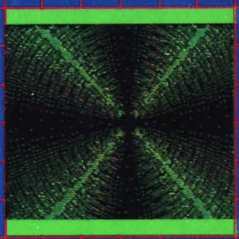
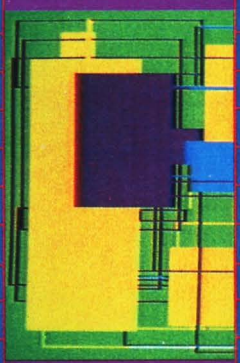
COLOR COMPUTER GRAPHICS

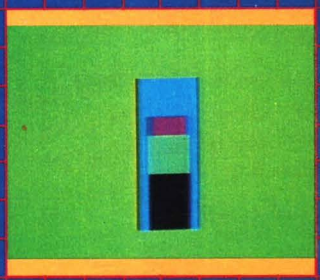
Cat. No. 62-2076

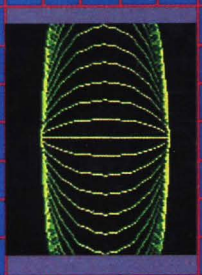
by

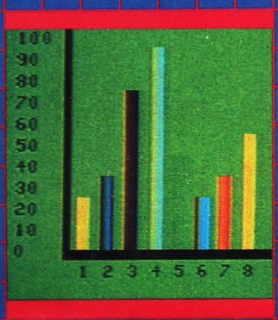
William Barden, Jr.

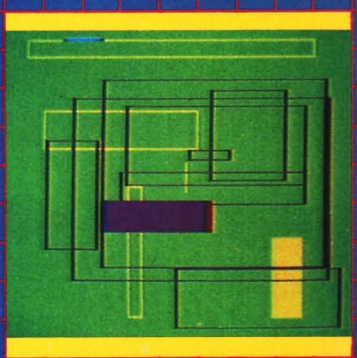












COLOR COMPUTER GRAPHICS

William Barden, Jr.

COLOR COMPUTER GRAPHICS

by

William Barden, Jr.

Radio Shack
A Division of Tandy Corp.
Ft. Worth, Texas 76102

Design and Production by Ellen Klempner.
Editorial Supervision by Tom McMillan.
Illustrations by Jack Wittry.
Typeset by LeWay Composing Service.

10 9 8 7 6 5 4 3

Foreword

We've known William Barden, Jr. for several months now. A bunch of the guys and I were watching the Dodgers on the color television in my coffee shop, the BYTE TO EAT, when suddenly in walked a guy carrying what appeared to be a gray box with keys on it. He said, "You guys aren't watching this, are you?" and switched off the TV, disconnected its antenna, and hooked up the gray box. The next thing we knew, "OK" appeared on the screen in place of the ninth inning.

Well, to make a long story short, every day since then, Barden has come in and connected his gray box to our large television. For hours on end, he'll sit there watching really weird patterns that appear on the screen — circles, boxes, tiny little figures that jump up and down.

We're really glad he was able to finish the book, despite what he says in the Preface. Now, maybe next season we can watch the Dodgers again.

Max and the Guys


COMPUTE IN COLOR!!!
A WEALTH OF COMPUTER EXCITEMENT
WITHEY

Preface

Want to use your Color Computer to draw a "pie chart"? How about constructing a chessboard with all chess pieces represented in four colors? Interested in doing a precise plot of a mathematical function? How about animating figures in your Space War game?

The Color Computer can easily do all of these things with its built-in graphics capability. The Color Computer has hardware to provide up to eight colors on your display, and it can plot 49,152 points on the screen!

Hardware is nothing without software, and the Color Computer has plenty of that contained in read-only memory (ROM). Much of the Color BASIC and Extended Color BASIC is devoted to making use of the powerful graphics capabilities of the hardware. Commands are available to draw circles, rectangles, arcs, angled lines, to rotate figures, to scale figures up and down, and many, many more.

In this book we'll show you how to use the Color BASIC and Extended Color BASIC commands to do all kinds of graphics. We'll also describe some graphics capabilities that are hidden in the Color Computer.

Color Computer Graphics consists of two sections: a tutorial section in the first eight chapters and a collection of graphics techniques in the last half of the book.

The first section, "Using Color Computer Graphics Modes and Commands," discusses the built-in graphics modes and all of the graphics commands available in both Color BASIC and Extended Color BASIC.

Chapter 1 describes the *memory mapping* of the Color Computer, or how the display memory and the memory available for user programs is divided up among random-access memory (RAM). The chapter also discusses the graphics modes that allow you to display text data and what the Color Computer calls *semigraphics* modes, graphics that occupy text character positions.

Chapter 2 discusses the remaining graphics modes of the Color Computer. These are the modes that allow you to plot up to 256 by 192 points on the display in as many as four colors. Graphics *pages*, or RAM areas set aside to hold graphics display data, are also discussed.

For those of you that have only Color BASIC, Chapter 3 describes how to use the Color BASIC commands.

Chapters 4 through 8 describe all of the commands for Extended Color BASIC in detail, with many examples.

The second section of this book, "Programming Techniques for Color Computer Graphics," is a collection of graphics techniques and applications for both Color BASIC and Extended Color BASIC. There are dozens of separate descriptions, each one discussing a separate technique.

Some of the techniques described are moving dots, PAINTing, bar graphs, drawing shades of grey, hidden lines, filled-in circles, animation, and many more!

This section duplicates some of the material found in the first section of the book but is meant more as a quick reference to a desired graphic effect. The description of the technique, the sequence to follow with the proper BASIC format, a practical example, and notes on the technique are provided for each technique.

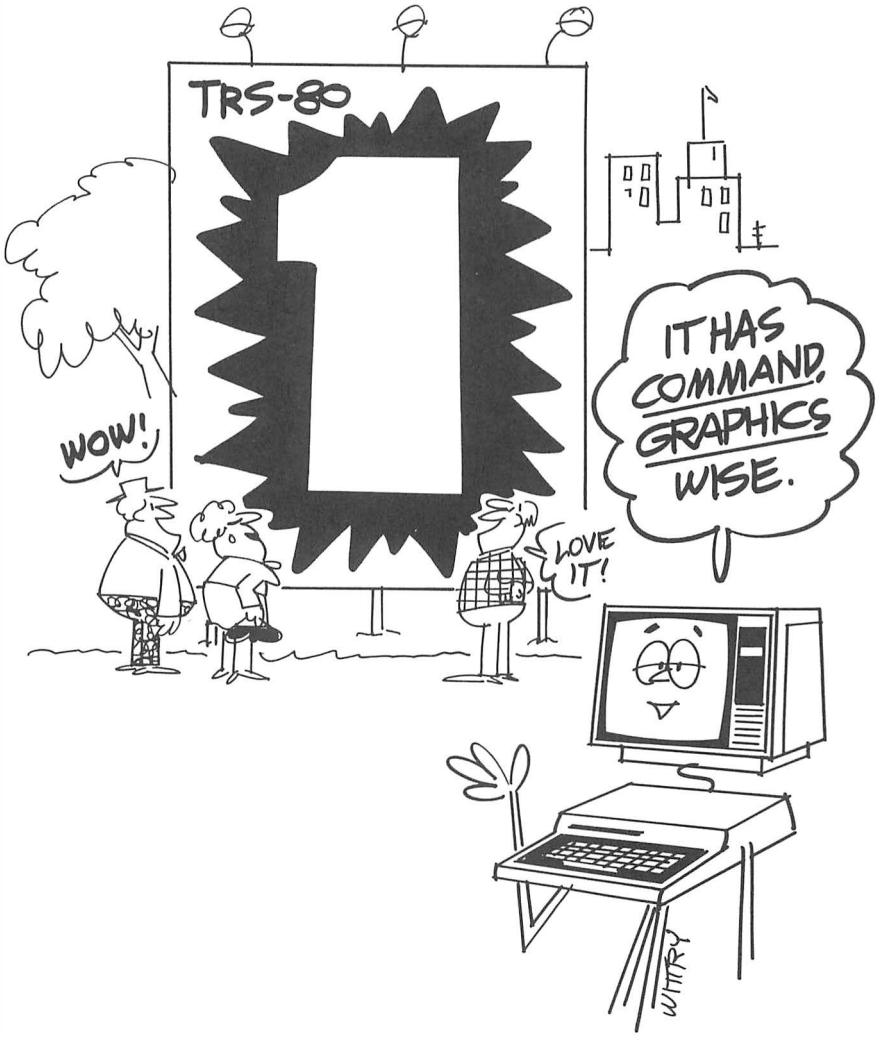
The graphics capabilities of the Color Computer are excellent. It's really just a matter of learning them then getting some practice in application. *Color Computer Graphics* should help you to do both. Now you can see your computer dreams in technicolor!

To Max and the guys at the BYTE TO EAT, none of whom gave a shred of encouragement on this book.

To Steve Mussatti.

Table of Contents

COLOR COMPUTER GRAPHICS


TRS-80
WOW!
I LOVE IT!
IT HAS COMMAND, GRAPHICS WISE.
WITTRY

SECTION I

Using Color Computer Graphics Modes and Commands


TOLLWAYS
CHICAGO
ROUTES
PARIS
POLAND
CLEVELAND
BOSTON
HONG KONG
MEMORIZE MAPS- WHAT A MODE OF LIVING!!
WITTRY

CHAPTER 1

Color Computer Memory Mapping and Semigraphics Modes

To get a clear picture of Color Computer graphics, we must discuss two topics: the memory mapping of the Color Computer and the graphics modes that are available. These concepts are essential to a complete understanding of Color Computer graphics. If some of this material seems difficult, don't hesitate to read ahead in other chapters and come back to the first two chapters later. The chapters get easier as they go on!

Memory mapping refers to the way the memory space in the Color Computer is divided — what portion is read-only memory (ROM), what portion is random-access memory (RAM), and what portion is dedicated to other functions. The memory mapping is tied into the way the Color Computer was engineered and designed.

The graphics modes are also built into the hardware of the Color Computer. A special *graphics generator* chip controls the memory modes. It's not necessary to understand how the graphics generator chip functions, so we won't be giving you a course in computer design. (For those who *are* interested in computer design, read the sequel to this book, *The Color Computer Meets the Mad Computer Scientist*.)

MEMORY MAPPING

The 6809 microprocessor in the Color Computer can normally use up to 65,536 bytes of memory. That *address space* is divided up by the system designer into ROM, RAM, and input-output as shown in Figure 1.1.

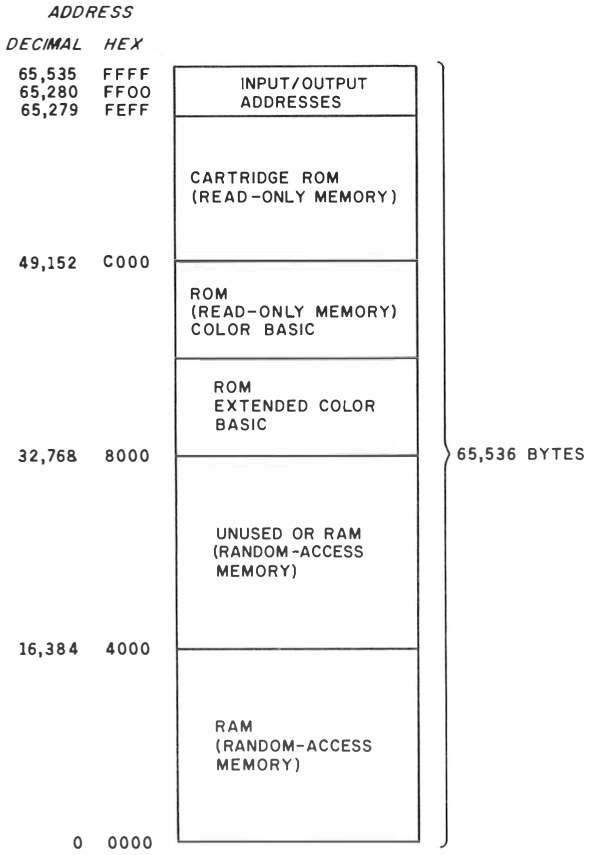


Figure 1.1: Color Computer address space.

Bits, Bytes, and Memory

Each of the 65,536 bytes of memory is made up of eight *bits*, as shown in Figure 1.2. Bit is a contraction of binary digit. A binary digit can hold an on or off condition, represented by a 1 or 0.

Each byte of memory can hold binary values of 00000000 through 11111111, which represent decimal values of 0 through 255. You won't have to become proficient in manipulating binary data to do Color Computer graphics, but you should understand the relationship of bits, bytes, and memory. We've included Appendix I to help in converting decimal and binary values.

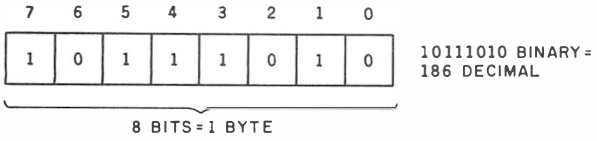


Figure 1.2: Bits in memory.

ROM Memory

ROM is used to hold the BASIC interpreter program. The BASIC interpreter is put into ROM because the contents of ROM are never destroyed when power is turned off on the system. Since BASIC programs are most often run in the Color Computer, it's a logical choice for the firmware.

Another portion of the memory space is used to hold cartridge ROM. Cartridge ROM is similar to the BASIC interpreter ROM, except that it holds precanned applications programs such as games, programmed instructions, or business packages.

RAM Memory

RAM is used for holding user BASIC programs and user machine-language programs, for storage of variables used by the BASIC interpreter, and for video memory. RAM is volatile and its contents are completely destroyed whenever power is turned off, so RAM must be reloaded with programs or data by reading in cassette or disk files.

Input-Output Addresses

Certain portions of the 65,535 bytes of addressing space are dedicated to input-output addresses. These addresses look like memory locations. They are decoded by the Color Computer hardware as a means to communicate with Color Computer ports such as serial (RS-232) input-output, joysticks, and sound generation.

VIDEO MEMORY

Contrary to reports, video memory is *not* total recall of every plot of "I Love Lucy." Video memory is the portion of normal RAM used for screen display.

The area from memory location 1024 up is devoted to video

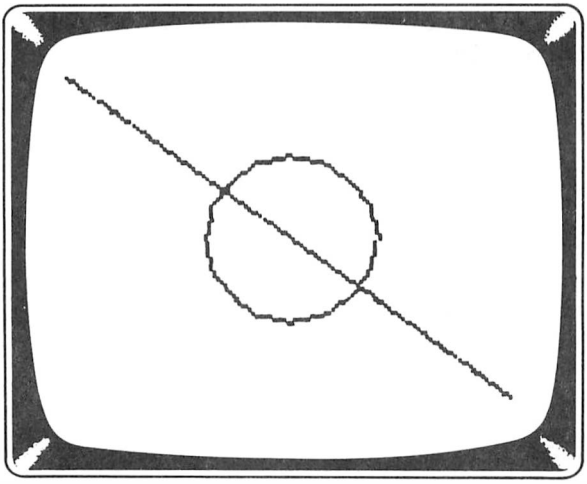
memory. The minimum size of this area for a single screen's worth of data is 512 bytes; the maximum size of this area is 6,144 bytes. Naturally, the more video memory required, the less area that will be available for user programs.

RESOLUTION

The minimum video memory area occurs when a *low-resolution* display mode is used. In a low-resolution mode, the display is made up of a small number of blocks of data. The lowest resolution mode displays 64 blocks horizontally by 32 blocks vertically, for a total of 32 × 64 or 2048 blocks. These 2048 blocks can each be represented by two bits. Since four groups of two bits can be squeezed into each byte, the total number of video memory bytes required is 2048/4, or 512 bytes.

The maximum video memory area occurs when a high-resolution display mode is used. In a high-resolution mode, the display is made up of many smaller blocks of data. The highest resolution mode displays 256 blocks horizontally by 192 blocks vertically for a total of 49,152 blocks. These 49,152 blocks can each be represented by one bit. Since we can squeeze eight bits into one byte, the total number of video memory bytes required is 49,152/8 or 6144 bytes.

How different grades of resolution affect a display is shown in Figure 1.3, which illustrates the same scene for coarse to fine resolution.



LEAST
RESOLUTION

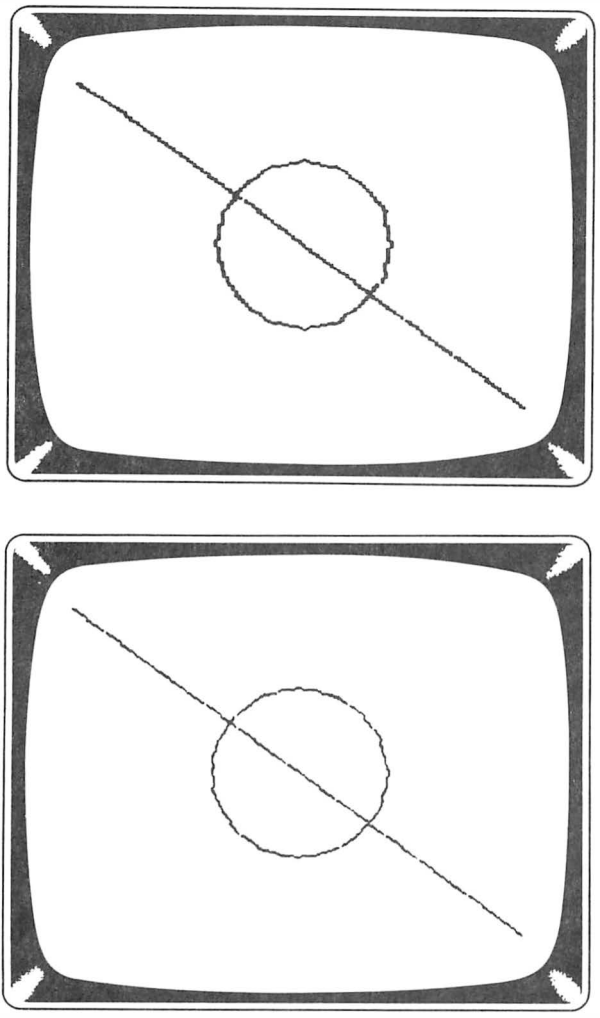


Figure 1.3: Resolution in graphics.

TRADEOFFS IN RESOLUTION

When should low resolution be used and when should high resolution be used? In general, low resolution requires less memory while high resolution requires greater memory. Since video memory shares the same RAM space as BASIC and other programs, the more resolution required, the less RAM memory that is available for BASIC programs.

Another thing to consider is the amount of color required. Generally, the lower the resolution, the greater the number of colors

permitted. This is because there is only so much RAM memory to go around. We can't specify 49,152 blocks of display and permit eight colors for each block. To do so would require 16,384 bytes of RAM!

Another factor is speed. It takes eight times as long to process 16,384 blocks of display data as it does to process 2,048 blocks of display data! If you are displaying animation on the video, it might be better to use lower resolution and get more frames per second.

There are tradeoffs in considering the resolution to use. We'll explain more of these in detail as we go through the book.

Blocks vs. Pixels

We've been referring to blocks of data on the display. The common term for such a block is a *picture element,* shortened to *pixel.* In the rest of this book we'll use the term pixel in place of block, but there's nothing mysterious or esoteric about it. The number of pixels in a display may be anywhere from 2048 (64 horizontal by 32 vertical) to 49,152 (256 horizontal by 192 vertical).

GRAPHICS MODE

We're now ready to tackle the various graphics modes that are available on the Color Computer. There are 16 of these modes, ranging from display of alphanumeric characters to high-resolution graphics displays of 256 by 192 pixels. Why so many modes? The chief reason is that the Color Computer uses a special-purpose semiconductor integrated circuit called the video display generator, or VDG. This chip is a general-purpose display device, and we get all of these modes for free! Some of them are more useful than others. We'll describe all of them and allow your experience to determine which will be the most useful.

There are three basic display modes: alphanumeric, alpha semigraphic, and graphic. *Alphanumeric mode* allows display of text characters. The character set is defined by an internal character generator in the VDG chip which produces 5-by-7 dot-matrix characters as shown in Table 1.1.

@	␣ blank
A	!
B	"
C	#
D	$
E	%

F	&
G	'
H	(
I	)
J	*
K	+
L	,
M	–
N	.
O	/
P	0
Q	1
R	2
S	3
T	4
U	5
V	6
W	7
X	8
Y	9
Z	:
[	;
\	<
]	=
↑	>
←	?

Table 1.1: Character Set of VDG

The *alpha semigraphic* mode is a low-resolution mode that is related to the alphanumeric mode. The *graphic* mode is a full graphic mode that generally permits higher resolution.

All of the modes can be selected under program control. Most of them can be selected very easily under Extended Color BASIC. We'll start with the simplest mode and work upwards, showing you the BASIC commands which set the mode, the appearance of the mode, and the video memory affected.

ALPHANUMERIC MODE (INVERTED)

This is the normal mode of the Color Computer system which is set on power on or reset, or upon return to the BASIC interpreter after execution of a BASIC program.

This mode is shown in Figure 1.4. The border is always black. There are 32 characters per line and 16 lines per screen, making for a total of 512 character positions.

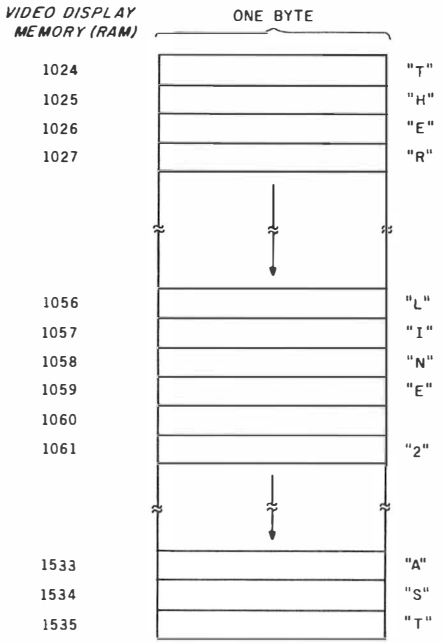


Figure 1.4: Alphanumeric mode screen format.

Each character position is made up of 8 by 12 pixels. Since a character generated by the VDG character generator is 5 by 7 pixels, there is a buffer area of 2 pixels on top, 3 pixels on the bottom, 2 pixels on the right, and 1 pixel on the left, as shown in Figure 1.5.

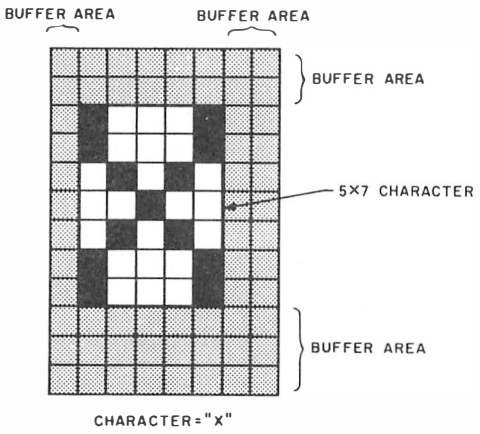


Figure 1.5: Character position pixels.

The character in each character position is defined by one byte in the video display memory. There is a one-for-one relationship between a video display memory byte and a character position on the screen, as shown in Figure 1.4. The code in each video display memory byte is the code shown in Table 1.2. The code is a modification of the American Standard Code for Information Interchange (ASCII), which is a standard computer code for alphabetic, numeric, and special characters.

Character	Decimal	Hex	Character	Decimal	Hex
@	64	40	♭	96	60
A	65	41	!	97	61
B	66	42	"	98	62
C	67	43	#	99	63
D	68	44	$	100	64
E	69	45	%	101	65
F	70	46	&	102	66
G	71	47	'	103	67
H	72	48	(	104	68
I	73	49	)	105	69
J	74	4A	*	106	6A
K	75	4B	+	107	6B
L	76	4C	,	108	6C
M	77	4D	–	109	6D
N	78	4E	.	110	6E
O	79	4F	/	111	6F
P	80	50	0	112	70
Q	81	51	1	113	71
R	82	52	2	114	72
S	83	53	3	115	73
T	84	54	4	116	74
U	85	55	5	117	75
V	86	56	6	118	76
W	87	57	7	119	77
X	88	58	8	120	78
Y	89	59	9	121	79
Z	90	5A	:	122	7A
[	91	5B	;	123	7B
\	92	5C	<	124	7C
]	93	5D	=	125	7D
↑	94	5E	>	126	7E
←	95	5F	?	127	7F

Table 1.2: Character Codes in Color Computer.

Figure 1.6 shows a typical display of alphanumeric data on the video screen and the corresponding characters in video memory. Note that all values in video memory are less than 128. Values that are less than 128 have bit 7, the *most significant bit,* set to a zero.

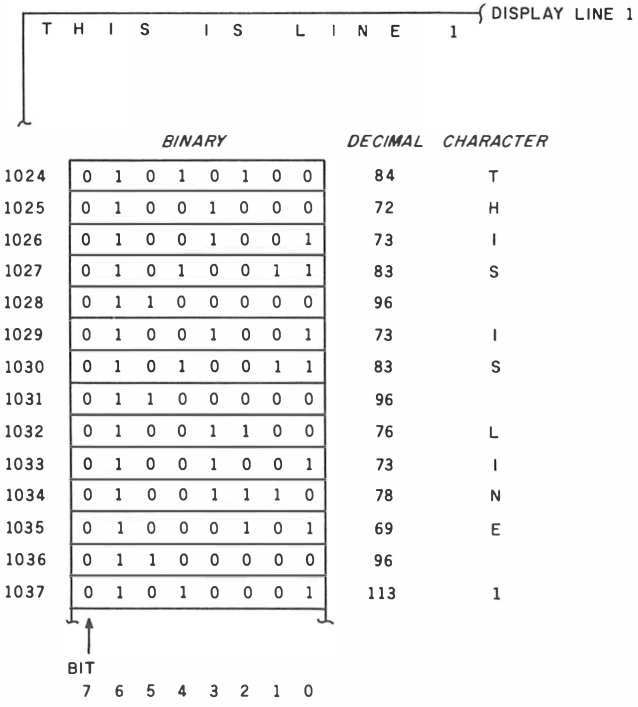


Figure 1.6: Alphanumeric data display.

Whenever bit 7 of any byte in video display memory is set to zero, the VDG will display an alphanumeric character in the corresponding character position on the screen.

To see how this mode works, run the following Extended Color BASIC program to fill all of video display memory with an "A" character. You'll see the screen fill up with the "A" character.

```
100 REM FILL SCREEN WITH "A" CHARACTERS
110 FOR VM=1024  TO 1024+511
120 POKE VM,65
130 NEXT VM
140 GOTO 140
```

The 65 here is the ASCII code for "A". The 1024 is the starting address of video memory from Figure 1.1. Normally you would not have to know this starting address to do graphics work, but, on the other hand, it won't hurt you either!

