no priority in relation to the other sprites. Sprite 1 has priority over sprites 2-7, etc. If you put two sprites in the same position, the sprite with the higher priority will appear IN FRONT OF the sprite with the lower priority. The sprite with lower priority will either be obscured, or will "show through" (from "behind") the sprite with higher priority.

## USING MULTI-COLOR:

You can create multi-colored sprites although using multi-color mode requires that you use PAIRS of pixels instead of individual pixels in your sprite picture (in other words each colored "dot" or "block" in the sprite will consist of two pixels side by side). You have 4 colors to choose from: Sprite Color (chart above), Multi-Color 1, Multi-Color 2 and "Background Color" (background is achieved by using zero settings which let the background color "show through"). Consider one horizontal 8-pixel block in a sprite picture. The color of each PAIR of pixels is determined according to whether the left, right, or both pixels are solid, like this:



**BACKGROUND** (Making BOTH PIXELS BLANK (zero) lets the INNER SCREEN COLOR (background) show through.)



**MULTI-COLOR 1** (Making the RIGHT PIXEL SOLID in a pair of pixels sets BOTH PIXELS to Multi-Color 1.)

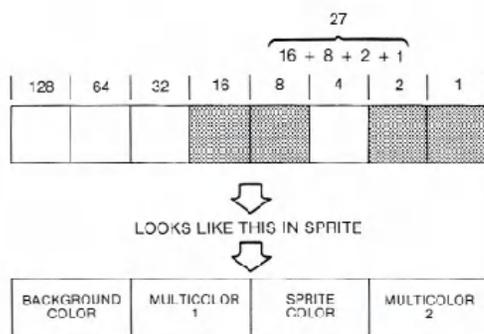


**SPRITE COLOR** (Making the LEFT PIXEL SOLID in a pair of pixels sets BOTH PIXELS to Sprite Color.)



**MULTI-COLOR 2** (Making BOTH PIXELS SOLID in a pair of pixels sets BOTH PIXELS to Multi-Color 2.)

Look at the horizontal 8-pixel row shown below. This block sets the first two pixels to background color, the second two pixels to Multi-Color 1, the third two pixels to Sprite Color and the fourth two pixels to Multi-Color 2. The color of each PAIR of pixels depends on which bits in each pair are solid and which are blank, according to the illustration above. After you determine which colors you want in each pair of pixels, the next step is to add the values of the solid pixels in the 8-pixel block, and POKE that number into the proper memory location. For example, if the 8-pixel row shown below is the first block in a sprite which begins at memory location 832, the value of the solid pixels is  $16+8+2+1 = 27$ , so you would POKE 832,27.



### COLLISION:

You can detect whether a sprite has collided with another sprite by using this line: `IF PEEK(V+30)ANDX=XTHEN [insert action here]`. This line checks to see if a particular sprite has collided with ANY OTHER SPRITE, where X equals 1 for sprite 0, 2 for sprite 1, 4 for sprite 2, 8 for sprite 3, 16 for sprite 4, 32 for sprite 5, 64 for sprite 6, and 128 for sprite 7. To check to see if the sprite has collided with a "BACKGROUND CHARACTER" use this line: `IF PEEK(V+31)ANDX=XTHEN [insert action here]`.

## USING GRAPHIC CHARACTERS IN DATA STATEMENTS

The following program allows you to create a sprite using blanks and solid circles ( **SHIFT** **Q** ) in DATA statements. The sprite and the numbers POKEd into the sprite data registers are displayed.

```

SHIFT CLR/HOME
10 PRINT " ":FOR I=0 TO 63:POKE832+I,0:NEXT
20 GOSUB 60000
300 END
60000 DATA"          @@@@@@@"          "
60001 DATA"        @@@@|@@@@@@"        "
60002 DATA"      @@@@@@@@@@@@@@@"      "
60003 DATA"    @@@@|      @@@@@@@"    "
60004 DATA"  @@@@@| @@@@| @@@@@@"  "
60005 DATA" @@@@@| @@@| @@@@@@" @ "
60006 DATA" @@@@@| @@@| @@@@@@" @ "
60007 DATA" @@@@@|      @@@@@@" @ "
60008 DATA" @@@@@@@@@@@@@@@" @ "
60009 DATA" @@@@@@@@@@@@@@@" @ "
60010 DATA" @ @@@@@@@@@@@" @ "
60011 DATA" @ @@@@@@@@@@" @ "
60012 DATA" @ @@@@@@" @ "
60013 DATA" @ @@@@@" @ "
60014 DATA" @ @@@@" @ "
60015 DATA" @ @@" @ "
60016 DATA" @ @@" @ "
60017 DATA" @ @@" @ "
60018 DATA" @ @@" @ "
60019 DATA" @ @@" @ "
60020 DATA" @ @@" @ "
60100 V=53248:POKEV,200:POKEV+1,100:POKEV+2,1:
POKEV+3,14:POKE2040,13
60105 POKEV+23,1:POKEV+29,1
60110 FOR I=0 TO 20:READA$:FOR K=0 TO 2:T=0:FOR J=0 TO 7:I=0
60140 IF MID$(A$,J+1,1)="#":THEN I=1
60150 T=T+B*2+(7-J):NEXT:PRINTT:POKE832+I#3-K,T:
NEXT:PRINT:NEXT
60200 RETURN

```