ALPHANUMERIC MODE (NON-INVERTED)

This alphanumeric mode is related to the alphanumeric mode inverted. In the alphanumeric inverted mode, the character is black on green. In the non-inverted alphanumeric mode, the character is green on black. Table 1.3 shows the character codes for this mode. Note that there are 64 fewer character codes than for the inverted mode, indicating that bit 6 (binary weight of 64) is reset.

Character	Decimal	Hex	Character	Decimal	Hex
@	0	00	b	32	20
A	1	01	!	33	21
B	2	02	"	34	22
C	3	03	#	35	23
D	4	04	$	36	24
E	5	05	%	37	25
F	6	06	&	38	26
G	7	07	'	39	27
H	8	08	(	40	28
I	9	09	)	41	29
J	10	0A	*	42	2A
K	11	0B	+	43	2B
L	12	0C	,	44	2C
M	13	0D	–	45	2D
N	14	0E	.	46	2E
O	15	0F	/	47	2F
P	16	10	0	48	30
Q	17	11	1	49	31
R	18	12	2	50	32
S	19	13	3	51	33
T	20	14	4	52	34
U	21	15	5	53	35
V	22	16	6	54	36
W	23	17	7	55	37
X	24	18	8	56	38
Y	25	19	9	57	39
Z	26	1A	:	58	3A
[	27	1B	;	59	3B
\	28	1C	<	60	3C

]	29	1D	=	61	3D
↑	30	1E	>	62	3E
←	31	1F	?	63	3F

Table 1.3: Non-Inverted Character Codes

The VDG automatically performs this reverse-field operation for any character which has bit 6 reset, as shown in Figure 1.7. Those readers with Extended Color BASIC can run the following program to show how video memory bytes can be intermixed with both inverted and normal characters.

```
100 REM FILL SCREEN WITH INVERTED AND NON-
    INV CHARACTERS
110 FOR VM=1024 TO 1024+511 STEP 2
120 POKE VM,1
130 POKE VM+1,65
140 NEXT VM
150 GOTO 150
```

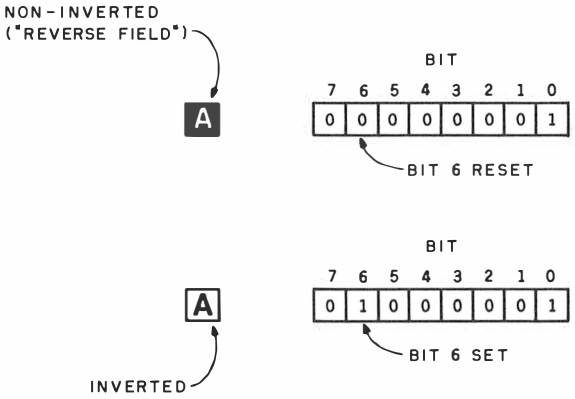
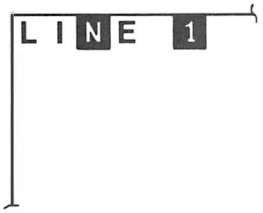


Figure 1.7: Reverse field display.

The BASIC interpreter (Color or Extended Color) normally enters a non-inverted character code in the proper video memory location. If the SHIFT and 0 keys are held down simultaneously, then the BASIC interpreter switches from the current mode, inverted or non-inverted, to the other mode. Try entering some simple text on the screen and use the SHIFT and 0 key to see this effect. Figure 1.8 shows the result in memory.



	BINARY								DECIMAL	CHARACTER
1024	0	1	0	0	1	1	0	0	76	L
1025	0	1	0	0	1	0	0	1	73	I
1026	0	0	0	0	1	1	1	0	14	N (NON-INVERTED)
1027	0	1	0	0	0	1	0	1	69	E
1028	0	1	1	0	0	0	0	0	96	
1029	0	0	1	1	0	0	0	1	49	1 (NON-INVERTED)

Figure 1.8: Inverted and non-inverted displays.

SEMIGRAPHICS MODES

There are four *semigraphics modes* in the Color Computer. I'd like to tell you that the semigraphics modes were named after that English computer scientist, Alfred E. Semigraphics. However, this confusing term really comes from the fact that all semigraphics modes use the same character position as the alphanumerics mode, a block of 8 by 12 pixels.

It is possible to intermix text and graphics in some of the semigraphics modes, unlike the true graphics modes.

Depending upon the mode, the semigraphics modes divide the block up into 4, 6, 8, 12, or 24 elements as shown in Figure 1.9. The semigraphics modes are called semigraphic 4, semigraphic 6, semigraphic 8, semigraphic 12, and semigraphic 24, each name reflecting the number of elements. Each mode divides the block into 2 columns 4 pixels wide.

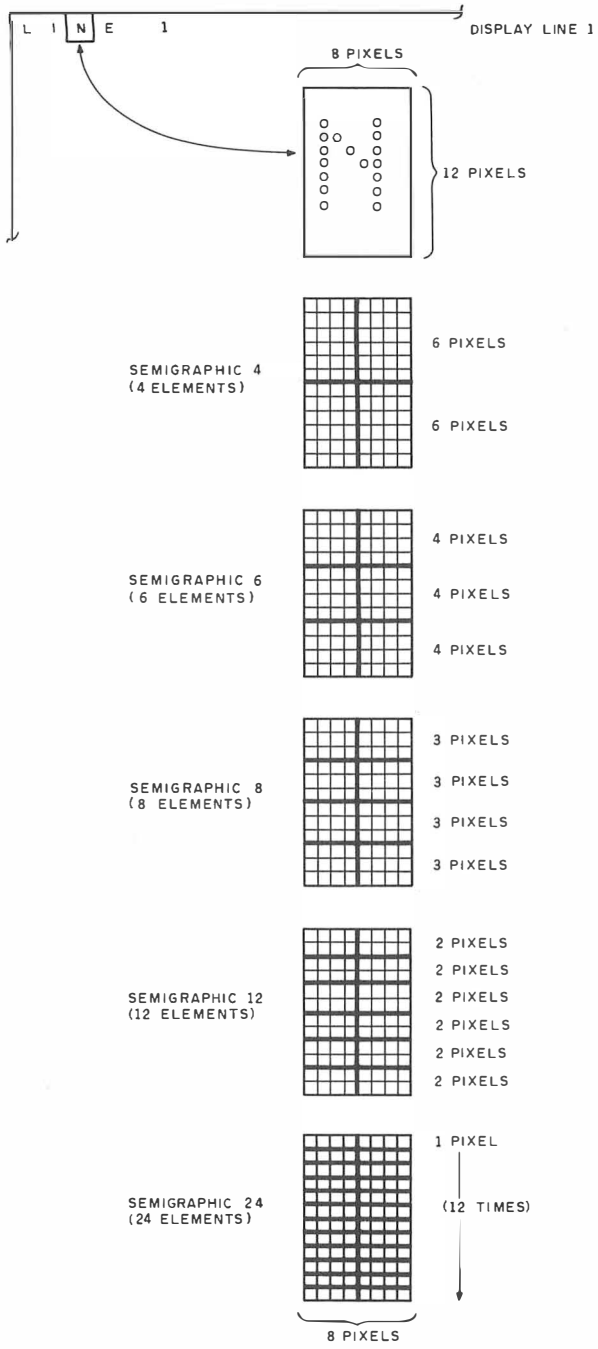


Figure 1.9: Semigraphics format.

Semigraphic 4 Mode

This mode divides up a character position into 4 elements. Each character position of the 32 character positions per each of the 16 lines is represented by 1 byte in video memory. A total of 512 bytes are required in video memory. The mapping for this scheme is shown in Figure 1.10.

Each element in the character position is either on or off. The color for off elements is black. The color for on elements is one of the eight standard colors in the VDG: green, yellow, blue, red, buff, cyan, magenta, or orange.

Each byte in video memory holds a number of fields, as shown in Figure 1.11. The first bit is a 1, indicating the semigraphics mode. The next field indicates the color; it holds binary values from 000 through 111, or 0 through 7. The next 4 bits represent the on/off state of each of the four elements — a zero is off (black) while a one is on (color).

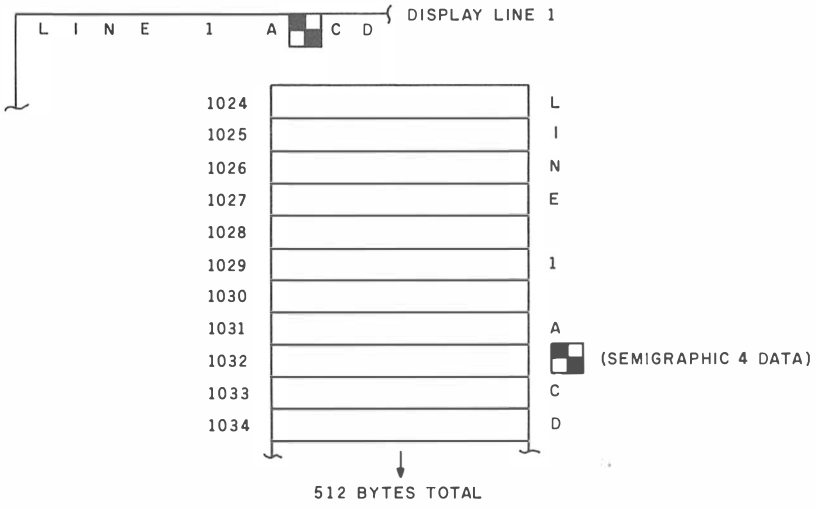


Figure 1.10: Semigraphic 4 mapping.

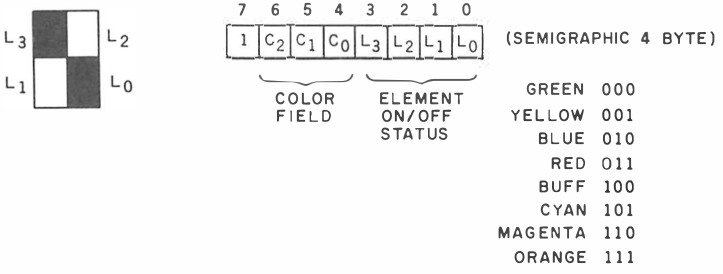


Figure 1.11: Fields in semigraphic 4 mode.

Note that in this mode all four elements have the same color; there cannot be four separate colors in each of the four elements.

This mode is the Color BASIC high-resolution mode of 64 pixels horizontally by 32 pixels vertically. It is used in the SET, RESET, and POINT commands. In these commands, an *X* value of 0 through 63 specifies the horizontal position, while a *Y* value of 0 through 31 specifies the vertical position. The following BASIC code uses the SET command to set the screen to this mode with alternating colors from 0 through 7 (green through orange).

```
100 REM SEMIGRAPHICS 4 MODE
110 C=1
120 FOR Y=0 TO 31 STEP 2
130 FOR X=0 TO 63 STEP 2
140 SET (X,Y,C) : SET (X+1,Y,C)
150 SET (X,Y+1,C) : SET (X+1,Y+1,C)
160 C=C+1 : IF C=9 THEN C=1
170 NEXT X
180 NEXT Y
190 GOTO 190
```

Color BASIC converts the *C* value in the SET command to a value of 0 through 7 by subtracting one. It then stores this value in the color field. The *X* and *Y* values from the set turn on one of the four elements in the character position block. As the color value must be the same for all four elements of the block, the program above uses the same color value for all four elements. Typical values in video memory for a small number of points are shown in Figure 1.12.

In addition to setting the most significant bit in the video memory byte associated with the character position, the BASIC interpreter sets the VDG to the semigraphics mode. This is done prior to any display work. The entire screen is either in the alphanumeric/semigraphic mode or in the graphic mode. The location associated with setting the mode is location 65314. The following code will set the semigraphic 4 mode.

```
100 A=PEEK (65314) : POKE 65314,(A AND 7)
110 POKE 65476,0 : POKE 65474,0 :
    POKE 65472,0
```

Note that if you are using BASIC, the BASIC interpreter will automatically set the alphanumeric/semigraphic mode whenever a return is made to the BASIC interpreter. You do not have to be concerned about any POKEs to the locations above.

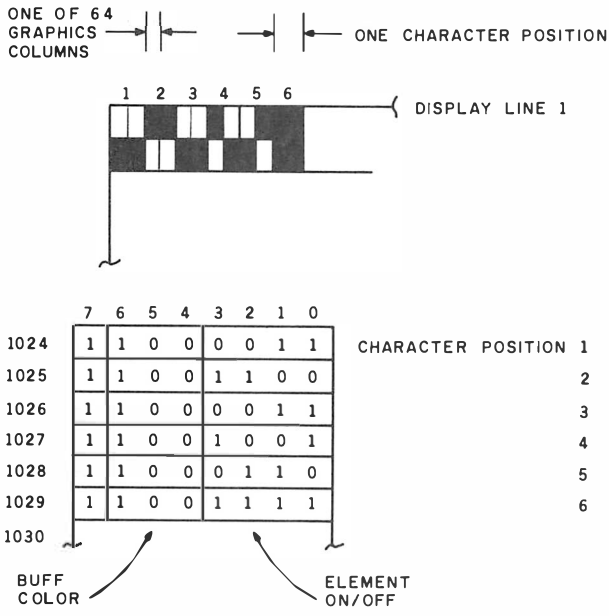


Figure 1.12: Semigraphic 4 data storage example.

Semigraphic 6 Mode

Semigraphic 6 and the remaining semigraphics modes (semigraphics 8 and semigraphics 12) are the forgotten semigraphic modes. (Years ago, these modes were implemented in the VDG. Paperwork defining the modes was shuffled and lost, however, and neither the Color BASIC interpreter nor the Extended Color BASIC interpreter contain commands to set the modes.) If the modes are used, they must be set by POKEs to address 65314 and the addresses in the range 65472 through 65477.

In addition, the video memory area must be set to the proper byte configurations. This is not difficult in the semigraphic 6 mode, where one byte is used for each character position. But it's somewhat more involved in semigraphics 8 and semigraphics 12 modes. We'll discuss each of the modes in detail, but remember that you may never want to use any semigraphic mode other than semigraphic 4, and you may want to do that automatically through the BASIC interpreter.

Semigraphic 6 mode divides up a character position into six elements. Each of the 32 character positions per line in each of the 16 lines is represented by one byte in video memory. A total of 512 bytes are required in video memory. The mapping for this scheme is shown in Figure 1.13.

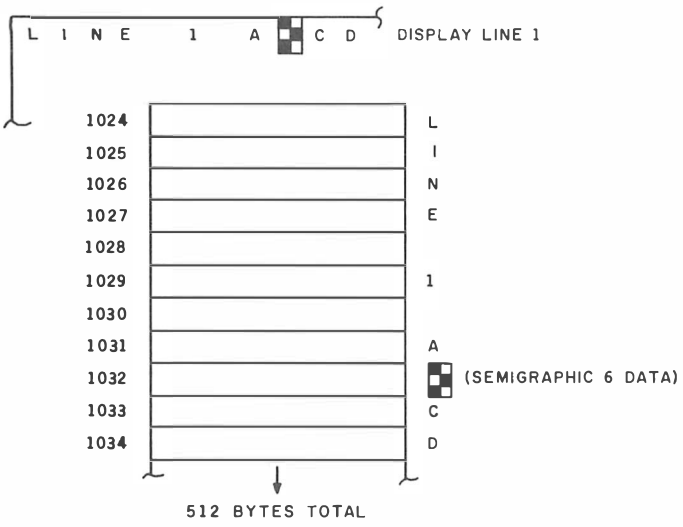


Figure 1.13: Semigraphic 6 mapping.

This mode makes two sets of two colors available. The border color is always black, as it is for all semigraphics modes. One set of colors that can be chosen is green, yellow, blue, and red. A second set is buff, cyan, magenta, and orange. As in the semigraphic 4 mode, all on elements have the same color, while all off elements are black.

Each byte in video memory holds a number of fields, as shown in Figure 1.14. The first field is made up of two bits that determine the color as shown in the figure. The next six bits determine the on/off state of each of the six elements in the character position; a one is on while a zero is off.

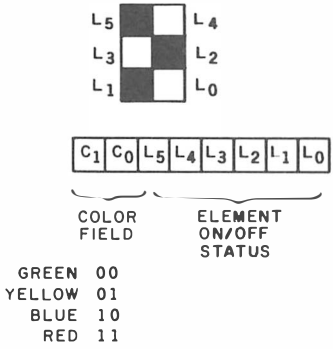


Figure 1.14: Fields in semigraphic 6 mode.

The selection of the color set is made by setting or resetting bit 4 of address 65314. A zero in bit 4 generates magenta or orange, depending upon the color bit in the video memory byte. A one in bit 4 generates blue or red, depending upon the color bit in the video memory byte.

To utilize this mode in BASIC, set or reset the 65314 address bit, change the 65472–65477 addresses, and then fill the 512 bytes of video memory with the graphics you want displayed. The following program uses an on color of blue and sets a checkerboard pattern on the video display.

```
100 REM SEMIGRAPHICS 6 MODE
110 A=PEEK (65314) : POKE 65314,(A AND
    7)+8+16 'USE 16 OR 0
120 POKE 65476,0 : POKE 65474,0 :
    POKE 65472,0
130 FOR VM= 1024 TO 1024+511
140 IF (VM AND 32)=0 THEN POKE VM,166
    ELSE POKE VM,153
150 NEXT VM
160 GOTO 160
```

The appearance of the display is shown in Figure 1.15.

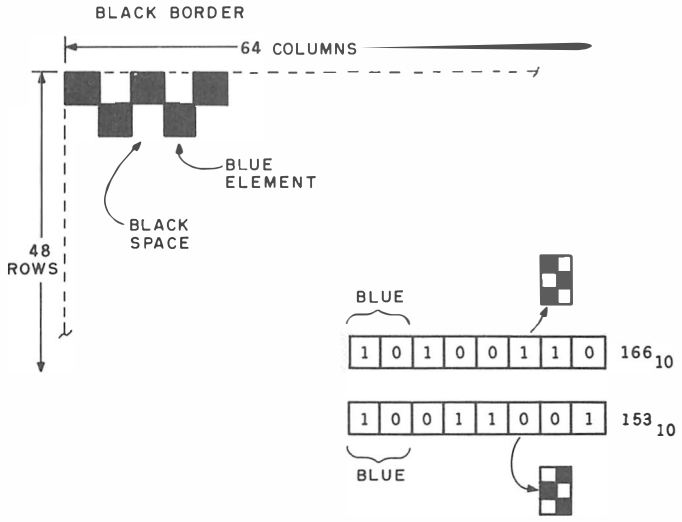


Figure 1.15: Semigraphic 6 example.

Semigraphics 8, 12, and 24 Memory Requirements

Two preceding semigraphic modes used 1 byte in memory for all graphics elements and required 512 bytes of memory. The semigraphics 8, 12, and 24 modes require 4, 6, and 12 bytes of memory respectively for each set of graphics elements. This increases the video memory requirements to 2048, 3072, and 6144 respectively.

There is 1 byte of memory for each of the rows in the graphic character position for these three modes. The memory *mapping* of each character position is somewhat involved. The byte for each row is displaced 32 bytes from the last row rather than being adjacent in memory. We'll give you examples in the discussions following.

Semigraphics 8 Modes

This mode divides the character position into 8 elements, each element being 4 by 3 pixels, as shown in Figure 1.16. Each character position of 8 elements is represented by 4 bytes in video display memory, making a total of 4×512 bytes, or 2048 bytes of video memory.

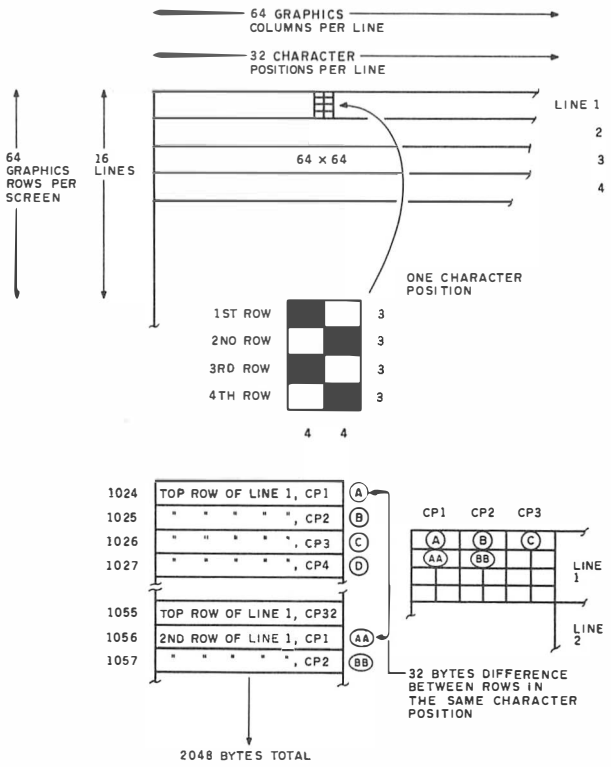


Figure 1.16: Semigraphic 8 mapping.

The relationship between the video memory bytes and the elements of the character position is shown in Figure 1.17. There is a memory byte for each row. Each byte of the 4 memory bytes has bit 7, the most significant bit, set. The next 3 bits of each byte are the color for the row. The remainder of the bytes hold the on/off state for each of the 8 elements, 2 elements being specified per byte.

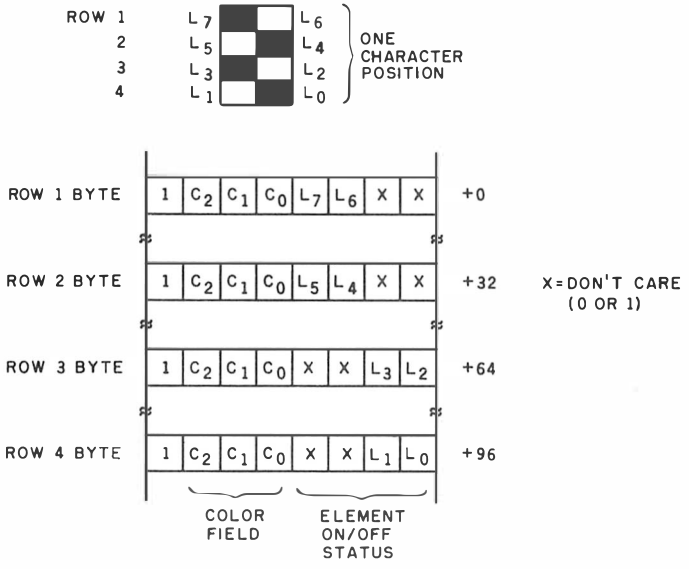


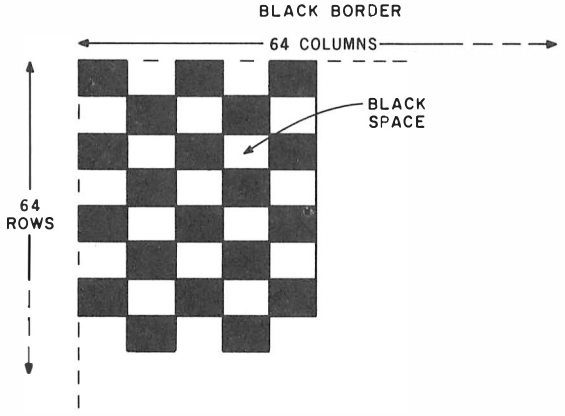
Figure 1.17: Semigraphic 8 format.

Eight colors can be specified for each row. It is possible to have green in row 0, red in row 1, orange in row 2, and buff in row 3, for example. As in the other semigraphics mode, the border is always black.

As we mentioned earlier, this mode is not implemented in either Color BASIC or Extended Color BASIC. It must be invoked by setting up the proper data in memory and issuing a POKE to address 65314 and POKEs to addresses 65472–65477.

The program below shows how the mode is used. A POKE to 65314 of 0 and a POKE 65475,1 sets the mode. We've chosen to store a checkboard pattern of 64 elements by 64 elements. Each new on element will be set to a different color. There will be off (black) elements between each on element. The appearance and layout is shown in Figure 1.18.

```
100 REM SEMIGRAPHICS 8 MODE
110 A=PEEK (65314) : POKE 65314,(A AND 7)
120 POKE 65476,0 : POKE 65475,1 : POKE
    65472,0
130 C=0 : DT=128+0+8 : N=8
140 FOR LN=0 TO 63
150 FOR CP=0 TO 62 STEP 2
160 POKE 1024+LN*32+CP/2,DT
170 NEXT CP
180 C=C+16 : IF C=128 THEN C=0
190 N=N/2 : IF N<1 THEN N=8
200 DT=128+C+N
210 NEXT LN
220 GOTO 220
```



GREEN	1	0	0	0	1	0	0	0
YELLOW	1	0	0	1	0	1	0	0
BLUE	1	0	1	0	0	0	1	0
RED	1	0	1	1	0	0	0	1
BUFF	1	1	0	0	1	0	0	0
CYAN	1	1	0	1	0	1	0	0
MAGENTA	1	1	1	0	0	0	1	0
ORANGE	1	1	1	1	0	0	0	1
		COLOR (REPEATS)			ON BIT (REPEATS)			

Figure 1.18: Semigraphic 8 example.

The program above uses two loops. The outer loop (FOR LN = 0 TO 64) increments the row number LN from 0 through 63. The inner loop (FOR CP = 0 TO 31) increments the column number CP from 0 through 63. A total of 64 × 32 or 2048 passes are made through the loops, representing 2048 POKEs for the 2048 bytes of video memory.

For each byte of video memory, a POKE of DT is done. DT is the byte for the current row and 2 columns. It is equal to 128 (most significant bit is always a one) + *C* (the current color value) + *N* (the current bit to be set). The color value *C* changes by one for each new row, cycling through 0 through 7. The bit value changes to produce the checkerboard pattern.

Prior to the POKEs, the semigraphics 8 mode is set by POKEs to 65314 and 65472–65477.

Semigraphics 12 Mode

This is the fourth semigraphics mode. It divides the character position into 12 elements as shown in Figure 1.19. Each element is 4 by 2 pixels, as shown in the figure. Each character position of 12 elements is represented by 6 bytes in video display memory, making a total of 6 × 512 bytes or 3072 bytes of video memory.

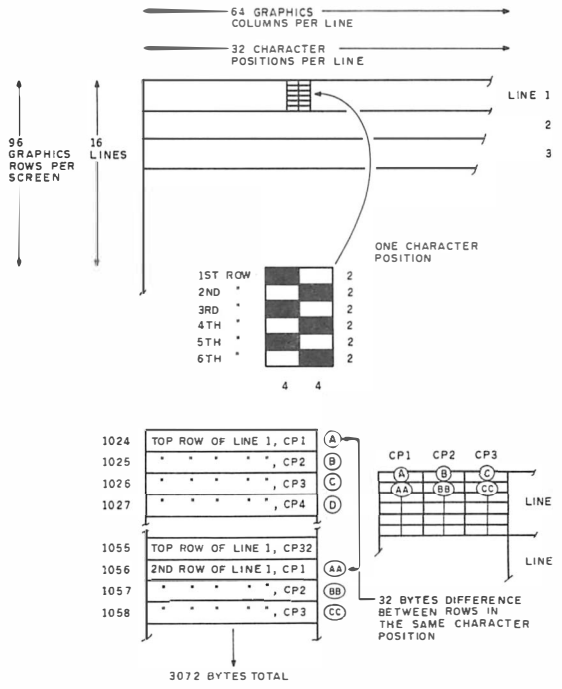


Figure 1.19: Semigraphic 12 mapping.

As Figure 1.20 shows, there is a memory byte for each row, similar to the mapping of semigraphics 8 mode. Each of the 6 bytes has bit 7, the most significant bit set. The next 3 bits of each byte are the color for the row. The remainder of the bytes hold the on/off state for each of the 12 elements, 2 elements being specified per byte.

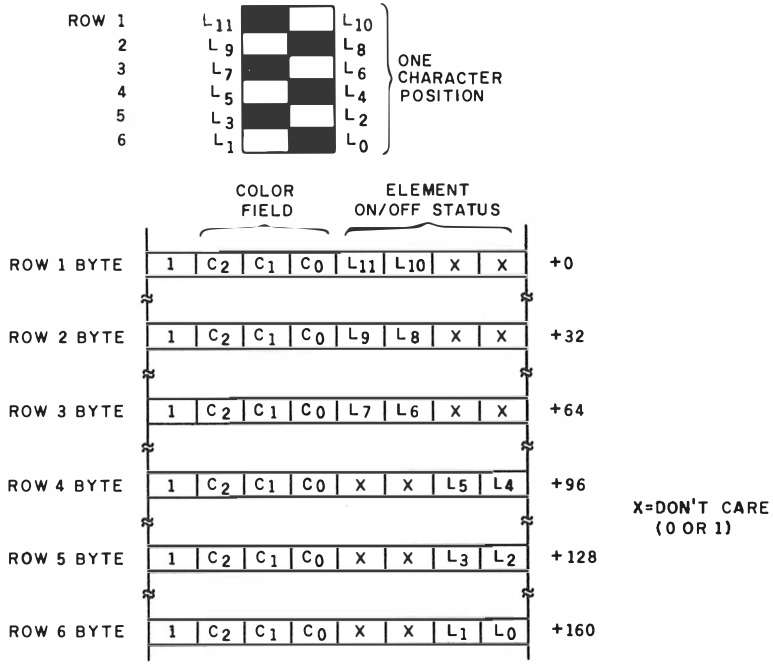


Figure 1.20: Semigraphic 12 format.

Eight colors can be specified for each row. The border is always black. Again, this mode is not implemented in either Color BASIC or Extended Color BASIC but must be set up by storing the proper data in memory and issuing 2 POKEs.

The program below shows how the mode is used. POKEs to 65314 and 65472–65477 set the mode. We'll perform a similar program to the one for semigraphics 8 and store a checkerboard pattern of 64 by 96 elements. Each new on element will be a different color. There will be off elements between each element. The appearance and layout is shown in Figure 1.21.

```
100 REM SEMIGRAPHICS 12
110 A=PEEK (65314) : POKE 65314,(A AND 7)
120 POKE 65477,1 : POKE 65474,0 : POKE
    65472,0
130 C=0 : DT=128+0+10 : N=10
140 FOR LN=0 TO 95
150 FOR CP=0 TO 32
160 POKE 1024+LN*32+CP,DT
170 NEXT CP
180 C=C+16 : IF C=128 THEN C=0
190 IF N=10 THEN N=5 ELSE N=10
200 DT=128+C+N
210 NEXT LN
220 GOTO 220
```

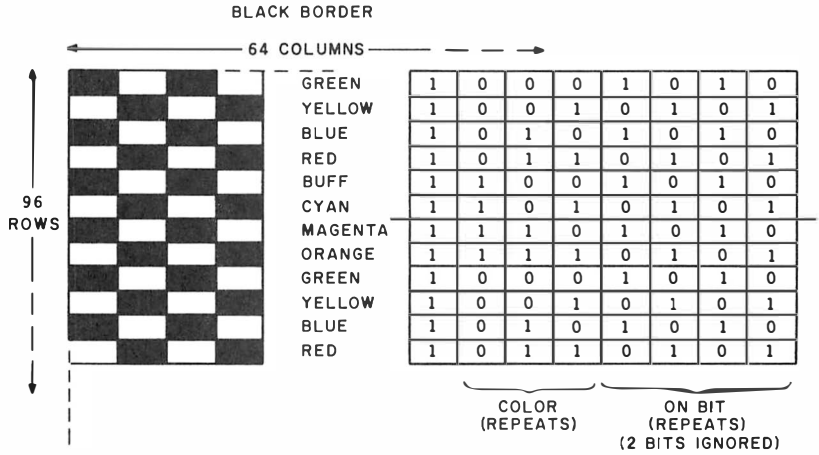


Figure 1.21: Semigraphic 12 example.

This program is very similar to the program for semigraphics 8. (What the heck, it saves wasted energy in being creative . . .) It operates from rows 0 through 95, however, reflecting the fact that there are 96 vertical rows. The color variable *C* and bit on value *N* change as before, except that *N* is flipped back and forth between a 1010 and 0101 configuration. (Only 2 of the on bits are used in each 1010 or 0101 value.)

Semigraphics 24 Mode

This is the super semigraphics mode. It divides the character position into 24 elements, each element being 2 by 1 pixels as shown in Figure 1.22. Each character position of 24 elements is represented by 12 bytes in video display memory, making a total of 12×512 bytes, or 6144 bytes of video memory.

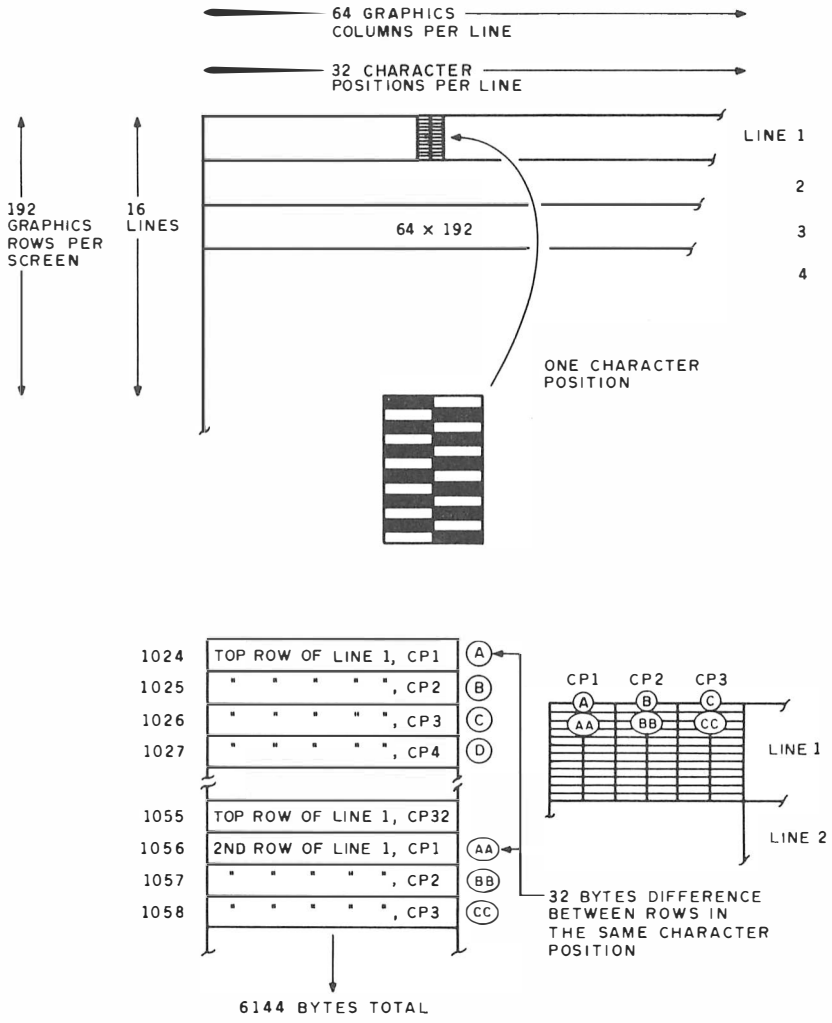
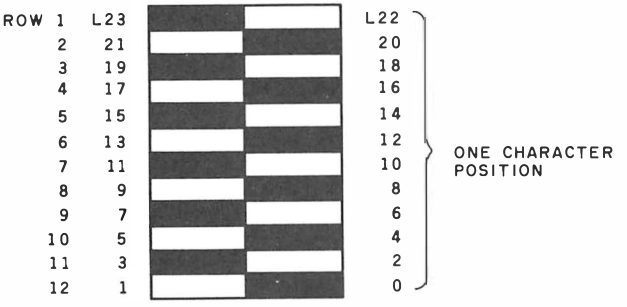


Figure 1.22: Semigraphic 24 mapping.

Each row with 2 columns is represented by 1 memory byte as in the preceding two semigraphics modes, as shown in Figure 1.23. Each row may be one of eight colors, as in the other modes. Again, the format for each byte is virtually identical to the other modes, as shown in the figure. The border is black as before. Again, this is a forgotten mode as far as the BASIC interpreters.



ROW		BYTE								
"	2	"	1	C_2	C_1	C_0	L_{23}	22	X	X	+0
"	3	"	1	C_2	C_1	C_0	21	20	X	X	+32
"	4	"	1	C_2	C_1	C_0	19	18	X	X	+64
"	5	"	1	C_2	C_1	C_0	17	16	X	X	+96
"	6	"	1	C_2	C_1	C_0	15	14	X	X	+128
"	7	"	1	C_2	C_1	C_0	13	12	X	X	+160
"	8	"	1	C_2	C_1	C_0	X	X	11	10	+192
"	9	"	1	C_2	C_1	C_0	X	X	9	8	+224
"	10	"	1	C_2	C_1	C_0	X	X	7	6	+256
"	11	"	1	C_2	C_1	C_0	X	X	5	4	+288
"	12	"	1	C_2	C_1	C_0	X	X	3	2	+320
			1	C_2	C_1	C_0	X	X	1	0	+352

THESE BYTES ARE NOT CONTIGUOUS!

COLOR FIELD

ELEMENT ON/OFF STATUS

X = DON'T CARE

Figure 1.23: Semigraphic 24 format.

The program below is almost identical to the semigraphics 12 mode. (Writers are lazy by nature.) In this case, however, the number of rows goes from 0 through 191. As in the other examples, the mode is set by POKEs to 65475 and 65472–65477.

```
100 REM SEMIGRAPHICS 24
110 A=PEEK (65314) : POKE 65314,(A AND 7)
120 POKE 65477,1 : POKE 65475,1 : POKE
    65472,0
130 C=0 : DT=128+0+10
140 FOR LN=0 TO 191
150 FOR CP=0 TO 32
160 POKE 1024+LN*32+CP,DT
170 NEXT CP
180 C=C+16 : IF C=128 THEN C=0
190 IF N=10 THEN N=5 ELSE N=10
200 DT=128+C+N
210 NEXT LN
220 GOTO 220
```

USING THE SEMIGRAPHICS MODES

The semigraphics 8, 12, and 24 modes are somewhat cumbersome to use because there are no BASIC functions that implement them. As we'll see in the next chapter, however, they do offer one advantage over the true graphics modes — they provide eight colors with a maximum resolution of 64 by 192, which is a fairly fine resolution. Keep them in mind for possible use in BASIC or assembly-language programs if the graphics modes do not provide enough colors for your application.

Let's emphasize again that the semigraphics 6, 8, 12, and 24 modes are much harder to use than the graphics modes handled automatically in the BASIC interpreters. You may not want to bother with POKEing into the address locations. This information is presented for your amazement and amusement only!

In the next chapter we'll look at the remaining graphics modes, the true graphics modes. These are implemented in Extended BASIC and can be used much more easily than the semigraphics modes covered in this chapter.

NOTES


'GULP'
IT'S TRUE OFFICER.
HMMM
WITTEY

CHAPTER 2

True Graphics Modes

In this chapter we'll look at the *true* graphics modes of the Color Computer. Many of the modes are supported by the Extended Color BASIC but can be used with Color BASIC if you have enough random-access memory (RAM) for the video memory area and are willing to set up the data in video memory in your own BASIC code.

There are 8 different graphics modes in the Color Computer, ranging from a four-color 64 by 64 mode, which requires 1024 bytes of video memory, to a two-color 256 by 192 mode, which requires 6,144 bytes of video memory. Generally, the higher the resolution, the fewer colors and the greater the amount of video memory required. The number of colors is limited to either two or four, plus a border color. The available graphics modes are shown in Figure 2.1.

As you can see from the figure, Extended Color BASIC supports the 5 highest resolution modes. We'll discuss each of the modes in detail, showing you how to use them in Extended BASIC and in your own code. Before we get into the detailed discussion, however, let's look at some of the characteristics of these modes.

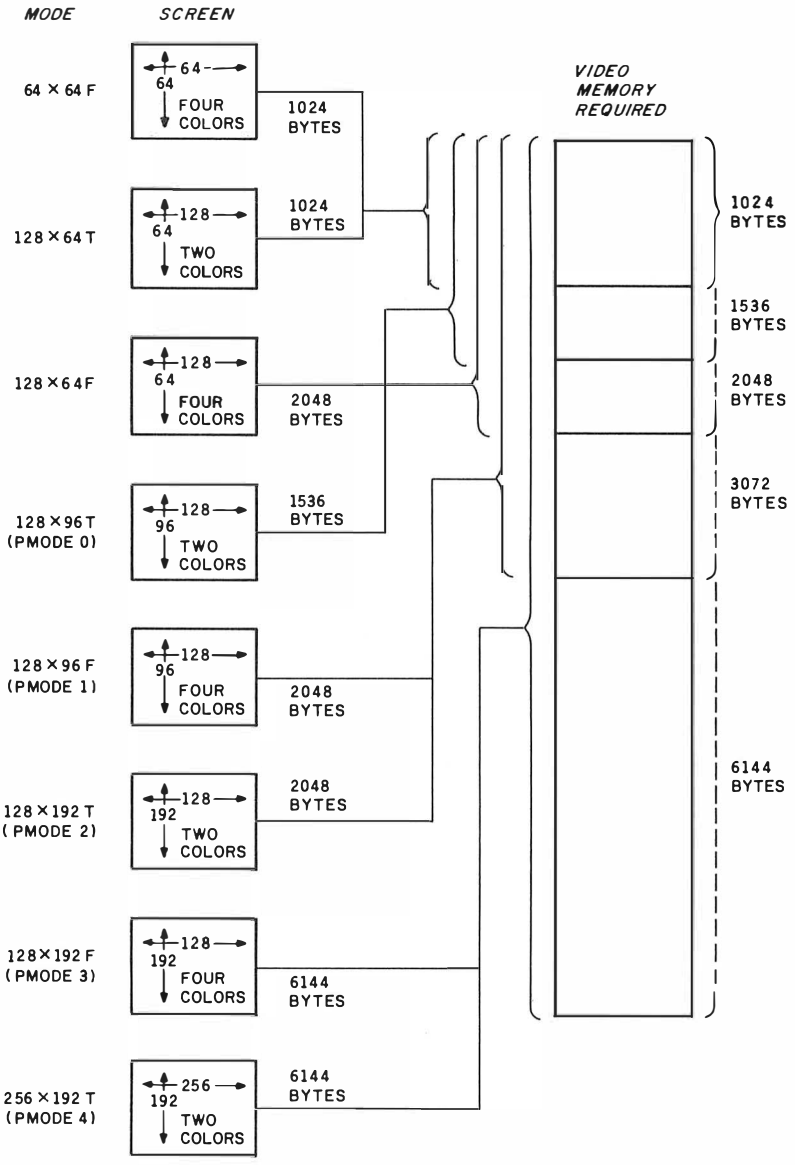


Figure 2.1: Graphics modes.

BORDER AND COLOR SET FOR GRAPHICS MODES

The border in the alphanumeric and semigraphics modes was always black, no matter what colors were used on the image area of the screen. In the graphics modes we have a choice of either a green or a buff color for the border. By border, of course, we mean the space surrounding the area in which we can actually store graphics characters or text (see Figure 2.2). The choice of border is hardwired by the video display generator (VDG) chip and can't be changed.

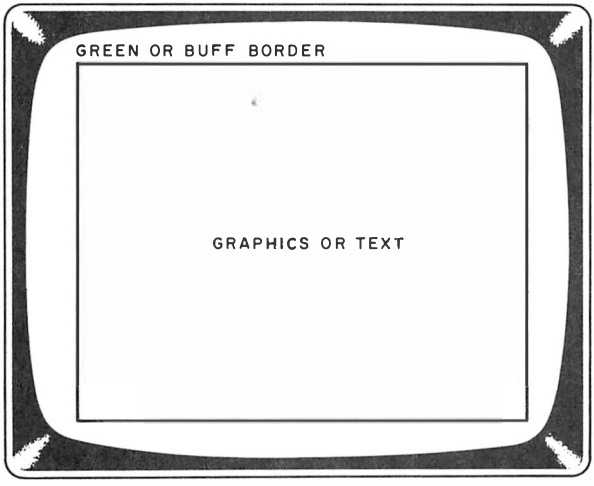


Figure 2.2: Border area.

The *color set* in the graphics modes is also predefined, as shown in Table 2.1. The color sets in the four-color mode are green, yellow, blue, and red with a green border, or buff, cyan, magenta, and orange, with a buff border. Note that the border color for each set is always the first color of the set. The color sets in the two-color mode are black and green with a green border or black and buff with a buff-colored border. Here the border is the last color of the set.

The color set, including border, is selected in Extended BASIC by the SCREEN command, which has the format

SCREEN *type*,*color set*

In the command above, type is 0 for text screen or 1 for graphics screen, and color set is 0 or 1 for the two-color sets.

	Four-Color			Two-Color		
	Color	HW	BASIC	Color	HW	BASIC
Color Set 0	*Green	0	1	Black	0	0
	Yellow	1	2	*Green	1	1
	Blue	2	3			
	Red	3	4			
Color Set 1	*Buff	0	5	Black	0	0
	Cyan	1	6	*Buff	1	1
	Magenta	2	7			
	Orange	3	8			

HW = hardware code BASIC = BASIC code * = border color

Table 2.1: Graphics Modes Color Sets.

The SCREEN command selects the color set by outputting a color set select (CSS) signal to the VDG. This signal is either a 0 or 1, selecting one of the two color sets available. There are never more than 2 color sets available in any VDG mode, be it alphanumeric, semigraphics, or graphics.

The type parameter switches from a text screen to graphics screen. In fact, this simply involves setting the alphanumeric/semigraphic mode or the graphics mode. The SCREEN command can be used to switch color sets in the text screen mode, resulting in red on black or black on green. In the graphics mode it switches colors in a similar fashion.

VIDEO MEMORY MAPPING IN THE GRAPHICS MODES

In the semigraphics modes of the last chapter, the video memory mapping was somewhat less than straightforward. (This chapter was simplified from an earlier version which put a Color Computer user in a trauma ward in Des Moines.) The memory mapping in the graphics mode is much easier to understand, and while such an understanding is not essential to successful use of this mode, it will enable you to solve problems more easily and create more sophisticated applications.

In all of the graphics modes the data for display is in video memory as a string of 1- or 2-bit fields. The upper left element of the display is represented by the first field, the next element to the right by the second field, and so forth down to the field for the lower right element. Each

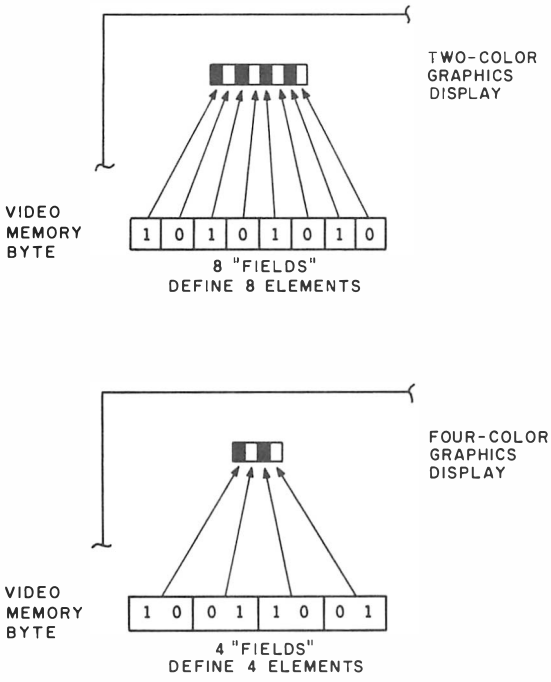


Figure 2.3: Fields in the graphics modes.

field is 1 bit for a two-color mode or 2 bits for a four-color mode, as shown in Figure 2.3.

The total amount of video memory in bytes required is $1 \times H \times V/8$ or $2 \times V \times H/8$, where H is the number of horizontal elements and V is the number of vertical elements. The product is divided by 8 to find the number of 8-bit bytes required. As an example of this, suppose that we have a graphics mode of 128 by 192. In this mode we have only two colors. The total number of bits is $1 \times H \times V = 1 \times 128 \times 192 = 24{,}576$, and the total number of bytes required is $24{,}576/8 = 3{,}072$.

Each memory byte represents 4 or 8 elements, depending upon whether a four- or two-color mode is in force. For example, the upper row of a 128 horizontal element mode with two colors is represented by 128/8 or 16 bytes. These bytes would occupy consecutive memory locations starting at video memory location 1536 and ending in video memory location 3071.

As in the other video modes, video memory is a part of RAM and continues upward, as shown in Figure 2.4.

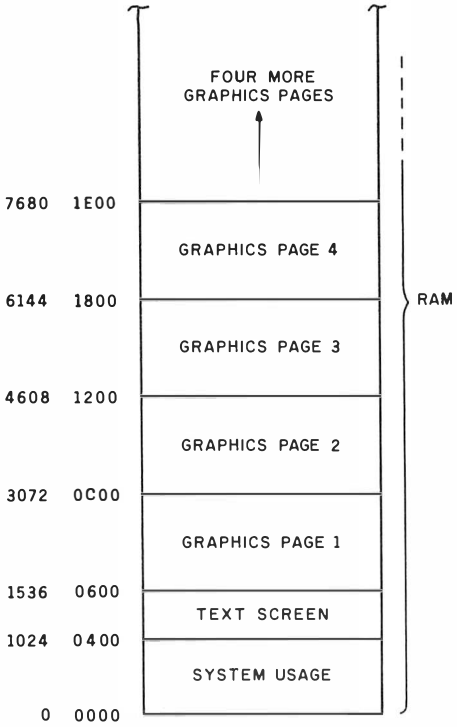


Figure 2.4: Video memory in graphics.

PMODE COMMAND

The PMODE command in extended BASIC selects one of 5 higher resolution graphics modes. The format of the PMODE command is

PMODE *mode ,start-page*

The mode value in PMODE may be 0 through 4. The mode values correspond to the resolutions and colors shown in Table 2.2. The start-page selects a video memory page from 1 to 8. We'll discuss pages a little later in this chapter, but for now assume that we're talking about the first page, starting at video memory location 1536.

As you can see from the table, there are three modes not supported by the PMODE command. These three modes are the 64 × 64 F mode, the 128 × 64 F mode, and the 128 × 64 T mode. The first number is

PMODE	Resolution (H×V)	Bytes	Pages	Colors
0	128×96	1536	1	Blk/Grn or Blk/Buff
1	128×96	3072	2	GYBR or BCMO
2	128×192	3072	2	Blk/Grn or Blk/Buff
3	128×192	6144	3	GYBR or BCMO
4	256×192	6144	4	Blk/Grn or Blk/Buff

GYBR is green, yellow, blue, red
BCMO is buff, cyan, magenta, orange

Table 2.2: PMODE Values vs. Resolution.

the horizontal resolution, the second is the vertical resolution, the "F" is four-color, and the "T" is two-color. This is how we'll refer to these modes in the following discussion. We'll use the PMODE designators for the other five modes.

The 64 × 64 F Mode

This mode is not supported by BASIC. It is a four-color mode that displays graphics in a 64 by 64 matrix. As this is a total of 8192 bits, it requires 1024 bytes of video memory. The mapping for this mode is shown in Figure 2.5.

This mode can be set by POKEs, providing that the video memory is properly set up. The following BASIC program shows how to utilize this mode. The program draws alternating blue and red vertical stripes and then switches to the alternate color set for magenta and orange stripes.

```
100 REM G4XG4F GRAPHICS
110 POKE 65473,1
120 POKE 65474,0
130 POKE 65476,0
140 A=(PEEK(65314) AND 7)+128
150 POKE 65314,A
160 FOR VM=1024 TO 1024+16*31+15
170 POKE VM,187
180 NEXT VM
190 FORI=1 TO 600 : NEXT I
200 A=(PEEK(65314) AND 7)+128+8
210 POKE 65314,A
220 GOTO 220
```

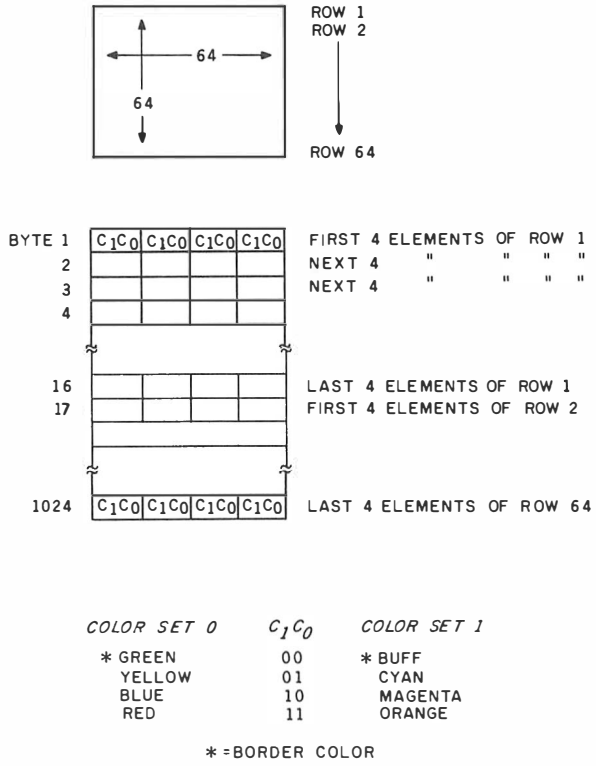


Figure 2.5: 64 × 64 F mode mapping.

The first 3 POKEs set the VDG to the 64 by 64 mode. The next PEEK statement sets the graphics mode. The four statements must be used exactly as they appear. The FOR...TO loop stores a value of 187 into the start of video memory through the halfway point. There are 16 bytes per row, and the last location to be stored is
1024 + 16 × 31 + 15.

After a short timing loop, a second POKE to 65314 is done to change the color set select.

The data value of 187 stored in each byte is 10111011 in binary. There are 2 bits for each element, so this represents alternating 10s and 11s, or blues and reds (color set 0).

The figure drawn is shown in Figure 2.6. There will be "garbage" in the bottom of the screen. A full screen in this mode is 1024 bytes. As only one half of the screen was written, extraneous characters are present in the bottom half of the screen.

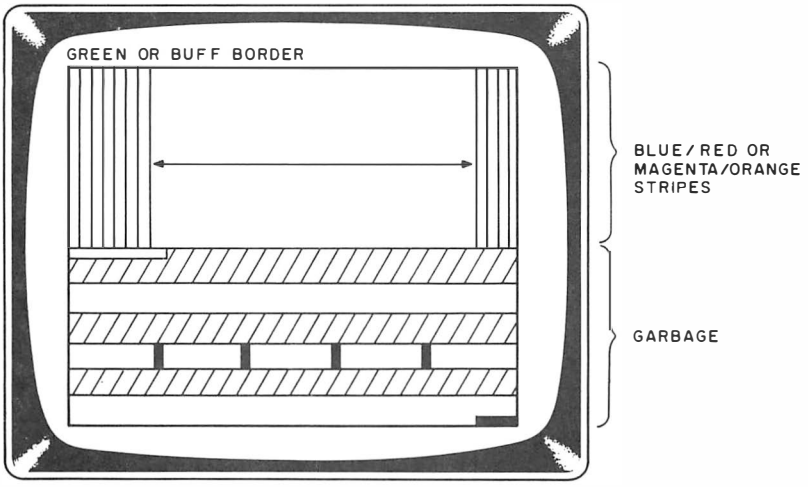


Figure 2.6: 64 × 64 F mode example.

The 128 × 64 T Mode

This is the second mode not supported by BASIC. It is a two-color mode that displays graphics in a 128 by 64 matrix. It requires the same amount of memory as the 64 × 64 F mode — 1024 bytes. The horizontal resolution has been doubled at the expense of halving the number of colors available. The mapping for this mode is shown in Figure 2.7.

This mode can again be set by POKEs in a manner very similar to the 64 × 64 F mode. The following program is almost identical to the 64 × 64 F program except that it initializes the VDG to the 128 × 64 T mode and stores a value of 10101010 (170 decimal). These represent alternating black and green stripes in the first color set and alternating black and buff in the second color set. Note that one of the colors used in each case is the border color, so the edges merge into the border.

```
100 REM 128X64T GRAPHICS
110 POKE 65473,1
120 POKE 65474,0
130 POKE 65476,0
140 A=(PEEK(65314) AND 7)+128+16
150 POKE 65314,A
160 FOR VM=1024 TO 1024+16*31+15
170 POKE VM,170
```

```
180 NEXT VM
190 FOR I=1 TO 600 : NEXT I
200 A=PEEK(65314)+8
210 POKE 65314,A
220 GOTO 220
```

In this mode, there is also garbage in the bottom half of the screen, as only 512 bytes were written.

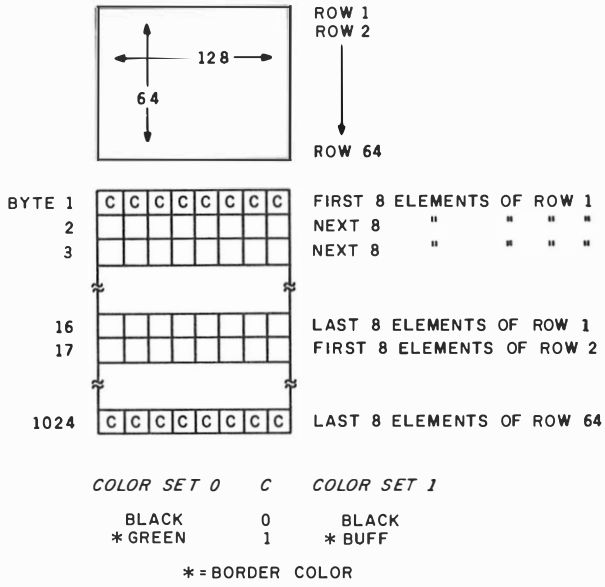


Figure 2.7: 128 × 64 T mode mapping.

The 128 × 64 F Mode

This mode is the third mode not supported by BASIC. It is a four-color mode that displays graphics in a 128 by 64 matrix. As the number of colors has been doubled over the 128 × 64 T, twice as much memory is required — 2048 bytes. The mapping for this mode is shown in Figure 2.8.

This mode, again set by POKEs, is very similar to the previous two modes. The following program is almost identical to the previous programs except that it initializes the VDG to the 128 × 64 F mode and

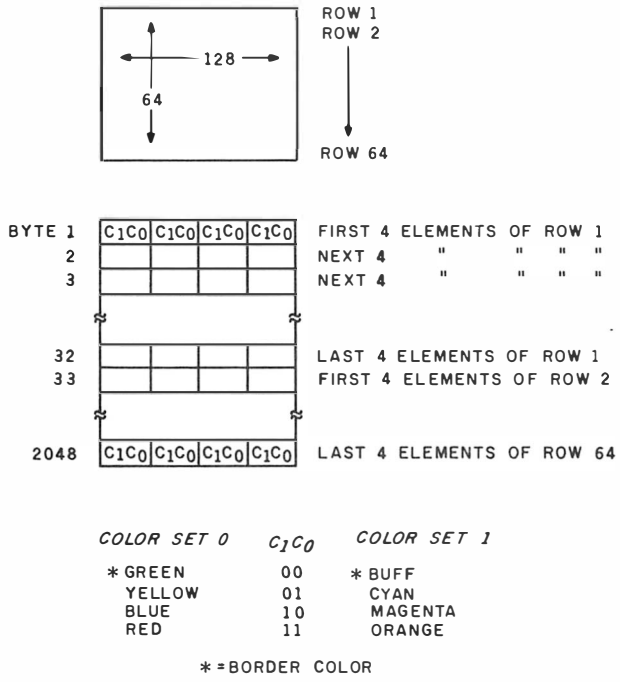


Figure 2.8: 128 × 64 F mode mapping.

stores a value of 10111011 (187 decimal). This value represents alternating blue and red (color set 0) or magenta and orange (color set 1) stripes as in the 64 × 64 F program.

```
100 REM 128X64F GRAPHICS
110 POKE 65472,0
120 POKE 65475,1
130 POKE 65476,1
140 A=(PEEK(65314) AND 7)+128+32
150 POKE 65314,A
160 FOR VM=1024 TO 1024+32*31+31
170 POKE VM,187
180 NEXT VM
190 FORI=1 TO 600 : NEXT I
200 A=PEEK(65314)+8
210 POKE 65314,A
220 GOTO 220
```

Here, 2048 bytes are required, but only 1024 bytes (32 × 31 + 31) are stored, leaving garbage in the bottom half of the screen as before.

Graphics Modes in Extended Color BASIC

The next graphics modes are all supported in Extended Color BASIC, much to the relief of the VDG chip designers. If you plan to use the 64 × 64 F, the 128 × 64 T, or the 128 × 64 F, be sure to follow the POKE sequences above in setting the proper modes. At the end of the PMODE graphics we'll investigate which addresses and bit configurations set all modes so that you may utilize the forgotten modes in your BASIC or assembly-language programming.

Also realize that in the examples above, the area used for graphics started at RAM location 1024, the text screen. In the Extended Color BASIC implementation of the following modes, the text screen is left intact, and the graphics screens (pages) start at location 1536.

PMODE 0 — 128 × 96 T

This mode is a two-color mode that displays graphics in a 128 by 96 matrix. The number of bits required is 128 by 96 by 1 (two colors) or 12,288, which is 1536 bytes. The mapping for this mode is shown in Figure 2.9.

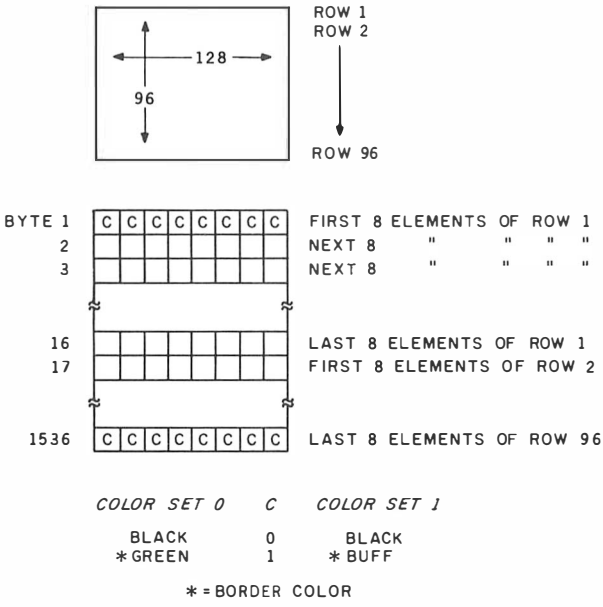


Figure 2.9: PMODE 0 mapping.

"PMODE 0,*start-page*" sets this mode. After this command has been executed in a BASIC program, all graphics will be referenced to a 128 by 96 matrix. Standard colors of black and green (green border) and black and buff (buff border) are used and selected by the color set parameter in the SCREEN command.

PMODE 1 — 128 × 96 F

This mode is a four-color mode that displays graphics in a 128 by 96 matrix. The number of bits required is 128 by 96 by 2 (four colors) or 24,576, which is 3072 bytes. The mapping for this mode is shown in Figure 2.10.

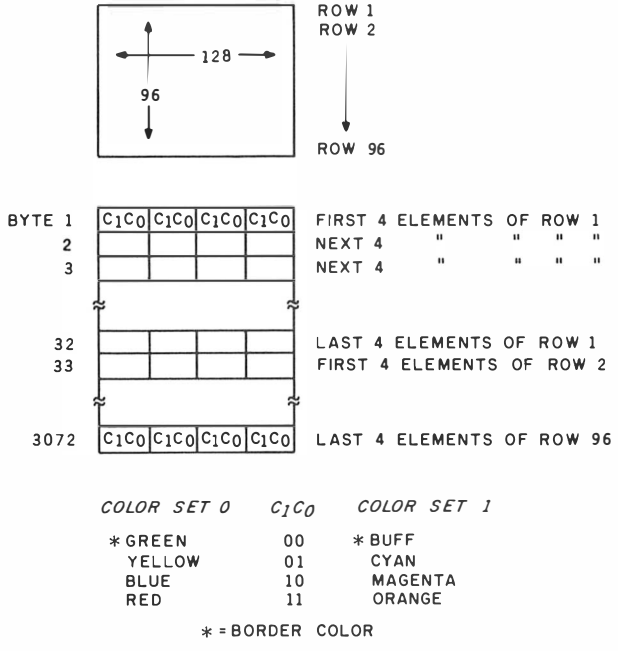


Figure 2.10: PMODE 1 mapping.

"PMODE 1,*start-page*" sets this mode. After this command has been executed in a BASIC program, all graphics will be referenced to a 128 by 96 matrix. Standard colors of green, yellow, blue, and red (green border) and buff, cyan, magenta, and orange (buff border) are used and selected by the color set parameter in the SCREEN command.

PMODE 2 — 128 × 192 T

This is a two-color mode that displays graphics in a 128 by 192 matrix. The number of bits required is 128 by 192 by 1 (two colors) or 24,576, which is 3072 bytes. The mapping for this mode is shown in Figure 2.11.

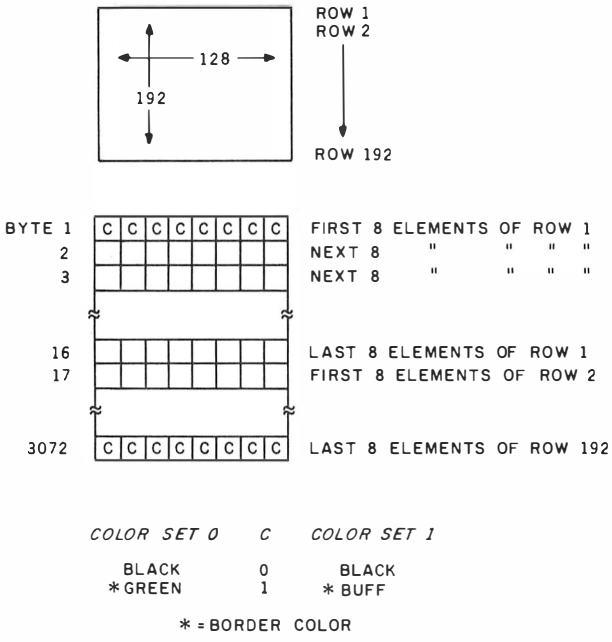


Figure 2.11: PMODE 2 mapping.

"PMODE 2,*start-page*" sets this mode. After this command has been executed in a BASIC program, all graphics will be referenced to a 128 by 192 matrix. Standard colors of black and green (green border) and black and buff (buff border) are used and selected by the color set parameter in the SCREEN command.

PMODE 3 — 128 × 192 F

This mode is a four-color mode that displays graphics in a 128 by 192 matrix. The number of bits required is 128 by 192 by 2 (four colors) or 49,152, which is 6144 bytes. The mapping for this mode is shown in Figure 2.12.

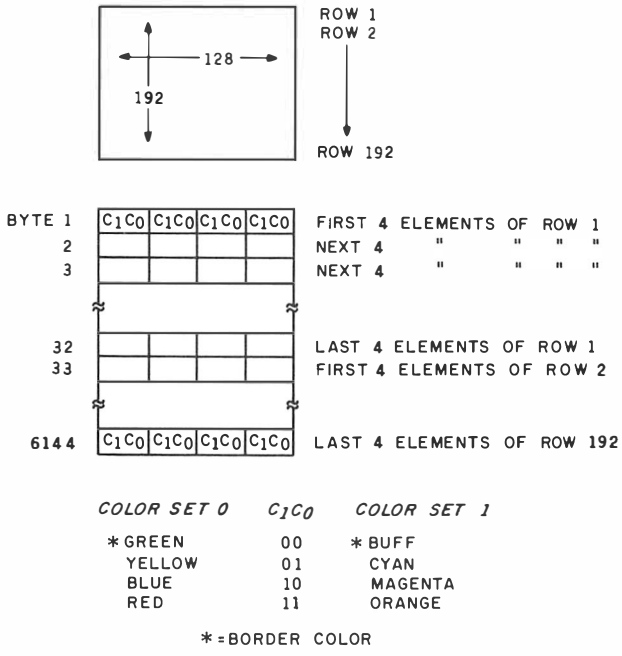


Figure 2.12: PMODE 3 mapping.

"PMODE 3,*start-page*" sets this mode. After this command has been executed in a BASIC program, all graphics will be referenced to a 128 by 192 matrix. Standard colors of green, yellow, blue, and red (green border) and buff, cyan, magenta, and orange (buff border) are used and selected by the color set parameter in the SCREEN command.

PMODE 4 — 256 × 192 T

This is the highest resolution mode, horizontally. (The semigraphics 24 mode has the same vertical resolution of 192 elements, but allows only 64 elements horizontally.)

This mode is a two-color mode that displays graphics in a 256 by 192 matrix. The number of bits required is 256 by 192 by 1 (four colors) or 49152, which is 6,144 bytes. The mapping for this mode is shown in Figure 2.13.

"PMODE 4,*start-page*" sets this mode. After this command has been executed in a BASIC program, all graphics will be referenced to a 256 by 192 matrix. Standard colors of black and green (green border) and black and buff (buff border) are used and selected by the color set parameter in the SCREEN command.

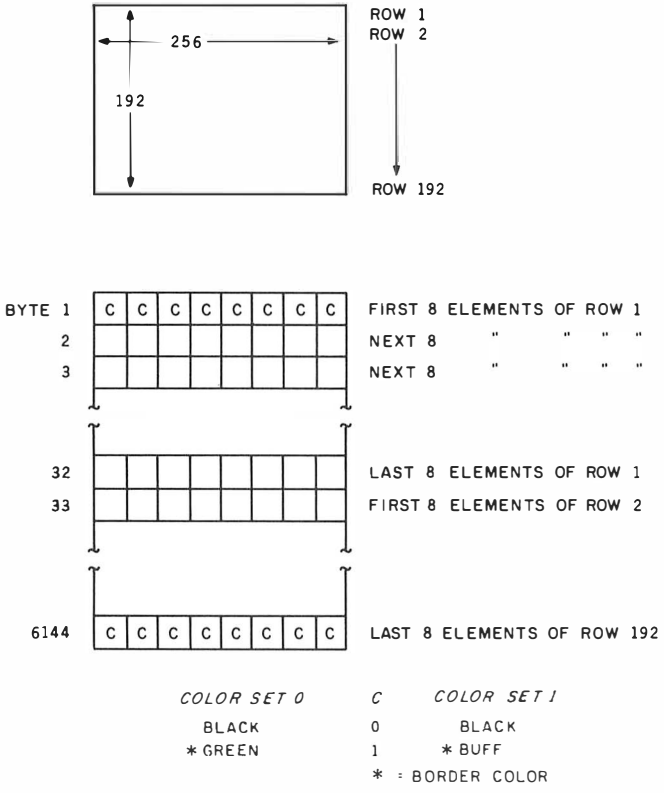


Figure 2.13: PMODE 4 mapping.

USING THE PMODE GRAPHICS MODES

Because the five modes above are supported in Extended Color BASIC by the PMODE command, and because its graphics capabilities are excellent, you will probably want them utilized by the standard Extended Color BASIC graphics commands. All of the modes, however, are selectable by POKEs, providing that the video memory area has been set up with the correct data for display.

The following BASIC program illustrates how POKEs can be used to set up the resolutions for PMODES 0 through 4. A single line is drawn across the center of the screen for each of the modes. The screen starts at location 1024 instead of the normal location 1536 for graphics.

```
100 REM PROGRAM TO SET ALL PMODES
110 PCLEAR 4
120 INPUT PM
130 IF PM<0 OR PM>4 GOTO 120
140 IF (PM=0 OR PM=2) THEN BR=16 ELSE
    BR=32
150 IF PM>1 THEN MR=96 ELSE MR=48
160 DI=MR*BR
170 IF (PM AND 1)=0 THEN VA=0 ELSE VA=255
180 PI=(PM+3)*16+128
190 VD=PM+3 : IF PM=4 THEN VD=VD-1
200 IF (VD AND 4)=4 THEN POKE &HFFC5,1
    ELSE POKE &HFFC4,0
210 IF (VD AND 2)=2 THEN POKE &HFFC3,1
    ELSE POKE &HFFC2,0
220 IF (VD AND 1)=1 THEN POKE &HFFC1,1
    ELSE POKE &HFFC0,0
230 A=(PEEK(&HFF22) AND 7)
240 POKE &HFF22,PI+A
250 FOR VM=1024 TO 1024+MR*BR*2-1
260 IF VA=0 THEN POKE VM,255 ELSE POKE
    VM,0
270 NEXT VM
280 FOR VM=1024+DI TO 1024+DI+BR=1
290 POKE VM,VA
300 NEXT VM
310 GOTO 310
```

The first active statement clears 4 screen pages of 1536 bytes each. The next statement INPUTs a PMODE value of 0 through 4. A check is then made of the PMODE value; if it is other than 0 through 4, the INPUT statement is executed again.

Variable *BR* is the number of bytes per row. The number per row is 16 for a 128 horizontal two-color mode (128 × 1/16) and 32 for a 128 horizontal four-color mode (128 × 2/8) or a 256 horizontal two-color mode (256 × 1/8).

If *PM* is greater than 1, the number of the middle row (variable *MR*) is 96; otherwise it is 48. The displacement in bytes from the start of the video memory is in variable *DI* and is equal to *BR* × *MR*.

Even-numbered PMODEs in *PM* will cause (PM AND 1) to be equal to zero. Even-numbered PMODEs are two-color modes. The value stored in all two-color modes will be 0, which creates black on a border of green. The value of four-color modes will be 255 representing 11111111 for red codes per byte; the display here will be red on a green

border. The value to be stored is held in *VA*.

VD is the value to be sent to the VDG in address &HFFC5–&HFFC0 (65472–65476). *PI* is the value to be sent to address &HFF22 (65314). The coding for these values is shown in Figure 2.14.

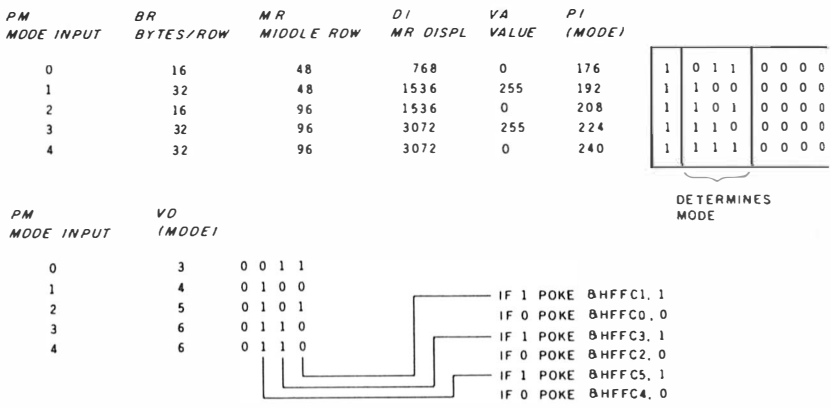


Figure 2.14: POKEing graphics modes example.

The VDG value is output in statements 200 through 220. The *PI* value is merged into address &HFF22 in statements 230 and 240.

The first FOR VM= loop clears the video memory to be used to the border color, green. The end of the video memory is given by 1024+MR × BR× 2–1. The second FOR VM= loop outputs the value to the middle row; the length of this row is *BR*.

VIDEO MEMORY PAGES

We saw from the last chapter and this chapter that the minimum amount of video memory used is 512 bytes. This minimum allocation occurs in the alphanumeric and some semigraphics modes. The maximum video memory used in one display is 6144 bytes. This occurs in the high-resolution semigraphics and graphics modes.

Color BASIC allocates 512 bytes for video memory. This area is the normal text screen. Extended Color BASIC automatically allocates 6144 bytes above the text screen starting at location 1536. Whenever you are working in high-resolution graphics you must be certain that you have

allocated enough memory. This is done in Extended Color BASIC by the PCLEAR command, which has the format

```
PCLEAR n
```

where *n* is the number of graphics pages to be reserved.

A graphics page is equal to 1536 bytes, the number of bytes required for the PMODE 0, the lowest resolution graphics mode. Up to 8 graphics pages can be reserved, a total of 12,288 bytes. Note that a graphics page is not related to a normal memory page for the 6809 microprocessor, which is 256 bytes. The graphics page is somewhat arbitrarily set at the PMODE 0 video memory requirement.

More than one display's worth of video memory is allocated because the start of the display area may be changed from the normal location of 1536 to any of the other 8 video pages. The normal method for doing this is Extended BASIC's PMODE command, in which the start-page parameter defines the starting point for the video display.

```
PMODE mode,start-page
```

Table 2.3 shows the starting addresses for the different values of *start-page* in Extended Color BASIC. The ending address depends upon the current PMODE setting, which may be as much as 6144 bytes beyond the start.

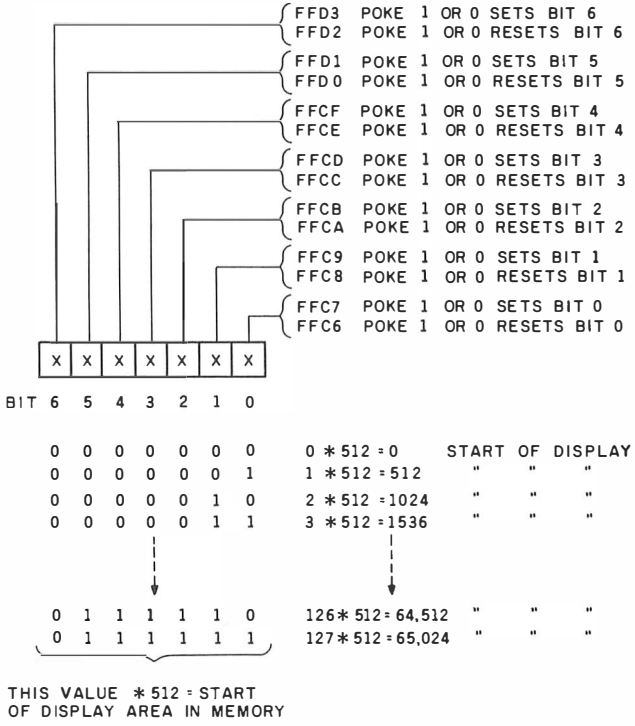
In fact, it's possible to change the start of video memory to any location of the 65,536 locations in RAM/ROM memory. You would

Start-Page	Starting Location (Dec.)	(Hex.)
1	1536	0600
2	3072	0C00
3	4608	1200
4	6144	1800
5	7680	1E00
6	9216	2400
7	10,752	2A00
8	12,288	3000

Table 2.3: Start-Page Values.

normally not want to do this while working within the confines of Extended Color BASIC, but you may want to do it in assembly-language programs or advanced BASIC applications.

The method for doing this is to address the VDG by a series of POKEs. Seven POKES are done to set 7 bits in the VDG. Seven bits represents 0 through 128 (0000000 through 1111111). Each increment represents an area of 512 bytes. For example, specifying 0100000 for the 7 bits would set the start of video memory to 32 × 512 or 16,384. This scheme is shown in Figure 2.15.




```
FFD3 = 65,491          FFCC = 65,484
  D2 = 65,490            CB = 65,483
  D1 = 65,489            CA = 65,482
  D0 = 65,488            C9 = 65,481
  CF = 65,487            C8 = 65,480
  CE = 65,486            C7 = 65,479
  CD = 65,485            C6 = 65,478
```

Figure 2.15: Modifying the video memory start.

To set the 7-bit value, address the VDG locations shown in Figure 2.15. Seven total locations must be addressed by a POKE XXXX,1 or POKE XXXX,0. We've used a POKE XXXX,1 to *set* a bit, and a POKE XXXX,0 to *reset* a bit so that it would be more apparent whether a set or reset was being performed. To set the start of video memory at location 0, the start of RAM, execute

```
100 POKE &HFFD2,0 : POKE &HFFD0,0
110 POKE &HFFCE,0 : POKE &HFFCC,0
120 POKE &HFFCA,0 : POKE &HFFC8,0
130 POKE &HFFC6,0
140 GOTO 140
```

This example is shown in Figure 2.16.

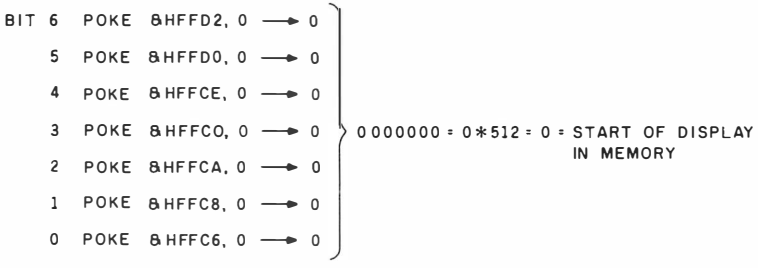


Figure 2.16: Modifying video memory start example.

If you do this, you will see the working storage of the BASIC interpreter. Because the values are a mix of alphanumerics and semigraphics, you will see some characters and some graphics. If you set this value and then run a BASIC program, you will see variables changing dynamically during execution of the program.

THE MYSTIFYING GRAPHICS ORACLE

We've covered a lot of ground in these two chapters. To help present a synopsis of everything we've learned, Appendix IV shows all of the graphics modes, their video and memory formats, color sets, BASIC support, and POKE sequences. In the following chapters we'll be exploring the BASIC and Extended BASIC commands and occasionally referring to the material presented in this chapter. Don't feel obligated to memorize the information already presented, but use it as a guide when you get confused in the following material.


ADDING COLOR- IT'S SUNSATIONAL!
WITTEY

CHAPTER 3

Color BASIC Graphics Capabilities

This chapter is for those users who have not upgraded to Extended Color BASIC. The built-in graphics commands for Extended Color BASIC are excellent — using them you can draw straight lines, circles, ellipses, rectangles, and odd-shaped figures, not to mention painting areas with colors and other graphics-oriented operations. Reproducing the same capabilities for Color BASIC would require an assembly-language program of moderate size and complexity. A BASIC program driver might conceivably add some of the features but would be very slow in comparison to Extended Color BASIC.

Our best advice to a user seriously interested in color graphics is to add Extended Color BASIC. (Let's see, 10 percent of every Extended Color BASIC upgrade kit purchased is...) In the meantime, we'll describe the graphics available in the Color BASIC and show you how to implement some color graphics capabilities.

COLOR BASIC COLOR CAPABILITY

There are 4 Color BASIC commands related to graphics: CLS, SET, RESET, and POINT. CLS clears the screen to one of eight colors. SET and RESET operate on a graphics point. The POINT command returns the value of a graphics point.

The graphics mode used for all commands is the semigraphics 4 mode described in the first chapter. If you will be using these commands and haven't read the first chapter, you may want to read the chapter now

and pay particular attention to this semigraphics mode. We'll provide a refresher course in the following, though.

The semigraphics 4 mode operates on a display of 64 horizontal elements by 32 vertical elements, as shown in Figure 3.1. The basic screen is 256 pixels (picture elements) wide and 192 pixels high. This means that every element in this mode is 4 pixels wide by 6 pixels high, as shown in the figure. As every pixel is roughly square, we're displaying an element that is a rectangle higher than it is wide.

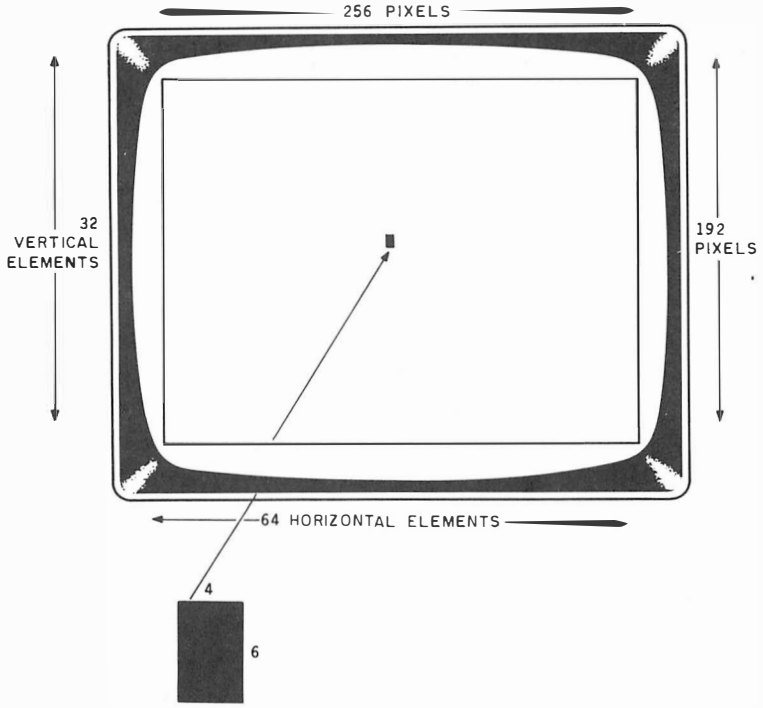


Figure 3.1: Color BASIC graphic screen format.

The semigraphics 4 mode uses 512 bytes of video memory for each display of 64 elements by 32 elements. The mapping for the display is shown in Figure 3.2. The normal location of video memory is in RAM location 1024 and continues through memory location 1535.

Each byte of memory controls four elements on the screen, as shown in Figure 3.2. The four elements are subdivisions of an alphabetic character position. If text is on the screen a single alphabetic character will be displayed in the character position.

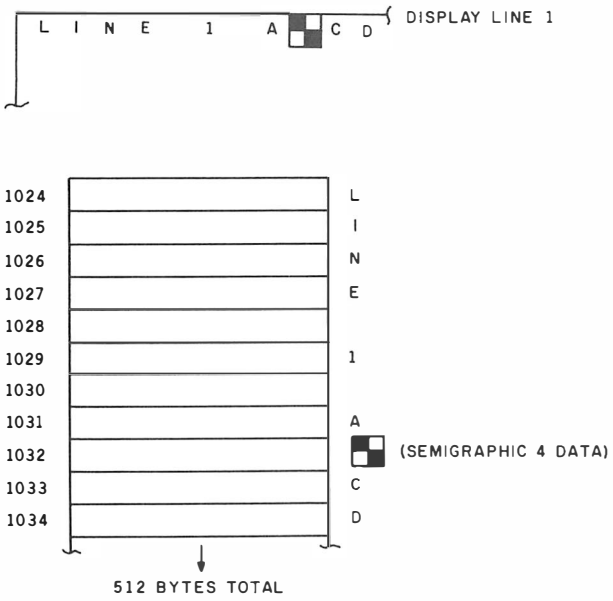


Figure 3.2: Color BASIC graphic mapping.

A byte has 8 bits (binary digits), each of which is a one or a zero. An on condition is represented by a one, while an off condition is represented by a zero.

If the first bit of any byte in video memory is a zero, the byte represents an alphabetic character. The next bit of an alphabetic character defines whether the character is inverted, i.e., black on a green background or red on an orange background. The remaining 6 bits of the character are the code for the character. Codes for Color Computer characters are shown in Tables 1.2 and 1.3.

If the first bit of any byte in video memory is not a zero, but a one, the byte represents a graphics character. In this case, the format for the byte is as shown in Figure 3.3.

The next 3 bits after the first bit define the color. The bits may be 000, 001, 010, 011, 100, 101, 110, or 111, representing colors of green, yellow, blue, red, buff, cyan, magenta, or orange. This will be the color of each of the 4 elements in the character position, providing each is turned on. The background or border color for this mode is always black. If an element is turned off, black will appear.

The next 4 bits define the on/off state of each of the 4 elements. A one is on, while a zero is off. Note that the four elements must have the same color, defined by the 3 color bits.

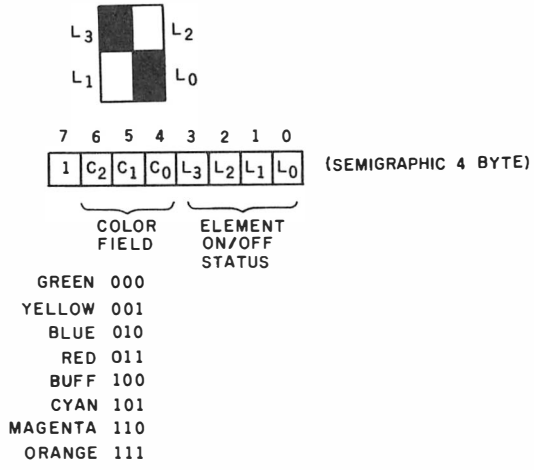


Figure 3.3: Graphic character format.

CLS COMMAND

The format for the CLS (clear screen) command is

```
CLS (c)
```

where *c* is a color code of 0 through 8, representing the colors shown in Table 3.1.

CLS (c)

(c)	Color
0	Black
1	*Green
2	Yellow
3	Blue
4	Red
5	Buff
6	Cyan
7	Magenta
8	Orange

*Default if (c) omitted

Table 3.1: CLS Colors.

The CLS command simply clears the 512 bytes of video memory to the specified color by setting the 3 color bits (it subtracts one from the *c* value), and turning on the 4 on/off bits. The exception for this is a *c* value of black, which simply turns off all on/off bits. To see how this works, run the following program. It will display the value of all video memory bits in the center of the screen, preceded by the video memory location.

```
100 REM DISPLAY CLS FUNCTION ACTION
110 INPUT A
120 CLS(A)
130 FOR I=1024 TO 1024+511
140 PRINT @ 256+10,I;PEEK(I);
150 NEXT I
```

A CLS value of 4 produces the display shown in Figure 3.4, with the indicated breakdown of the value.

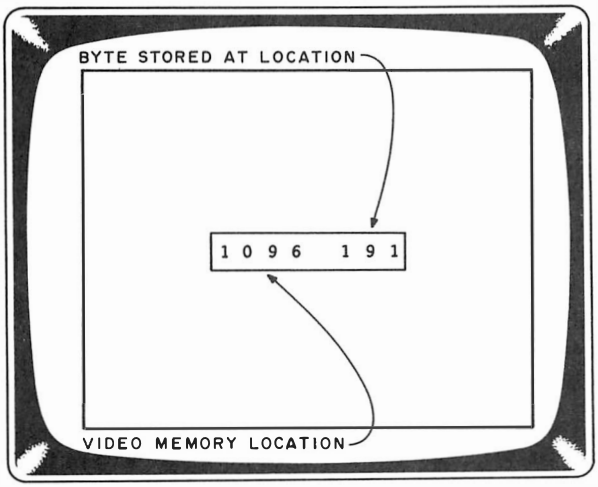


Figure 3.4: CLS example.

The color to which the screen has been cleared remains in force even when a return is made to the Color BASIC interpreter. As more text is entered, however, the screen is reset to the alphanumeric black on green for the character positions. Pressing the (CLEAR) key or executing a CLS without any argument will also reset the screen to a standard green

background. *This background is not the semigraphics mode background but rather the alphanumerics mode; each byte in video memory has a 96 code for a space rather than a 143 code for a green color with all elements on.*

The CLS takes approximately 1/100 second, making it very fast indeed, much faster than any BASIC FOR...NEXT loop could be.

SET, RESET COMMANDS

The formats of the SET and RESET commands are

```
SET (h,v,c)
RESET (h,v)
```

The *h* is a value specifying the horizontal element number from 0 through 63. The *v* is a value specifying the vertical element number from 0 through 31. The *c* is a color code that is the same as the CLS command.

The SET, of course, sets the element specified to the color specified; the RESET resets the element to the background color, or black. This produces some unexpected results. Look at Figure 3.5. The screen is first cleared to blue by a CLS(3). Next, the approximate center dot is set to red by SET(31,16,4). What should the result be? A red dot in the center of a blue screen? Wrong! The CLS turns on all four of the elements. The SET changes the color of one element. In doing so, it puts the red color code in the video memory byte for 31,16. However, this is also the video memory byte for 30,16; 30,17; and 31,17. Since these elements are on by the action of the CLS, changing the color for the SET changes the color for all four elements!

A following RESET turns off the 31,16 element by resetting the bit for that element to 0. A black element results!

Because of this implementation, you must work with a color on a black background if you are using SET/RESET and if you want the maximum resolution of 64 by 32 elements. If you work with one color on another color background, you will be working with a resolution of 32 by 16 elements. An alternative to this is working with a higher resolution semigraphics mode; we'll discuss this option shortly.

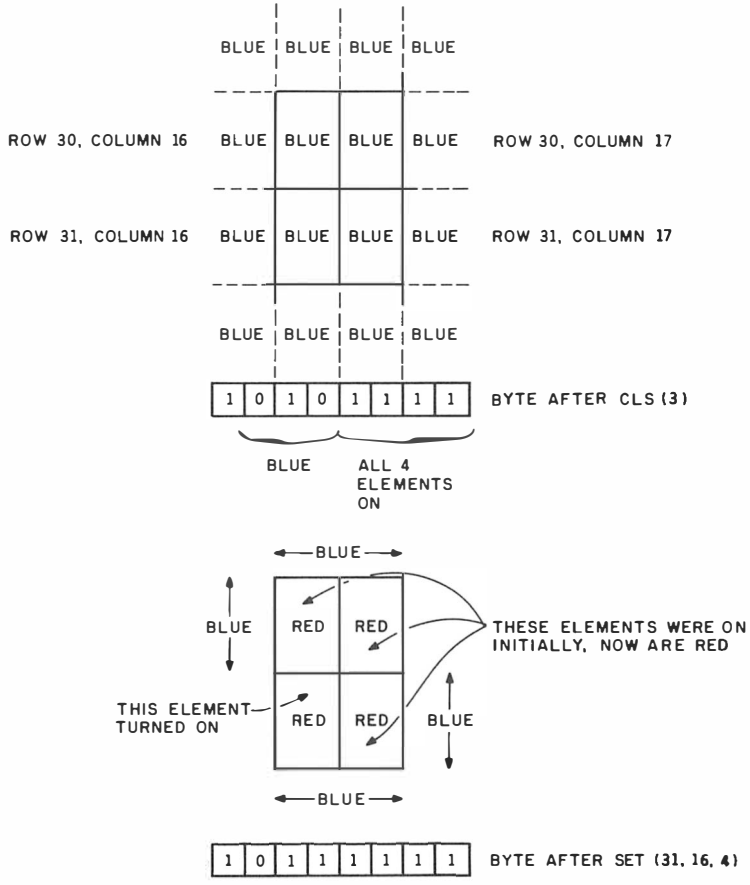


Figure 3.5: SET/RESET example.

Speed of SET/RESET

The SET/RESET commands execute at about 4.2 milliseconds (4.2/1000 second) per point. It takes about 8 seconds to SET 2048 points, not counting the overhead of a loop.

POINT COMMAND

The format of the POINT command is

```
POINT (h,v)
```

The *h* value is the horizontal element number of 0 through 63; the *v* is the vertical element number of 0 through 31. The POINT command tests the element specified. One of three conditions may be present:

- The character position may be in the text, or alphanumeric mode. In this case, POINT returns a -1.
- The element may be set to a color. In this case, POINT returns the color code of 1 through 8.
- The element may be reset (black). In this case, POINT returns a 0.

To see how this works, look at the following example.

```
100 REM SAMPLE USE OF POINT
110 CLS(1)
120 PRINT @ 256, "TEXT";
130 SET (0,18,3)
140 RESET (1,18)
150 A=POINT (0,16)
160 B=POINT (0,18)
170 C=POINT (1,18)
180 PRINT @ 320,A;B;C;
```

This first clears the screen to a green background. The "(1)" in the CLS statement isn't strictly necessary, as this is the default color. Next, a PRINT @ 256 of "TEXT" is done. This sets the 256th, 257th, 258th, and 259th bytes of the video memory (locations 1536 through 1539) to the alphanumeric characters "TEXT". Next, a SET (0,18,3) is done to SET the next element below the "T" to blue. This is followed by a RESET of (1,18). The situation in video memory is now as shown in Figure 3.6.

The POINT command is then used to test (0,32), (0,34), and (1,34). The element at (0,32) is the part of the alphanumeric character "T". The entire byte for this text character is in the text format, and a -1 is returned as variable A. The element at (0,34) is the upper left element of the character position below the "T". It has been SET to 3, or blue, and this value is returned in variable B. The element at (1,34) is the upper right element of the character position below the "T". It has been RESET to black, and a 0 is returned as variable C.

Is the POINT command useful? Not very. (Oh, oh. I expect thousands of letters here, informing me of a myriad of applications for POINT...) It is one of those BASIC commands included in the BASIC lexicon which can occasionally be used but will probably never appear in

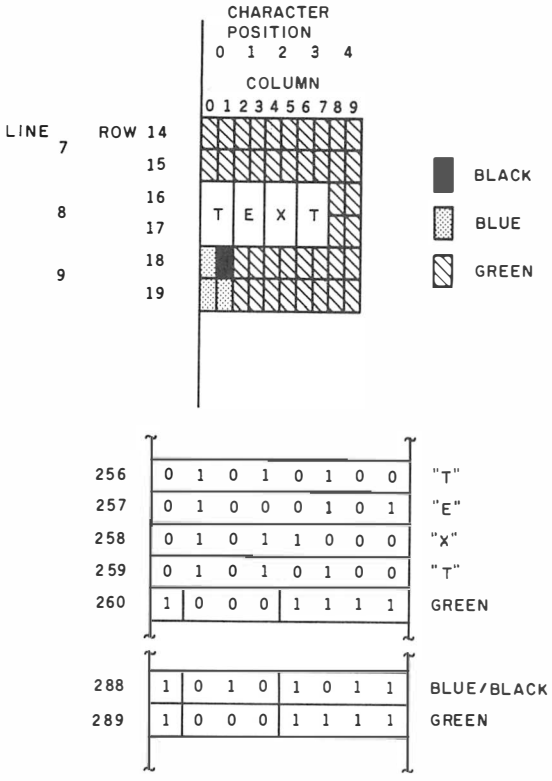


Figure 3.6: POINT example.

most BASIC programs. It could be used to test the on/off status of an element prior to RESETting a point, example. This would avoid having to store a table of values that have been SET and must be RESET before new graphics are displayed.

The POINT command takes the same amount of time to execute as the SET, about 4.2 milliseconds.

TYPICAL USES OF SET/RESET

Because of the low resolution, SET/RESET is useful for such things as bar graphs using several colors, plots that use a black background, or crude figures with possible animation.

The program below plots a bar graph of up to eight values, as shown in Figure 3.7. The background of the graph is green, the axes of the graph are black, and the bars on the graph are alternating colors.

```
100 REM BAR GRAPH PLOT
110 CLS
120 I=0: A(8)=-1
130 PRINT "INPUT 8 VALUES FROM 0 TO 100
    ENDING WITH -1"
140 INPUT A(I)
150 IF A(I)=-1 GOTO 180
160 I=I+1 : IF I=8 GOTO 180
170 GOTO 140
180 CLS(1)
190 J=100
200 FOR I=64 TO 64+32*10 STEP 32
210 PRINT @ I,J;
220 J=J-10
230 NEXT I
240 PRINT @ 422, "1 2 3 4 5 6 7 8"
250 FOR Y=4 TO 25
260 RESET (10,Y)
270 NEXT Y
280 FOR X=11 TO 45
290 RESET (X,25)
300 NEXT X
310 C=2
320 FOR I=0 TO 8
330 B=A(I)
340 IF B=-1 THEN GOTO 430
350 IF B<0 THEN B=0
360 IF B>100 THEN B=100
370 B=B/5 : IF B=0 GOTO 420
380 FOR Y=24 TO 24-B STEP -2
390 SET (I*4+12,Y,C)
400 NEXT Y
410 C=C+1 : IF C=9 THEN C=2
420 NEXT I
430 GOTO 430
```

This program can be divided into several parts, as shown in Figure 3.8. The statements from 100 through 170 input 8 values into a nine-element array, A. The last element of this array always holds a –1 as a terminating value. Entering a –1 at any time terminates the input. At that point the A array may be filled with 0 to 8 values.

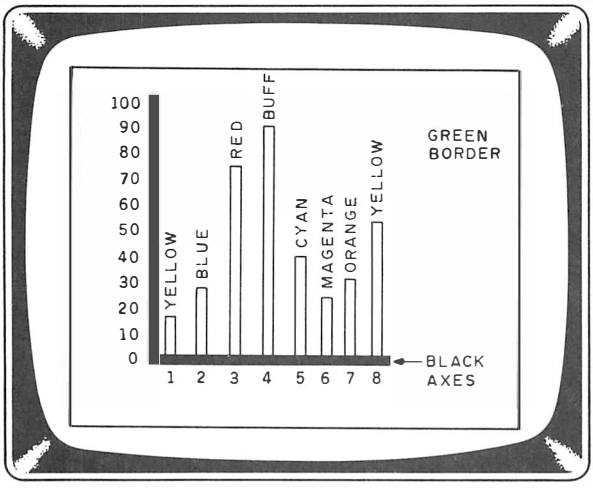


Figure 3.7: Bar graph example.

```
100 REM BAR GRAPH PLOT
110 CLS
120 I=0 : A(8)=-1
130 PRINT "INPUT 8 VALUES FROM
    0 TO 100 ENDING WITH -1"
140 INPUT A(I)
150 IF A(I)=-1 GOTO 180
160 I=I+1 : IF I=8 GOTO 180
170 GOTO 140
180 CLS(1)
190 J=100
200 FOR I=64 TO 64+32*10 STEP 32
210 PRINT @ I,J,
220 J=J-10
230 NEXT I
240 PRINT @ 422,"1 2 3 4 5 6 7 8"
250 FOR Y=4 TO 25
260 RESET (10,Y)
270 NEXT Y
280 FOR X=11 TO 45
290 RESET (X,25)
300 NEXT X
310 C=2
320 FOR I=0 TO 8
330 B=A(I)
340 IF B=-1 THEN GOTO 430
350 IF B<0 THEN B=0
360 IF B>100 THEN B=100
370 B=B/5 : IF B=0 THEN GOTO 420
380 FOR Y=24 TO 24-B STEP -2
390 SET (I*4+12,Y,C)
400 NEXT Y
410 C=C+1 : IF C=9 THEN C=2
420 NEXT I
430 GOTO 430
```

Input 0-8 values to A( )

Display "100,90... 10,0"

Print X coordinate values

Plot Y axis

Plot X axis

Display bars

Figure 3.8: Bar graph program structure.

The statements from 180 through 230 display the *Y* coordinate values of "100, 90, ..., 10, 0". The *X* coordinate values are simply PRINTed with the statement at 240.

The *Y* axis is plotted by statements 250 through 270. The CLS(1) in statement 180 cleared the screen by storing semigraphics characters. RESETting the points will turn off the left-hand elements in each character position, producing a black color. The *X* axis is plotted in statements 280 through 300, also producing a black background.

Statements 310 through 420 display the values as bars on the graph. The color codes for the SET are held in variable C. Colors from 2 through 8 will be displayed. (1 is green, and this will not show up.) For each of eight values, the value is obtained from the array (330), checked for validity and changed to a limit value if necessary (340 through 360), and then scaled.

Since there are 10 character positions or 20 vertical elements involved, each element represents a unit of five: 0, 5, 10, ..., 100. The value is divided by five to give the number of elements involved. This number may be a fractional value, but the SET/RESET command looks only at the integer value and discards the fraction. The bar is drawn by plotting from Y = 24 (bottom) to (25–value/5) in steps of –2. This means that the bar will be drawn from top to bottom.

The SET sets the current *Y* value. In doing so, it actually sets four values as it changes the color code in the byte from green to another color. STEPping in units of −2 avoids duplicating the SET for the next higher element and the problems of blacking out an element discussed earlier. Although the *numeric* resolution is within 5 units, the *graphics* resolution is within 10 units, or one character position, because of the black out problem.

For each value, the *X* coordinate is I × 4 + 12, producing columns at 12, 16, 20, and so forth. These correspond to the label placement under the *X* axis.

Each time a bar is drawn, the color code is changed (410) to a code of 2 through 8. The plotting stops when the next value from the array is –1, or when 8 values have been processed.

INTERMIXING ALPHANUMERICS AND GRAPHICS

The above plot shows how easy it is to mix alphanumerics and graphics characters. Simply be aware of the fact that there are four semigraphics elements in each alphanumeric character position and avoid overwriting of text.

TREATING GRAPHICS AS CHARACTER STRINGS

The semigraphics 4 mode in SET/RESET may be used to advantage in character string display. Each of the four elements for a graphics character position is held in 1 byte in the semigraphics 4 mode. One byte is used to hold one character's worth of data. We can exploit this fact and treat a block of four graphics elements as a *string character.*

The reason for using character strings to generate graphics is that it is fast! To give you an idea how this method works and to show you how speedy it can be, look at the following program:

```
100 REM GRAPHICS CHARACTER STRING EXAMPLE
110 A$=CHR$(128)+CHR$(128)+CHR$(128)+
    CHR$(128)
120 B$=A$+A$+A$+A$+A$+A$+A$+A$
130 FOR I=0 TO 15
140 PRINT B$;
150 NEXT I
```

This program uses the CHR$ function. CHR$ creates a one-character string with the value of the argument. It is used for those arguments that are not valid ASCII character codes, but still legitimate codes in the computer. Many of these codes cannot be generated from the keyboard (there are no keys for them), and CHR$ is a way of handling them in strings.

An A$ = CHR$(128) generates a one-character string that is treated like any other string. In place of an ASCII character, such as decimal 65 for "A", however, is the code 128.

Now in this case the code 128 represents a binary 10000000. We know from study of semigraphics that any character in video memory with the first bit set is a semigraphics character. So it is with CHR$(128). It is a semigraphics character with a color code of 000 (green) with the 4 bits for the elements off (black).

Statement 110 uses the *concatenation* (from Middle Latin "concatena," a stringed instrument — no, just kidding) feature of strings to form a long string of four of these codes. Statement 120 concatenates eight of these, making the total string of B$ equal to 32 codes of 128.

The B$ string is now PRINTed in the next loop, just as any other text string would be. The result is that for each line, 32 bytes of 128s are stored in video memory (which is synonymous with being displayed), just as 32 bytes of another text string would be stored and displayed.

The effect is to clear the screen with a black background. (The display scrolls up as the last character of the last line is output.) The entire clear takes only about ½ second. (We know that CLS(0) can do it

much more rapidly, but this is just to give you an idea of the relative speed of strings over SET/RESET.)

The program below utilizes graphics strings in animation. The figure drawn is a cylinder and piston of a two-stroke engine. It fires with a red explosion and the piston moves up and down. Figure 3.9 shows the display.

```
100 REM ANIMATION USING CHR$ STRINGS
110 CLS(1)
120 A$=CHR$(175)
130 B$=CHR$(128)
140 C$=A$+A$+A$+A$+A$+A$
150 D$=A$+B$+B$+B$+B$+A$
160 E$=CHR$(223)
170 F$=E$+E$+E$+E$
180 G$=B$+B$+B$+B$
190 H$=CHR$(191)
200 I$=H$+H$+H$+H$
210 PRINT @ 205,C$;
220 PRINT @ 237,C$;
230 FOR I=269 TO 454 STEP 32
240 PRINT @ I, D$
250 NEXT I
260 PRINT @ 302,F$;
270 PRINT @ 270,I$;
280 FOR I=0 TO 20: NEXT I
290 FOR I=302 TO 398 STEP 32
300 PRINT @ I-32,G$;
310 PRINT @ I+32,F$;
320 FOR J=0 TO 75: NEXT J
330 NEXT I
340 FOR I=398 TO 334 STEP -32
350 PRINT @ I-32,F$;
360 PRINT @ I+32,G$;
370 FOR J=0 TO 75: NEXT J
380 NEXT I
390 GOTO 270
```

To see how these strings are set up, look at Figure 3.10. The cylinder shell is blue. The inside of the cylinder is black. The piston is cyan. The explosion is red.

A blue color is represented by a semigraphics byte of 175, a black color by a byte of 128, a cyan color by 223, and a red color by 191.

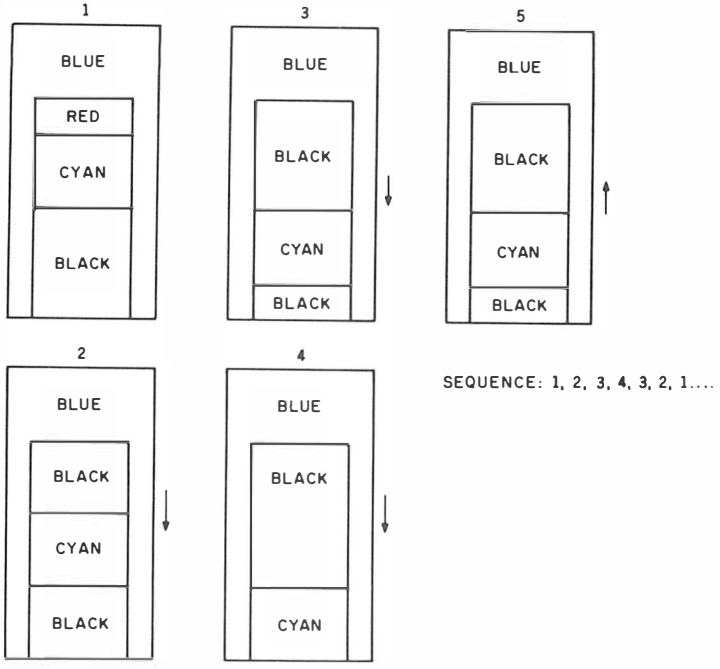


Figure 3.9: Animation example.

These are one-character strings A$, B$, E$, and H$, respectively.

The top 2 rows of the cylinder are 6 bytes of 175, held in C$. The cylinder shell is a blue byte, 4 black bytes, and a blue byte, held in string D$.

Each piston row is 4 bytes of 223, held by string E$. A row of black is held in string G$. The explosion will occur over 1 row, and is held in string I$, which is 4 bytes of red.

The basic outline is set up by statements 210 through 250. Statements 210 and 220 display the upper 2 rows, while statements 230 through 250 display the 6 rows for the cylinder shell.

Statement 260 displays the first row for the piston and statement 270 fires the cylinder by PRINTing I$. Statements 270 through 390 are the main loop for the program.

Each time through the loop the piston moves downward in 4 steps for statements 290 through 330. As each step is made, the previous upper row is erased by PRINTing a row of black. When the piston reaches the last position, statements 340 through 380 move it upward in 4 steps, erasing the previous bottom row of the piston for each step. At the top of the travel, statement 270 fires the cylinder again.

Timing loops at 320 and 370 create a delay for the animation effect.

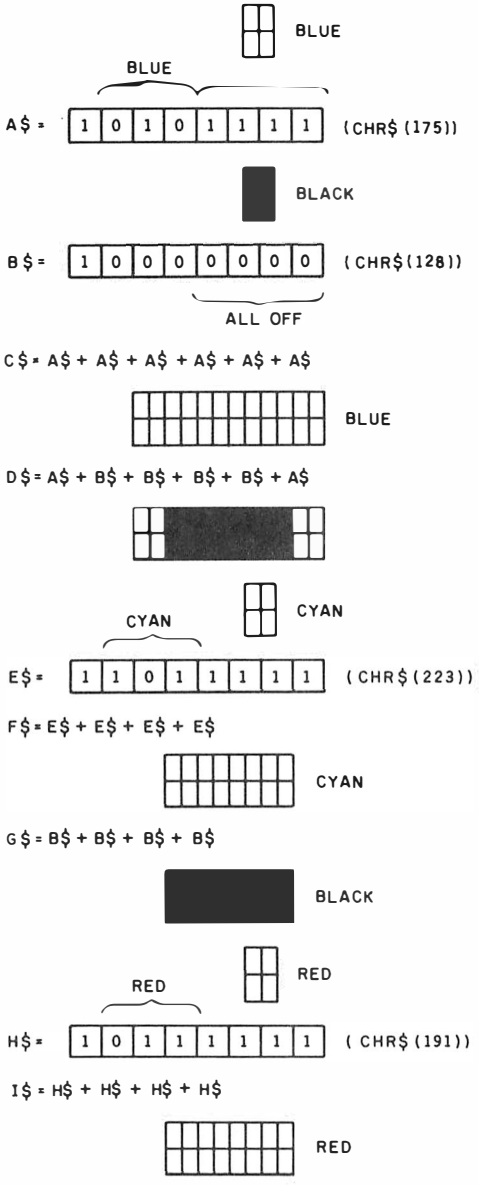


Figure 3.10: Animation example, strings.

OTHER METHODS FOR IMPLEMENTING COLOR BASIC GRAPHICS

Besides the string method, there are two alternatives to SET/RESET. These are the POKE method to change video display memory/semigraphics modes and assembly language.

POKE Method

In the first two chapters, we discussed all of the graphics modes that are available in the Color Computer. There are many more semigraphics modes than the ones used in Color BASIC, and some of these permit 8 colors at resolutions of up to 64 horizontal by 192 vertical. In addition, all of the true graphics modes are available, and these will permit high resolutions of up to 128 by 192. The only catch is that none of these are implemented in Color BASIC.

All of the modes, however, are available by setting the proper mode via POKE statements as shown in Appendix IV. In addition, you must POKE the proper data into video memory, and in some of the modes this is quite involved.

Another problem is that some of the modes will be incompatible with Color BASIC as far as video memory mapping. Color BASIC assumes a 512-byte video memory area, and many of the other modes require a much larger memory area than this. One solution to this is to change the location of the display memory, as described in the previous chapter.

If the display area is changed to a high memory location (dependent upon your RAM size and video memory required for the mode) and that area protected by a CLEAR high memory parameter, then you can alter this memory area with impunity.

It will still be a formidable job to POKE the proper data into video memory, however. Even such a seemingly simple task as drawing a line from point A to point B is not a trivial matter. Here again we must stress that Extended Color BASIC does all of this for you without the agony. (If you *like* agony, perhaps I could persuade you to help me on the next Color Computer book.)

Assembly Language

If you want high-speed graphics with Color BASIC and are clever at programming, then perhaps 6809 assembly language is your best bet. Try your hand with Radio Shack's Editor Assembler for the Color Computer. Assembly language will certainly give you the speed for rapid manipulation of graphics. The complexity is still there, however, in spades, as assembly-language programming is another order of magnitude more difficult than BASIC.

It is possible to combine the features of BASIC *and* machine language by calling high-speed graphics subroutines using the USR call. If you have a limited number of graphics applications, perhaps you might code the graphics routines in machine language (the output of assembly language) and call them by USR calls.

In the next several chapters we'll describe the Extended BASIC graphics commands, and perhaps we'll convince you that the power of Extended BASIC is worth the cost.

NOTES


BORN TO COMMAND!
YAWN!
ZZZ
BURP!
CLICK!
WILLIAMS

CHAPTER 4

Extended Color BASIC: Initialization

In this chapter we'll talk about six of the twelve Extended Color BASIC commands, in addition to discussing the general mapping of the Extended Color BASIC graphics. The six commands are used to either initialize the color graphics or as commands to change colors, screen location, or graphics resolution. The six commands are

- SCREEN Select graphics or text screen
- PCLEAR Reserve graphics pages
- PMODE Set graphics resolution and page
- COLOR Select foreground, background color
- PCLS Clear graphics screen
- PCOPY Copy one graphics page to another

GRAPHICS MODES

We discussed the graphics modes in detail in Chapter 2. We'll give a recap of that material here, for those who didn't want the detailed explanations of Chapter 2. (Always taking the easy way out, eh? You'd make a good writer.)

There are 15 total alphanumerics, semigraphics, and graphics modes built into the Color Computer by the video display generator (VDG) chip. The VDG is a general-purpose semiconductor chip meant to provide many types of color graphics. Some of its modes are more useful than others.

Of the 15 modes, two provide a display of alphanumeric or text data on the screen, five provide a display of semigraphics, and eight provide a display of graphics.

The two alphanumeric modes are supported by the Color BASIC and Extended Color BASIC.

One of the five semigraphics modes is supported in Color BASIC and Extended Color BASIC — the semigraphics 4 mode that provides a 64-by-32 color display of eight colors on a black background. The remaining semigraphics modes are available only by special programming, either in BASIC or assembly language.

The eight graphics modes generally provide high-resolution graphics at the expense of the number of colors available. The modes provide either four or two colors, with resolutions from 64 by 64 to 256 by 192. The higher the resolution and greater the number of colors, the more video memory required. Five of the eight modes are supported by Extended Color BASIC. Those modes are:

- PMODE 0 : 128 horizontal by 96 vertical in two colors and 1536 bytes of video memory
- PMODE 1 : 128 horizontal by 96 vertical in four colors and 3072 bytes of video memory
- PMODE 2 : 128 horizontal by 192 vertical in two colors and 3072 bytes of video memory
- PMODE 3 : 128 horizontal by 192 vertical in four colors and 6144 bytes of video memory
- PMODE 4 : 256 horizontal by 192 vertical in two colors and 6144 bytes of video memory

COLOR SETS

Up to eight colors are available in the VDG — green, yellow, blue, red, buff, cyan, magenta, and orange. In addition, black, the absence of any color on the screen, is used.

In the graphics modes (PMODEs 0 through 4), two or four colors are available, depending upon the mode.

In the two-color mode (PMODE 0, 2, or 4) the two colors are either black and green on a green background, or black and buff on a buff background. The choice of *color set* is selectable by the SCREEN command.

If in the four-color mode (PMODE 1 or 3) the four colors are either green, yellow, blue, and red on a green background, or buff, cyan, magenta, and orange on a buff background. The choice of color set is selectable by the SCREEN command.

The colors available for the 5 PMODEs are shown in Table 4.1.

PMODE	Colors (Color Set 0 or Color Set 1)
0	black/green or black/buff
1	green, yellow, blue, red or buff, cyan, magenta, orange
2	black/green or black/buff
3	green, yellow, blue, red or buff, cyan, magenta, orange
4	black/green or black/buff

Table 4.1: PMODE Colors.

MEMORY MAPPING FOR PMODES

The memory mapping for a Color Computer with Extended Color BASIC is shown in Figure 4.1. RAM occupies the first 16,384 bytes of memory space from location 0 through 16,383 (or 32,767 on 32K systems). RAM is used for storage of BASIC variables (working storage), storage of the text screens, storage of the graphics pages, user BASIC programs, BASIC program variables, string storage, stack, and a possible reserved area for machine-language programs or other uses, as shown in Figure 4.2.

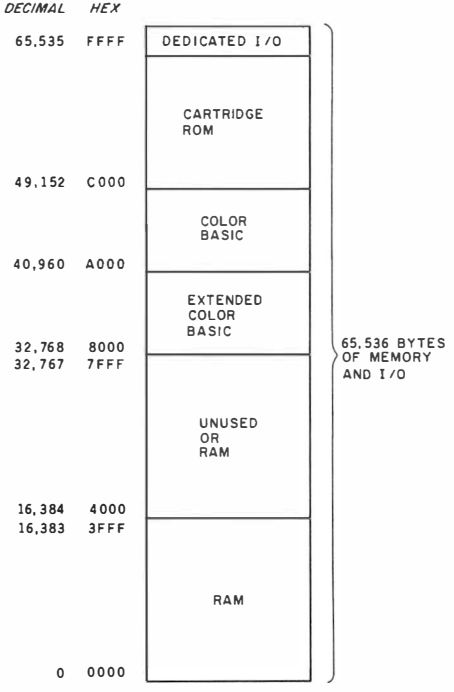


Figure 4.1: Memory mapping for Extended Color BASIC.

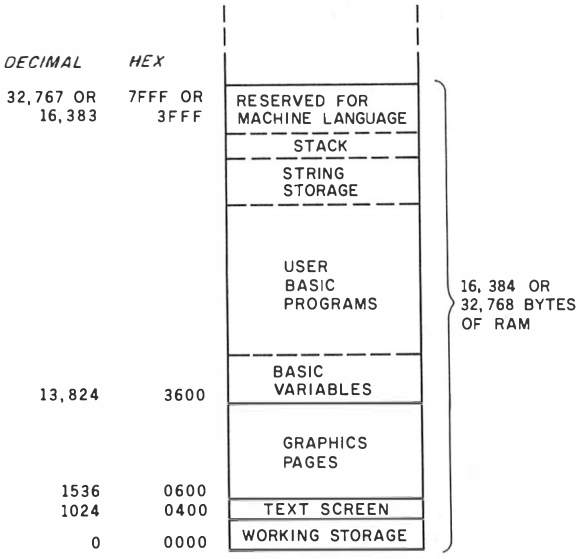


Figure 4.2: RAM storage.

Text Screen

The area from RAM location 1024 through 1535 is dedicated to the text screen (see Figure 4.3). This is the normal display of text that you see when you are entering BASIC statements. It is used any time that the Extended Color BASIC interpreter is reentered.

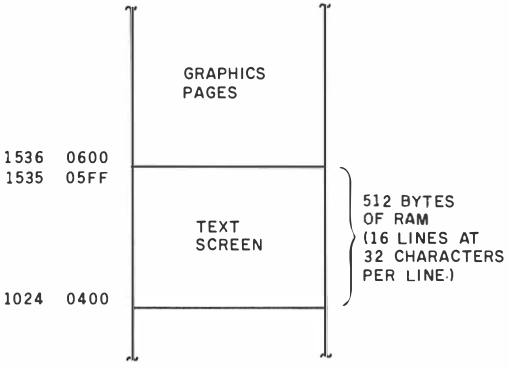


Figure 4.3: Text screen.

Graphics Screen

The area from 1536 upwards is divided into graphics pages (see Figure 4.4). The minimum size graphics area required is 1536 bytes (PMODE 0). The 1536 figure is used as a standard screen size in Extended Color BASIC. PMODEs 0, 1, 2, 3, and 4 use 1536, 3072, 3072, 6144, and 6144 bytes respectively. In terms of graphics pages, then, these modes use 1, 2, 2, 4, and 4 pages respectively.

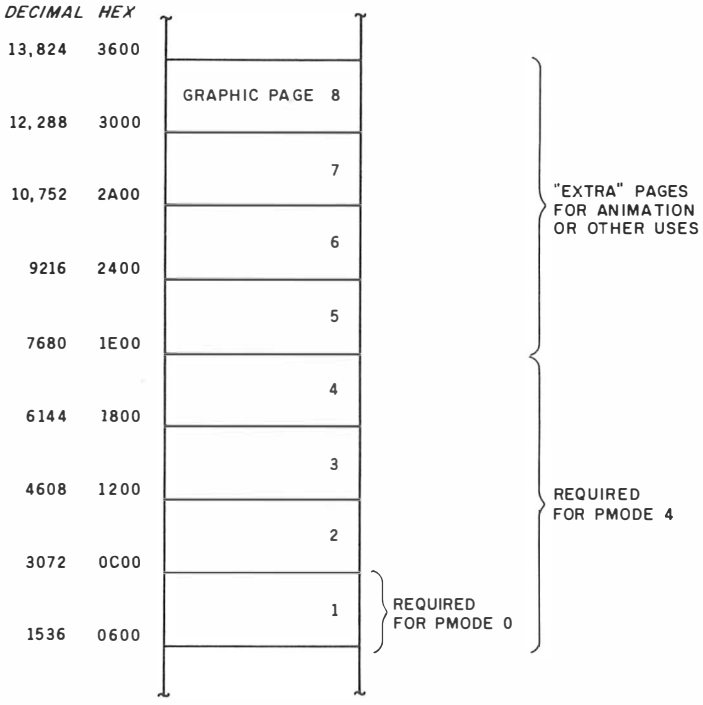


Figure 4.4: Graphics pages.

Up to 8 graphics pages can be allocated by the PCLEAR command. While only 4 pages are required in the highest resolution graphics mode (PMODE 4, 256 by 192, two colors), Extended Color BASIC has the ability to switch rapidly from one page to the other by the PMODE command and to copy one page to another by the PCOPY command. It might be convenient, for example, to rapidly display 8 pages at ⅛ second intervals in the PMODE 0 mode to provide an animation effect.

The Extended BASIC user doesn't have to be aware of the actual memory location of the graphics pages; all references to them are done by a page number from 1 through 8.

THE SCREEN COMMAND

The format of the SCREEN command is

```
SCREEN type , color set
```

where *type* is 0 for text screen or 1 for graphics screen and *color set* is 0 or 1.

The SCREEN command is used to switch from a display of the 512-byte text screen to the display of the current graphics page. The color set selects one of two color sets, as described above. The SCREEN command may be used at any time in an Extended BASIC program. The color set select may be used for the text screen in addition to selecting a set of graphics colors. The standard text display is black on green, but executing

```
SCREEN 0,1
```

will display red on orange for the text. Pressing (BREAK) always resets the text display to black on green.

The short program

```
100 SCREEN 0,1
110 FOR I=0 TO 1000 : NEXT I
120 SCREEN 1,0
130 FOR I=0 TO 1000 : NEXT I
140 SCREEN 1,1
150 FOR I=0 TO 1000 : NEXT I
```

illustrates how the screen may be switched from text to graphics and how the color set may be changed.

When the screen is changed from text to graphics, the graphics displays the *current* page and the *current* graphics resolution selected by the last PMODE statement. If there is garbage in the graphics screen area, the graphics display may be somewhat strange (expect to see lemon rinds, old balogna skins, and so forth). Normally, the graphics screen would contain meaningful data put in by the Extended Color BASIC graphics statements.

THE PCLEAR COMMAND

The PCLEAR command is used to reserve 1 to 8 graphics pages. Extended Color BASIC automatically reserves 4 graphics pages on initialization, enough for every graphics mode. If no PCLEAR is done, this number of pages will remain in force throughout the BASIC program execution.

The format of the PCLEAR is

```
PCLEARn
```

where *n* is 1 through 8.

The PCLEAR is normally issued at the beginning of a BASIC program. If it is issued *dynamically* in the middle of a graphics program after display processing has been done, the PCLEAR may cause unexpected results. When the Extended Color BASIC interpreter executes the PCLEAR, it reserves the appropriate amount of storage after the text screen. The area immediately after the graphics page area is used for the BASIC program. The pointer to this area is adjusted to point to the next location after the last byte of the graphics pages.

The PCLEAR may be used to reduce the default value of 4 graphics pages to 1 to 3, making more RAM available for BASIC program storage, or to make certain that enough video memory is available for programs that do a lot of graphics processing. Use as required.

THE PMODE COMMAND

The PMODE command is used to set one of the graphics modes (0 through 4) and to select the starting page number (1 through 8) for display. The format of PMODE is

```
PMODE mode,start-page
```

The default value for PMODE is PMODE 2,1, a resolution of 128 by 192 (two-color) with a *start-page* number of one.

PMODE may be used at any time to change the resolution of the graphics mode. Don't forget that the actual command that causes the graphics page to be displayed is SCREEN. SCREEN uses the last PMODE mode and start-page values (or the default values) to display the graphics page.

PMODE may be used to switch from one graphics page to the next while keeping the same resolution. For example: If we were in the lowest

resolution of 128 by 96, two-color (PMODE 0), each page would be 1536 bytes. We could rapidly display all 8 graphics pages by

```
100 FOR SP=1 TO 8
110 PMODE 0,SP
120 NEXT SP
```

In this case, the PCLEAR command must have previously allocated 8 graphics pages, and, of course, the pages would have been filled with some meaningful data.

THE PCOPY COMMAND

The PCOPY command is used to copy the contents of one graphic page to another. The format is

```
PCOPY n TO m
```

where *n* and *m* are graphics page numbers of 1 through 8, previously allocated by a PCLEAR command.

The PCOPY command is a convenient way to move display data from one graphics page to another without having to do a series of cumbersome PEEK and POKEs or some other scheme. Dependent upon the mode, don't forget that a screen full of data may take more than a single graphics page and that it may be necessary to do several PCOPYs for a screen's worth of data.

THE PCLS COMMAND

The PCLS command is the Extended Color BASIC equivalent of the CLS command. It clears the current graphics screen with a specified color. The format of PCLS is

```
PCLS color
```

where *color* is a value of 1 through 8, corresponding to the standard graphics colors. If no color is specified, the current background color is used.

The PCLS clears the entire screen; this means that more than 1 graphics page will be used if you are in a PMODE that requires more than 1 graphics page (PMODE 2 through 4).

"Ah there's the rub," as the Bard says in *A Merchant of Fort Worth*. As you know from the discussion earlier in this chapter, only four

or two colors are available at any given time, depending upon the PMODE. How can PCLS clear the current graphics screen with *all* colors? The answer is that it doesn't. Only the available colors are used and only in the current color *set* as specified by the last SCREEN command. Table 4.2 shows the proper codes for various PMODES, color sets, and PCLS colors.

The PCLS command may be used even if a graphics screen is not being displayed (SCREEN 0,X). The PCLS simply clears the current graphics page and the current display status is not a consideration.

PMODE	Color Set	PCLS Color	PCLS Code
0	0	black	0
		*green	1
	1	black	0
		*buff	5
1	0	*green	1
		yellow	2
		blue	3
		red	4
	1	*buff	5
		cyan	6
		magenta	7
		orange	8
2	0	black	0
		*green	1
	1	black	0
		*buff	5
3	0	*green	1
		yellow	2
		blue	3
		red	4
	1	*buff	5
		cyan	6
		magenta	7
		orange	8
4	0	black	0
		*green	1
	1	black	0
		*buff	1

* = border

Table 4.2: PCLS Actions.

THE COLOR COMMAND

The COLOR command is used to set the foreground and background colors. The format of the COLOR command is

COLOR *foreground* ,*background*

where foreground is a value from 1 to 8 and background is a value from 1 to 8.

Let's talk about what we mean by foreground, background, and border, since these are confusing terms.

The *border* for the alphanumeric and semigraphics modes is always black, as shown in Figure 4.5. The border for the graphics modes is either green or buff, depending upon whether we have selected color set 0 (green) or color set 1 (buff). In both the alphanumerics/semigraphics modes and the graphics modes, we can set the *background* color to the same color as the border, making the background disappear beyond the edge of the screen. We can also set the background to another color, creating a distinct border. The background, therefore, is the field upon which graphics figures will be drawn, by the various Extended Color BASIC commands. The *foreground* is the color used to draw the figures.

As with the PCLS command, there may be a limited number of colors available for either background or foreground. If we are in the two-color mode (PMODE 0, 2, or 4), the background can be green or black for color set 0 or buff or black for color set 1. The foreground can be the same two colors. Obviously, we want the foreground to show up, so we are left with green/black or buff/black for the two screen colors whether they are specified as foreground or background.

If we are in a four-color mode (PMODE 1 or 3), the background and foreground colors can be green, yellow, blue, or red, for color set 0 or buff, cyan, magenta, and orange, for color set 1. Here again, we want the colors to show up, so we have the following options for foreground and background colors: green/yellow, green/blue, green/red, yellow/blue, yellow/red, and blue/red, buff/cyan, buff/magenta, buff/orange, cyan/magenta, cyan/orange, and magenta/orange.

The COLOR command may be used at any time; if it is used in the four-color mode to change the foreground color, three separate colors may be displayed upon a one-color background. If that background color is the same as the border color, no border will appear.

As an example of the use of COLOR, look at the following program. It paints three rectangles upon a buff background using the LINE command discussed in the next chapter. The display is shown in Figure 4.5.

```
100 REM USE OF COLOR COMMAND
110 PMODE 1,1
120 SCREEN 1,1
130 PCLS 5
140 COLOR 6,5
150 LINE (48,26)-(84,54),PSET,BF
160 COLOR 7,5
170 LINE (120,82)-(156,114),PSET,BF
180 COLOR 8,5
190 LINE (192,142)-(228,170),PSET,BF
220 GOTO 200
```

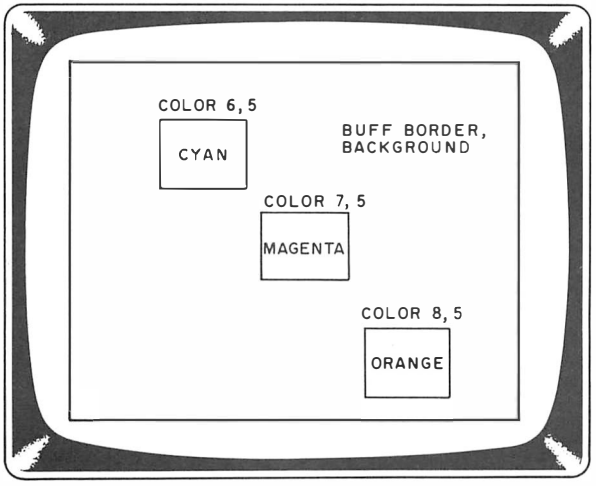
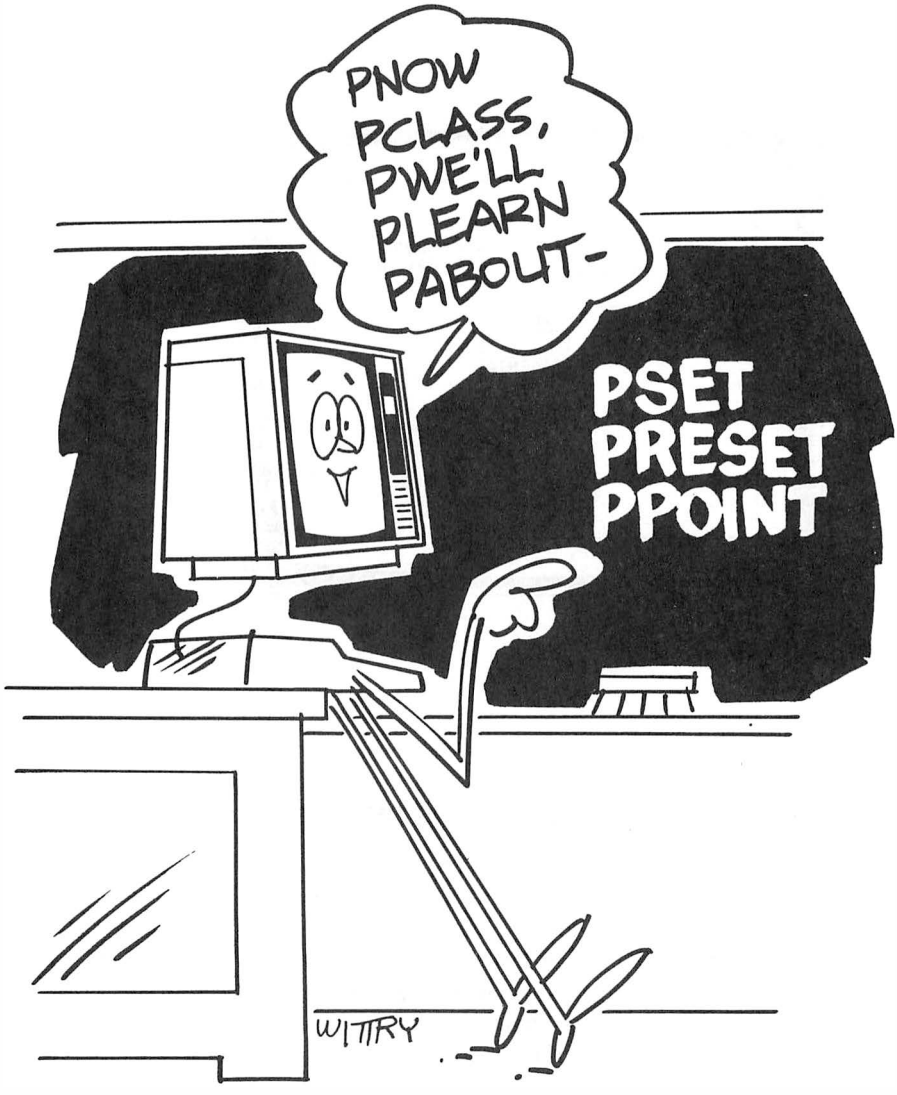


Figure 4.5: COLOR use example.

USING THE COMMANDS IN THIS CHAPTER

The commands discussed in this chapter were really the trivial commands in Extended Color BASIC; they do very little as far as producing graphics on the screen. Understanding SCREEN, PCLEAR, PMODE, COLOR, and PCLS, and PCOPY, however, is essential to working with the remaining graphics commands. In the rest of this book, we'll be discussing the graphics commands that can be used to draw plots, lines, boxes, filled-in boxes, circles, ellipses, arcs, irregular figures, etc., and we'll be using the material in this chapter as a base.


PNOW PCLASS, PWE'LL PLEARN PABOUT-
PSET
PRESET
PPOINT
WITTRY

CHAPTER 5

Extended Color BASIC: PSET, PRESET, PPOINT, and Plotting

We've covered the elementary commands of Extended Color BASIC in previous chapters. In this chapter we'll discuss three of the Extended Color BASIC commands that are used to plot points. These commands are very similar to the Color BASIC commands of SET, RESET, and POINT, which let us set, reset, or test any graphics point; they are PSET, PRESET, and PPOINT.

THE GRAPHICS MATRIX

As we've seen from the previous chapters, the graphics modes use different resolutions, ranging from 128 horizontal elements by 96 vertical elements to 256 by 192. To simplify switching from one resolution to another, all graphics commands in the remainder of the book use the highest resolution graphics mode of 256 horizontal by 192 vertical in specifying coordinates (See Figure 5.1).

We can see how this works by previewing the LINE command, which draws a line between any two screen points. The program below displays the line in the upper left quadrant, as shown in Figure 5.2, for the 5 graphics modes. Notice in running the program that the line becomes finer and finer as the graphics modes go to higher resolution. (It's a fine line between the proper use of PMODE 0 and PMODE 4!)

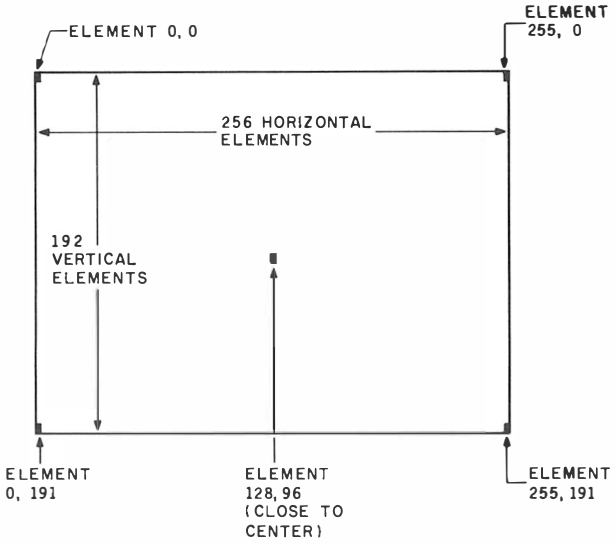


Figure 5.1: Graphics matrix.

```
100 REM DEMONSTRATION OF GRAPHICS
    ADDRESSING
110 FOR P=0 TO 4
120 PMODE P,1
130 SCREEN 1,0
140 PCLS
150 LINE (0,0)-(127,95),PSET
160 FOR I=0 TO 1000 : NEXT I
170 NEXT P
180 GOTO 180
```

USING PSET, PRESET, and PPOINT

The PSET, PRESET, and PPOINT are PMODE SETs, RESETs, and POINTs, hence the prefix "P". They do essentially the same thing as the Color BASIC SET, RESET, and POINT — they set, reset, or test one point on the screen. All coordinates for the commands use the 256-by-192 coordinate system of graphics.

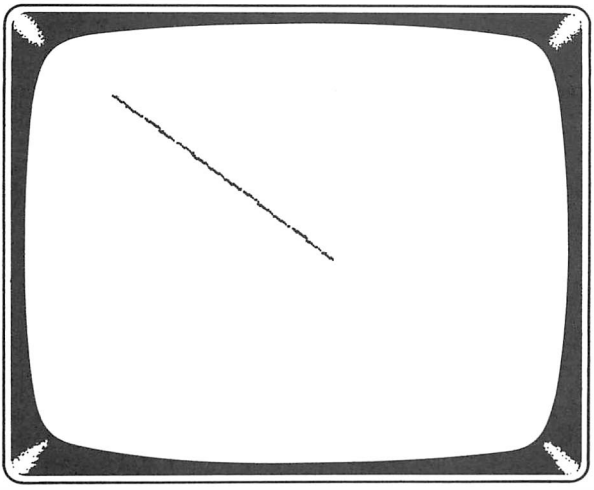


Figure 5.2: Resolution example.

PSET

The format for PSET is

```
PSET (h,v,c)
```

where *h* is a horizontal element number of 0 through 255, *v* is a vertical element number of 0 through 192, and *c* is a color value of 0 through 8. If the *c* parameter is omitted, the current foreground color is used.

PSET will set one element located at *h,v*. The size of the element that is set will be one element of whatever PMODE is in force. The actual physical size will be finer for higher resolution graphics modes. The size of the element in PMODE 4 is 1 unit, for example, but the size of the element in PMODE 0 is 4 units, due to the lower resolution.

This program sets one vertical line in the middle of the screen.

```
100 REM SET ONE VERTICAL LINE
110 PMODE 0,1
120 SCREEN 1,0
130 PCLS
140 FOR V=0 TO 192
150 PSET (127,V)
160 NEXT V
170 GOTO 170
```

The PSET command uses the current *foreground* color to set the points. In the program above, the background color defaulted to black and the foreground color defaulted to green, as PMODE 0 is a two-color mode.

In the program below, the same line is drawn, but PMODE 1 is used with the alternate color set (SCREEN 1,1). The PSET here specifies color 7, and the result turns out to be a blue line on a buff background.

```
100 REM SET ONE VERTICAL LINE
110 PMODE 1,1
120 SCREEN 1,1
130 PCLS
140 FOR V=0 TO 192
150 PSET (127,V,7)
160 NEXT V
170 GOTO 170
```

PRESET

The PRESET command works exactly the same as PSET in reverse. The PRESET resets an element. Since the reset is always assumed to be in the background color, no color value is specified in the format

```
PRESET (H,V)
```

The program below sets a vertical line, delays for a display, and then resets the same line.

```
100 REM SET ONE VERTICAL LINE
110 PMODE 1,1
120 SCREEN 1,1
130 PCLS
140 FOR V=0 TO 192
150 PSET (127,V,7)
160 NEXT V
170 FOR I=0 TO 1000 : NEXT I
180 FOR V=0 TO 192
190 PRESET (127,V)
200 NEXT V
210 GOTO 210
```

PPOINT

PPOINT is used in similar fashion to POINT. It tests the color of a specified graphics point. The format of PPOINT is

```
PPOINT (h,v)
```

PSET and PRESET will be used a lot less frequently than the more powerful Extended Color commands such as LINE and CIRCLE; PPOINT will be used even less frequently. Still, it is nice to have a command that can test the color of an element. As an example of the use of PPOINT, consider the program below, which clears the screen, sets one random element to magenta, and then scans for the element returning the location as shown in Figure 5.3.

```
100 REM EXAMPLE OF PPOINT
110 PMODE 1,1
120 SCREEN 1,1
130 PCLS
140 PSET (RND(255),RND(191),7)
150 FOR X=0 TO 255
160 FOR Y=0 TO 192
170 IF PPOINT(X,Y)=7 GOTO 200
180 NEXT Y
190 NEXT X
200 SCREEN 0,0
210 PRINT "POINT FOUND AT" ;X;Y
```

The program above first sets the PMODE to a four-color mode, PMODE 1. The screen is set to graphics with color set 1 by SCREEN 1,1. Next, the screen is cleared to the background color (buff) by PCLS. One random point is then set by PSET. The color for the PSET is 7, or blue.

A scan of all of the 3,072 points is then made by the *X*, *Y* loops. For each point, a test is made of the color by the PPOINT statement. If any point is color 7, the screen is reset to text and the location of the point is printed.

The maximum (worst-case) time for the scan here is 400 seconds, so be patient!

In thinking about it, PPOINT is about as useful as POINT. See earlier comments.

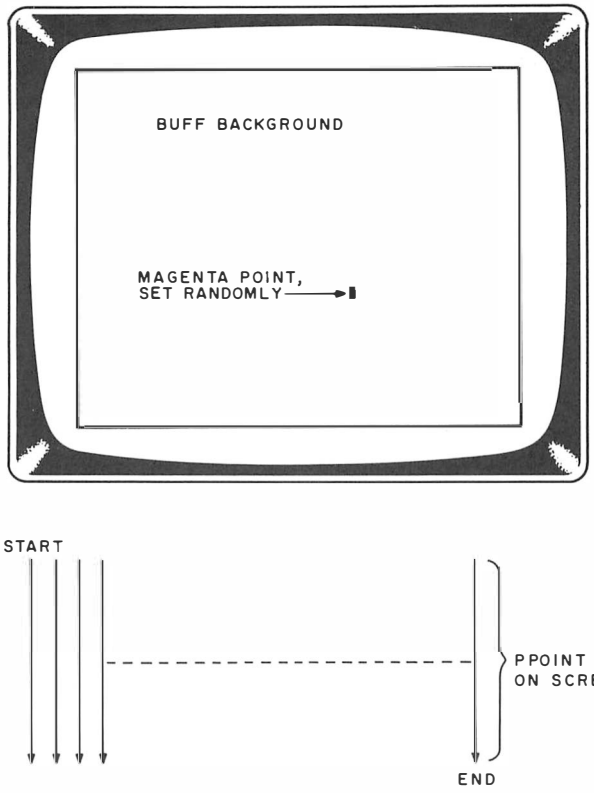


Figure 5.3: PPOINT use example.

TIME FOR PSET/PRESET

The scan above indicates a disadvantage of PSET/PRESET and PPOINT. All of these commands have fairly large overhead compared to the more powerful commands such as LINE and CIRCLE. It is much faster to draw a line with LINE, for example, than with a loop using PSET. The PSET/PRESET commands should be used only when you can't accomplish the same things with other Extended Color BASIC commands. One of the best uses for PSET/PRESET is in *plotting* data, which we'll discuss right now.

PLOTTING USING PSET/PRESET

In a plotting application, you can use the PSET/PRESET commands to plot the results of graphs. To do this, we must give some thought

about the *coordinates* of the graphics screen and the coordinates of the application. (We may be putting Descartes before de horse, but here goes...)

Most plotting is done in *Cartesian coordinates* as shown in Figure 5.4. In this system, there are *X* and *Y* axes.

The *X* axis is the horizontal axis. *X* values increase in a positive sense to the right, and decrease in a negative sense to the left. The *Y* axis is the vertical axis. *Y* values increase in a positive sense in the up direction, and decrease in a negative sense in the down direction. The intersection of the *X* and *Y* axes defines the *origin* of the graph, where *X* and *Y* are both zero.

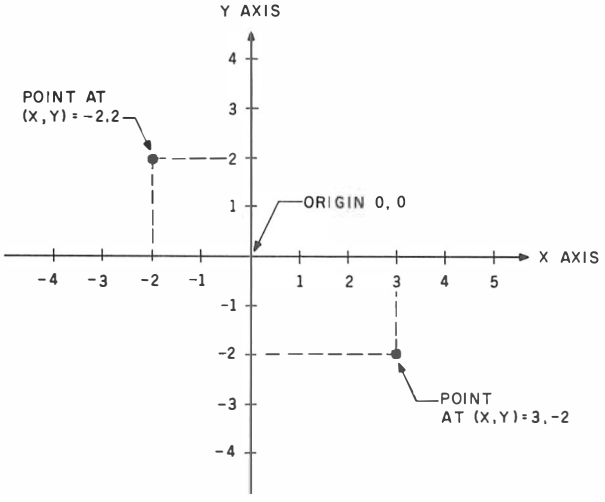


Figure 5.4: Cartesian coordinates.

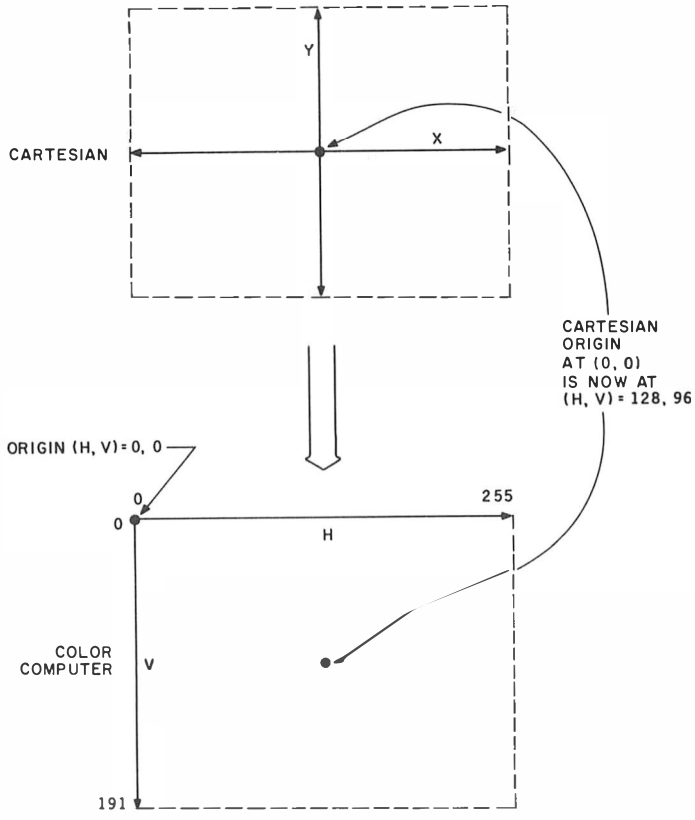


Figure 5.5: Conversion to Color Computer coordinates.

To convert from the *X,Y* coordinate system to the graphics of the Color Computer screen is a fairly easy task. It involves changing the *X,Y* coordinates to horizontal and vertical coordinates, as shown in Figure 5.5.

First of all, determine where the origin is to be on the screen. You might want a center origin for the display of a map, where the *X,Y* coordinates are referenced to the center of a state, as shown in Figure 5.6. For mathematical functions, you might also want a center origin where both positive and negative values are represented, as shown in the same figure. If you are displaying the elevation of a jogging course, you might want the origin to start in the left-hand corner, as shown in the figure. If you are displaying undersea mountain ranges, the origin might start at the midpoint of the top screen row.

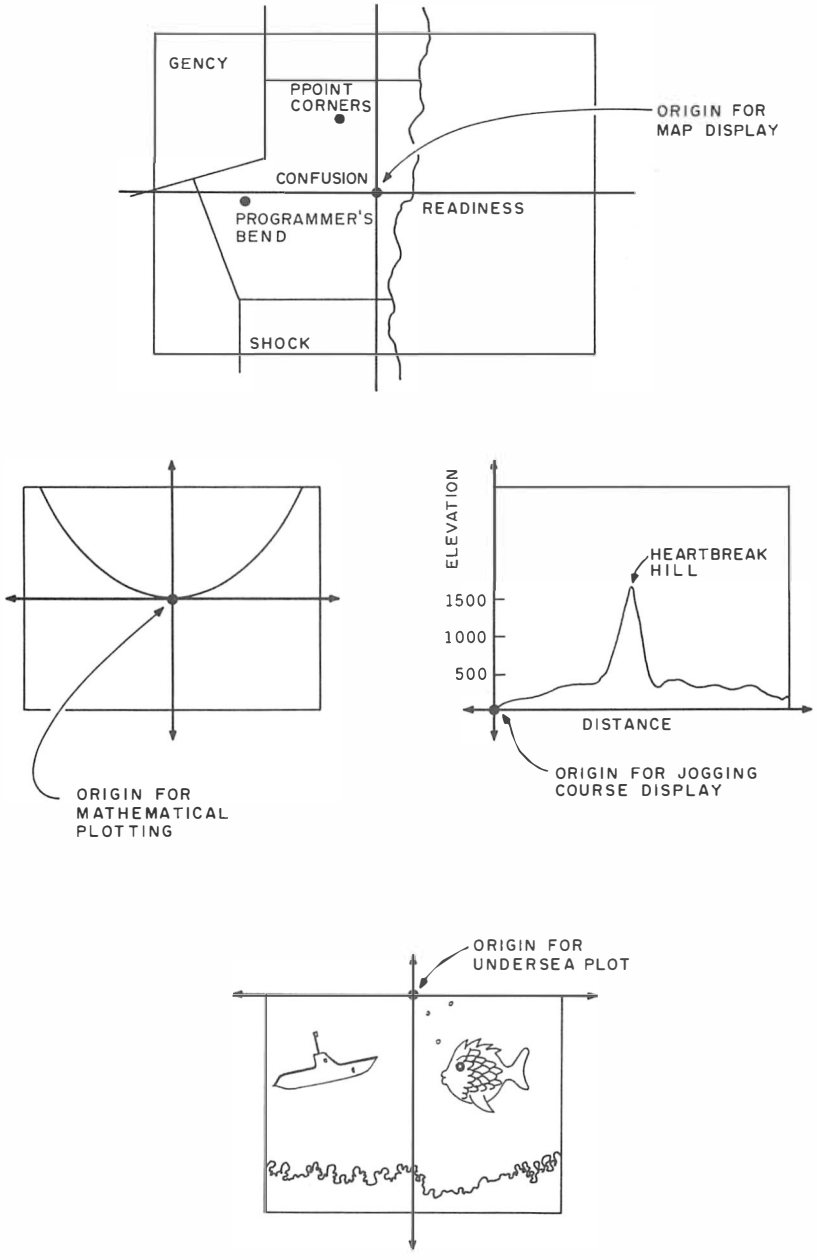


Figure 5.6: Determining the origin of a graph.

Next, determine the limits of the *X* and *Y* data you want displayed. The map coordinates might involve an area 400 miles by 200 miles. A mathematical graph might involve horizontal values of 720 units and vertical units of +1 to –1. The jogging course display might cover 26 miles, 385 yards and elevations from 0 to 500 feet.

To translate from *X,Y* coordinates to Color Computer screen coordinates, do the following:

1. Write down the maximum number of *X* units. Include both negative and positive ranges. If an *X,Y* plot goes from –50 to +50, for example, the maximum number of units is 100. Call this *XR*, for *X* range.
2. Divide 254 by *XR* for a new value *XS* (*X* scale factor). This is the number of *X* units for every horizontal unit on the screen minus one.
3. Write down the maximum number of *Y* units. Include both negative and positive ranges. If an *X,Y* plot goes from –10 to +10, for example, the maximum number of units is 20. Call this *YR*, for *Y* range.
4. Divide 191 by *YR* for a new value of *YS*, *Y* scale factor. This is the number of *Y* units for every vertical unit on the screen minus one.
5. Locate the origin that you want on the screen. Express it in *h,v* units. The origin will have a horizontal location of 0 through 255 and a vertical location of 0 through 191. Call these coordinates *H* and *V*.
6. Plot the values by

```
PSET (H,V,C)
```

where $H = H + X \times XR$, $V = V - Y \times YR$, and *C* is the PSET color value. Note that the equation for *V* has a minus sign.

As an example of this method, let's plot a sine graph. The sine waveform for 0 through 360 degrees is shown in Figure 5.7. It varies between 0 and +1 and 0 and –1, vertically. The *X* range, or *XR*, is in this case 360 degrees. The *XS*, or *X* scale factor is 254/360. The *Y* range, or *YR*, is 2 units. The *YS*, or *Y* scale factor, is 191/2.

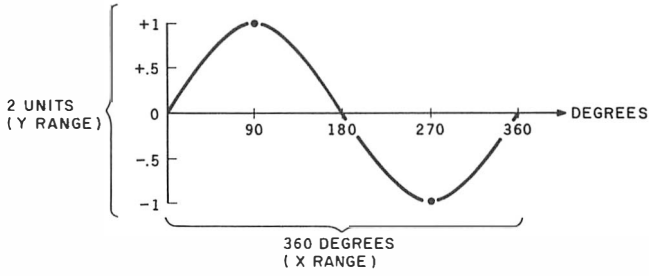


Figure 5.7: Sine wave example.

In this case there are no negative values of *X* (0 through 360 degrees), but there are both positive and negative values of *Y* (–1 through 1). We'll locate our origin at the midpoint of the left side of the screen, which is H = 0 and V = 96. *H* is therefore 0 and *V* is 96.

We are going to vary *X* from 0 through 360 and we should get values of *Y* from 0 to 1 back to 0 and down to –1 and then back to 0. The values of *Y* will be given by *Y* = SIN(X). A minor complication here, however... *X* is in radians, a unit of angular measurement which is equal to the number of degrees divided by 57.2957 (or 2 × PI/360). When we use this conversion, we have

```
Y=SIN(X/57.2957)
```

For our major loop in the program, then, it appears we have something like

```
170 FOR X=0 TO 360
180 H=H+X*XS
190 V=V-Y*YS
200 PSET(H,V,7)
210 NEXT X
```

Using the scale factor and origin values, we have

```
170 FOR X=0 TO 360
180 H=0+X*254/360
190 V=96-Y*(192/2)
200 PSET(H,V,7)
210 NEXT X
```

We also have to relate *Y* to *X* by the function Y = SIN(X/57.29578):

```
170 FOR X=0 TO 360
180 H=0+X*254/360
190 V=96-SIN(X/57.29578)*(192/2)
200 PSET(H,V,7)
210 NEXT X
```

When we integrate this with the graphics commands, the program looks like this:

```
100 REM PLOT SINE WAVE
110 PMODE 4,1
120 SCREEN 1,0
130 PCLS
170 FOR X=0 TO 360
180 H=0+X*254/360
190 V=96-SIN(X/57.29578)*(191/2)
200 PSET(H,V)
210 NEXT X
220 GOTO 220
```

The resulting display is shown in Figure 5.8.

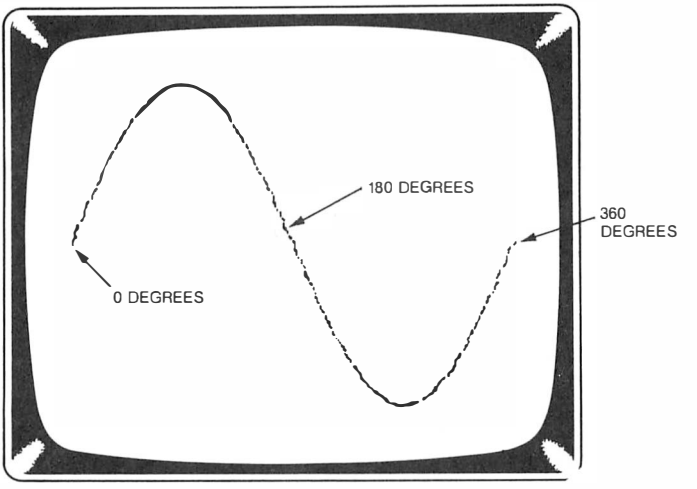


Figure 5.8: Sine wave plot.

In this program we took the approach of looping through the external range of *X*s — the range of degrees from 0 through 360. Another approach is to loop through the number of *H* or *V* points. In other words, instead of plotting 360 increments, why not plot 255 increments, one per horizontal unit? Depending upon the function that is being plotted, this can sometimes eliminate multiple PSETs of the same point. This approach is used in the program below, which plots multiple sine waves

over a user-specified number of degrees and a starting angular displacement (starting number of degrees).

```
100 REM PLOT MULTIPLE SINE WAVES
110 PMODE 4,1
120 PCLS
130 INPUT "NUMBER OF DEGREES";ND
140 INPUT "STARTING DEGREES";SD
150 ND=ND-SD
160 SCREEN 1,0
170 FOR X=0 TO 255
180 RD=(((X/255)*ND)+SD)/57.29578
190 PSET(X,96-SIN(RD)*95)
200 NEXT X
210 SCREEN 0,0
220 GOTO 130
```

In this program *X* is the number of the horizontal point and varies from 0 through 255. The range of *X* is variable and is an input parameter. It is set equal to *ND*, number of degrees. The starting number of degrees is input variable *ND*.

Initially, the PMODE is set equal to 4, the 255 - by - 192 two-color resolution. The graphics screen is then cleared. Note that this can be done even without displaying the screen. Statements 130 through 220 constitute the program loop. Each time through, the number of degrees to be displayed is input as *ND*, along with the starting point, *SD*. The SCREEN command then sets the graphic mode and the sine function is plotted. At the end of the plot, the screen is set back to text mode, and another pass is made. Note that the screen is only cleared at the beginning, and that subsequent passes plot the new plot over the old.


I'M BIG ON LINES, RECTANGLES, CIRCLES!
WITTEY

CHAPTER 6

Extended Color BASIC: Drawing Lines, Rectangles, Filled Rectangles, Circles, and Arcs with LINE and CIRCLE

Up to this point in *Color Computer Graphics*, we've discussed some of the more mundane commands. In this chapter and the next, we'll be discussing LINE, CIRCLE, DRAW, PAINT, and GET/PUT, the most powerful Extended Color BASIC commands that will enable you to draw a variety of lines, shapes, and figures and to color them easily.

This chapter discusses the LINE and CIRCLE commands which enable you to:

- Draw a straight line between any two points on the screen
- Draw a box of any size anywhere on the screen
- Draw a filled-in box of any size anywhere on the screen
- Draw a circle of any size anywhere on the screen
- Draw an arc of any size anywhere on the screen
- Draw an ellipse of any size anywhere on the screen

USING THE LINE COMMAND TO DRAW LINES

The LINE command allows you to draw a line between any two points on the screen. The format of the LINE for line drawing is

```
LINE (X1,Y1)-(X2,Y2),A
```

The *X1,Y1* and *X2,Y2* parameters define two points on the screen. These points may be in any relationship to each other. In other words, point 2 may be to the left or right of point 1 and above or below point 1. (About the only stipulation is that the points must be on the screen!) The points are defined by the standard *H,V* coordinates that we used in PSET, PRESET, and PPOINT — *X* is a value from 0 through 255 that specifies the horizontal coordinate; *Y* is a value from 0 through 192 that specifies the vertical coordinate. The coordinates reflect the screen matrix in the highest resolution mode, but, of course, are valid for any PMODE from 0 through 4.

The *A* coordinate is either PSET or PRESET. If PSET is used, then the current foreground color is used in drawing the line. If PRESET is used, then the current background color is used to draw the line. In other words, this command is really a PSET LINE or PRESET LINE, a command that draws or erases the line, depending upon whether the PSET or PRESET option is used.

That's all there is to drawing a line, any line! Those of you who are not impressed have probably never tried to implement a line-drawing routine in BASIC or assembly language. It is quite involved when the direction of the line, the limit conditions of the end points, and the increments are considered. LINE will do all of this automatically, leaving you free to do graphics applications.

To draw a line between a point at 23,23 and a point at 100,100, then, we'd have

```
LINE (23,23)-(100,100),PSET
```

The result would be as shown in Figure 6.1.

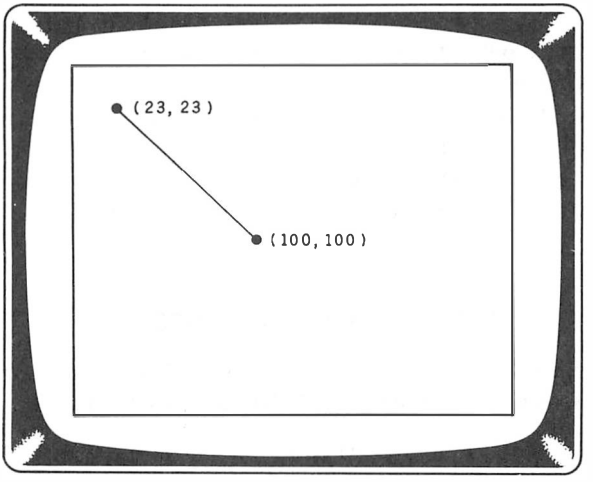


Figure 6.1: LINE example.

To show you that LINE does indeed work with any set of two points, run the following program. The program selects any two points at random and then uses LINE to connect them, using a random color. PMODE 3 (128 by 192 resolution) is used.

```
100 REM LINE DEMONSTRATION
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 X1=RND(256)-1
150 Y1=RND(192)-1
160 X2=RND(256)-1
170 Y2=RND(192)-1
180 C=RND(5)-1 : IF C=1 THEN GOTO 180
190 COLOR C,1
200 LINE (X1,Y1)-(X2,Y2),PSET
210 GOTO 140
```

Speed of LINE

How fast does LINE draw a line? To the eye it seems instantaneous. This is an interesting question, for with a great deal of graphics processing, execution speed may be a critical factor. We can easily find out by running this program

```
100 REM LINE TIMER
110 PMODE 4,1
120 SCREEN 1,0
130 PCLS
140 FOR Y1=0 TO 192
150 LINE (0,Y1)-(255,Y1),PSET
160 NEXT Y1
```

The program action is shown in Figure 6.2.

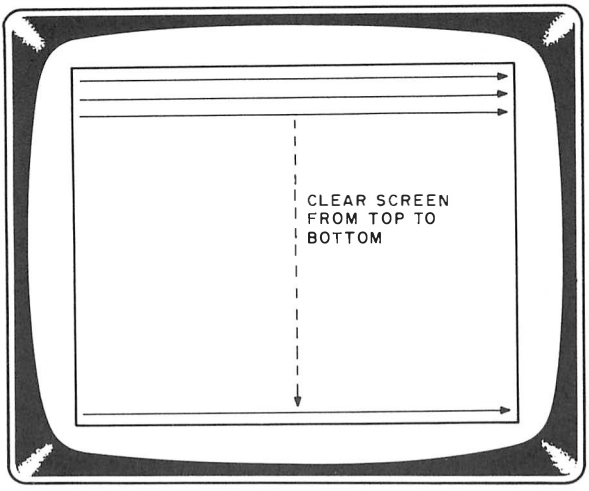


Figure 6.2: LINE timing display.

When this program is run and timed, we get about 8.3 seconds to clear the screen using LINE. If the statement in 150 is replaced by 150 REM AAAAAA, the time is about 1.3 seconds. The total time for LINE processing in this simple case is therefore about 7 seconds, or about 36 milliseconds (36/1000 second) per line. A similar test for LINE in PMODE 4 yields 519 random-length, random-direction lines in 50 seconds, or about one "average" line per 96 milliseconds. The worst case is probably about double this, or about 192 milliseconds. This is at least 20 times faster than the fastest BASIC line drawer!

Background and Foreground Colors in LINE

The PSET option uses the current foreground color to draw the line. This means that the foreground color must be changed by means of a COLOR statement to draw a line in a color other than the current PSET color. Using PRESET simply uses the current background color, which has the effect of erasing the line as this program illustrates:

```
100 REM LINE PRESET EXAMPLE
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 LINE (23,23)-(100,100),PSET
150 FOR I=0 TO 1000:NEXT I
160 LINE (23,23)-(100,100),PRESET
170 GOTO 170
```

USING LINE TO DRAW BOXES AND FILLED-IN BOXES

LINE Should really have been called LINEBOX (or maybe LINEBOXFILLEDINBOX), because it generates not only lines on the screen but also boxes (rectangles) and filled-in boxes. In this option, the two points of LINE define the opposing corners of a box as shown in Figure 6.3.

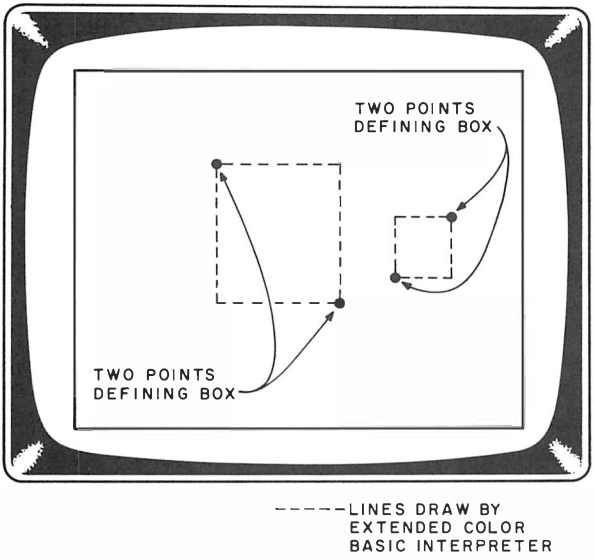


Figure 6.3: Using LINE to draw boxes.

The format of line for box generation is

```
LINE (X1,Y1)-(X2,Y2),PSET,B
```

or

```
LINE (X1,Y1)-(X2,Y2),PSET,BF
```

where B generates a box outline, and BF generates a box filled in with the current foreground color. The PRESET option can be used in place of PSET to reset the box to the background color.

Let's use the previous random line generator to generate boxes and filled-in boxes:

```
100 REM BOX DEMONSTRATION
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 X1=RND(256)-1
150 Y1=RND(192)-1
160 X2=(256)-1
170 Y2=RND(192)-1
180 C=RND(5)-1 : IF C=1 THEN GOTO 180
190 COLOR C,1
200 IF RND(2)=2 THEN LINE
    (X1,Y1)-(X2,Y2),PSET,B ELSE LINE
    (X1,Y1)-(X2,Y2),PSET,BF
210 GOTO 140
```

This program continually generates boxes and filled-in boxes of every size at every location (see Figure 6.4; as usual, this is a program of dubious practical value.) The overhead of generating a filled-in box is much greater than that of generating a line or outline of a box. Because of this, you can see that there is a definite direction to the fill, either up or down. Furthermore, the time in generating a filled-in box may go over one second for larger boxes.

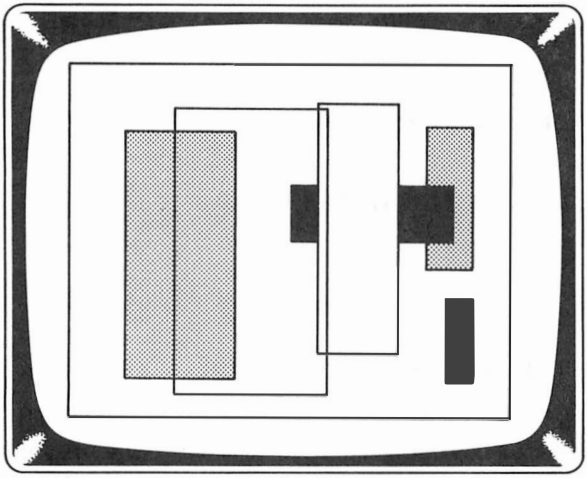


Figure 6.4: BOX example.

Direction of LINE

Some experimentation with LINE uncovers the fact that LINE operates just as you would expect. It draws a line from the beginning to end point and also generates a box from the horizontal line defined by the start point (*X1,Y1*) to the horizontal line defined by the end point (*X2,Y2*), as shown in Figure 6.5.

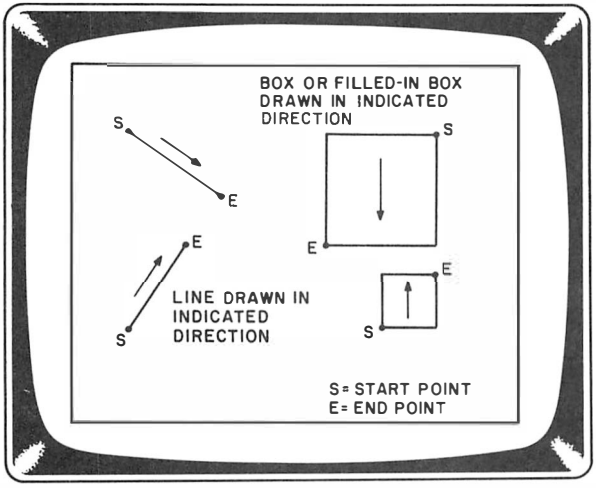


Figure 6.5: Direction of LINE.

Speed of LINE,BF

A test case program for LINE,BF generated 105 filled-in boxes in 50 seconds. The average filled-in box generation is therefore about ½ second, a significant time for a microcomputer action, but far less than it would take for the equivalent user code to perform such an action.

USING CIRCLE TO DRAW CIRCLES

The next Extended Color BASIC command that we'll discuss is CIRCLE. Circle can be used to draw not only circles, but also ellipses and arcs (CIRCLEELLIPSEARC, really).

The format of CIRCLE to draw a circle is

```
CIRCLE (X,Y),R,C
```

where *X* and *Y* are the horizontal and vertical coordinates of PMODE 4, the 256 horizontal by 192 vertical mode. *X* can therefore be 0 through 255 and *Y* can be 0 through 192. This coordinate defines the *center* of the circle, as shown in Figure 6.6.

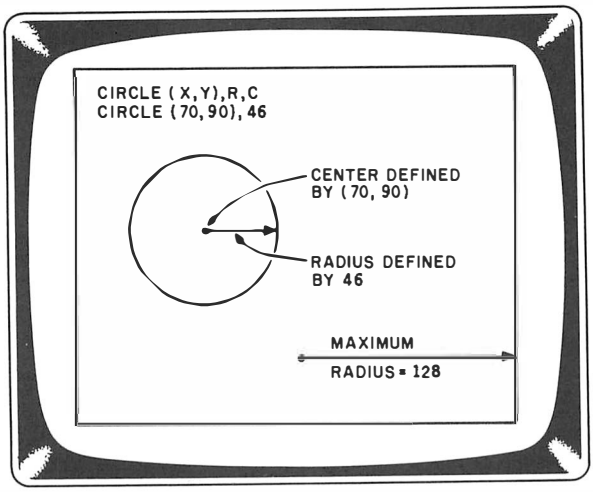


Figure 6.6: CIRCLE format.

The *R* parameter is the circle's *radius*, or distance from the center to any point on the circumference, as shown in the figure. The maximum radius for a display of a circle will be 127 for a center of 128,92. The radius may vary from 0 to 127 or above, although larger radii may result in border segments (discussed later in this chapter).

The *C* parameter is the color code for the foreground color. If this is omitted, the current foreground color is used. A color code that is the same as the background effectively causes a circle reset which erases a previously drawn circle.

To gain some familiarity with the CIRCLE, run the following program. It draws circles from the screen center with increasing radii, similar to what is shown in Figure 6.7.

```
100 REM CIRCLE DRAWER
110 PMODE 4,1
120 SCREEN 1,0
130 PCLS 1
140 FOR R=0 TO 200
```

```
150 CIRCLE (128,96),R,0
160 FOR I=0 TO 1000 : NEXT I
170 CIRCLE (128,96),R,1
180 NEXT R
```

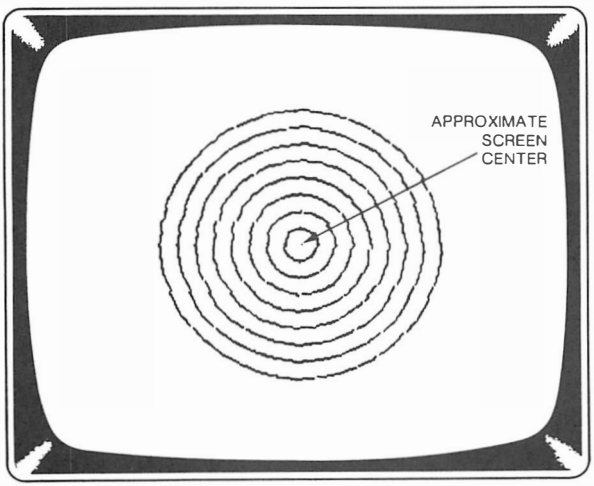


Figure 6.7: CIRCLE example.

There's one interesting thing about the circles created by this program. We specified a radius of up to 200; for the larger radii, the circles flattened out and finally occupied the border area of the screen, as shown in Figure 6.8.

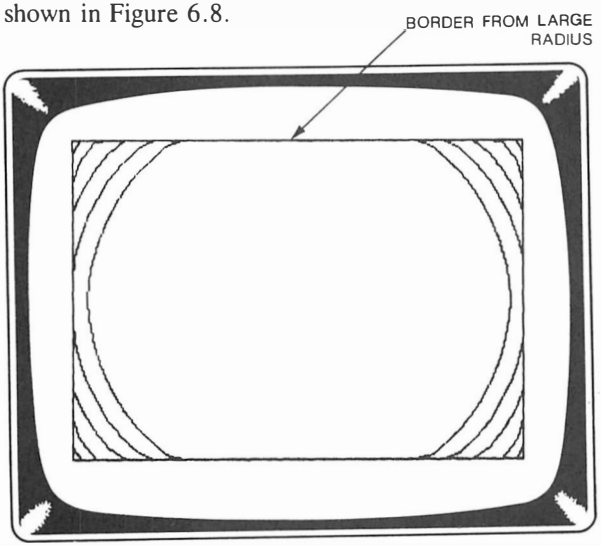


Figure 6.8: CIRCLE border case.

Height : Width Ratio

The normal aspect ratio of a television picture is 4 : 3, as shown in Figure 6.9. This figure means that the width of the picture is 4 units and that the height is 3 units. The VDG chip in the Color Computer divides the screen up into 256 by 192 pixels, which is also an aspect ratio of 4 : 3. A correctly adjusted television should show a perfectly round circle when the program above is run. Inexpensive televisions may produce an elongated circle which may have to be corrected by some adjustments to the controls on the back of the TV — vertical height and centering.

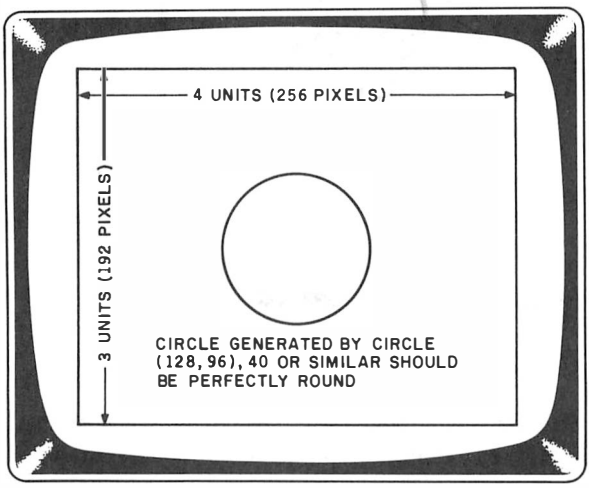


Figure 6.9: Aspect ratio

The height : width ratio for the circle is variable within the CIRCLE command, however. The format for a CIRCLE command with height : width ratio is

```
CIRCLE (X,Y)R,C,HW
```

where *HW* is the height : width ratio, expressed as a figure from 0 to 255. The default *HW* ratio is 1.

The *HW* ratio is just what the name says — the ratio of graphic height to width. A *HW* ratio of 1 says that the height of the circle will be 1 unit and the width of the circle will be 1 unit. Height divided by width is 1/1 and the *HW* ratio is 1.

Suppose that the height was 40 units and the width was 80 units. In this case the *HW* ratio would be 40/80, or .5, and the circle would appear as shown in Figure 6.10, a flattened circle, really an ellipse.

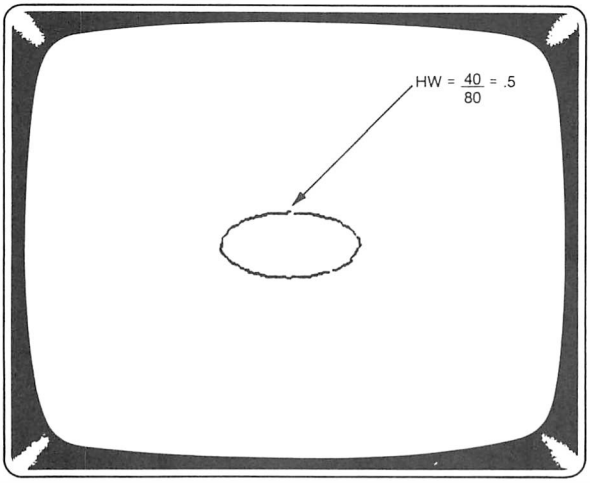


Figure 6.10: HW ratio example 1.

If the height was 80 and the width was 40, the circle would be squashed from the sides. The *HW* ratio would be 80/40, or 2, and the circle would appear as shown in Figure 6.11.

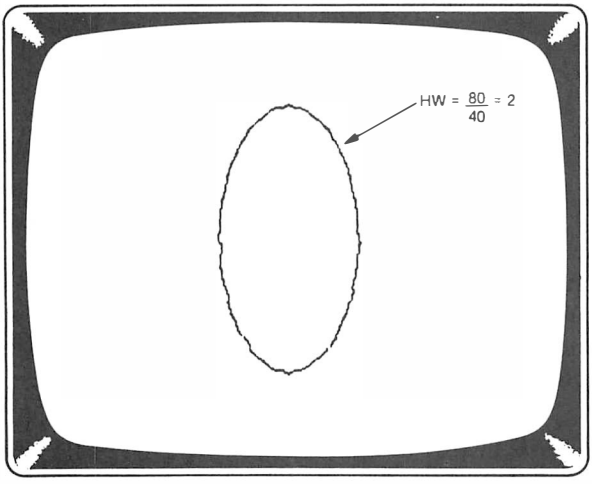


Figure 6.11: HW ratio example 2.

Cases from 0 through 3 are shown in Figure 6.12. These cases are generated by the program below.

```
100 REM HW RATIO FOR CIRCLES
110 PMODE 4,1
120 SCREEN 1,0
130 PCLS
140 FOR HW=0 TO 3 STEP .25
150 CIRCLE (128,96),50,1,HW
160 NEXT HW
170 GOTO 170
```

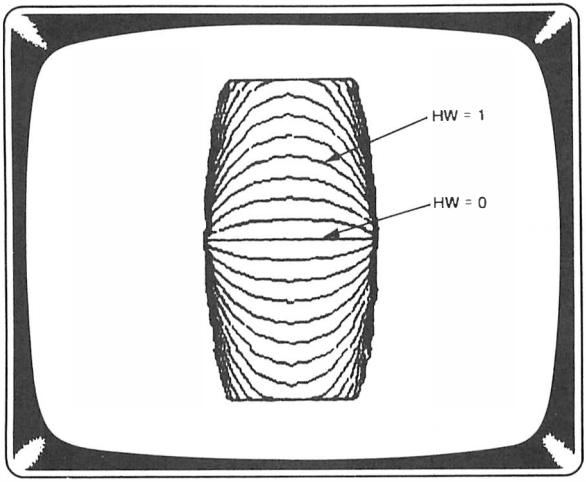


Figure 6.12: HW ratio program example.

You can see from the figure that an *HW* ratio of 0 produces a circle of no height — a horizontal line — and that an *HW* ratio greater than 2 or 3 draws a very elongated ellipse. An *HW* of 255 would with a small radius be close to a vertical line.

It's important to note that when the *HW* ratio is other than 1, the width of the circle is equal to the radius times 2 and the height of the circle is equal to the radius times 2 and the *HW* ratio. It may well have been the other way around, but one standard or the other had to be chosen.

When the circle produced by such an extreme *HW* ratio is drawn, any points that fall outside the border of the screen are drawn as being on the border. Which brings us to our next topic.

Borders for Oversized Circles

As you can see from Figure 6.8, any circle that falls outside of the border is drawn as the border itself. The center of the circle must appear within the screen display area, however, as there is no way to specify a center outside of screen limits!

Filling in Circles

Is there a way to fill in circles similar to the way we filled in rectangles? (Sorry, watercolors on the television are not allowed.) The PAINT command enables us to do this, but let's try another approach for the time being. We'll start with a circle of small diameter and increase an increment at a time to see the result.

```
100 REM FILLED-IN CIRCLE DRAWER
110 PMODE 4,1
120 SCREEN 1,0
130 PCLS 1
140 FOR R=0 TO 200
150 CIRCLE (128,96),R,0
160 NEXT R
```

The circle appeared as shown in Figure 6.13. It grew in size until it occupied the screen boundaries. As it grew, however, there were points that were not filled-in. Why is this? Because the increment of the radius can only be an integer value — 1, 2, 3, and so forth. However, it takes less than a one-unit increment to guarantee a fill of every pixel. So we can fill in circles by using this method, but we will have to live with some gaps in the result.

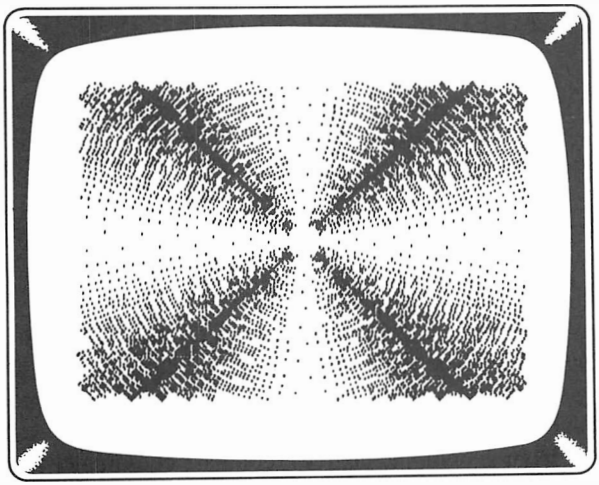


Figure 6.13: Filled-in circle example.

USING CIRCLE TO DRAW ARCS

CIRCLE can do more than draw circles, ellipses, or partial circles with boundary edges; it can also be used to draw arcs. Arcs are partial circumferences of circles, as shown in Figure 6.14.

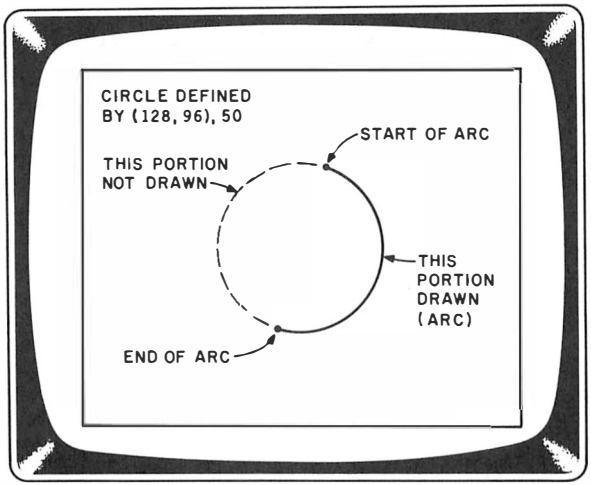


Figure 6.14: Drawing arcs.

The format of CIRCLE for drawing an arc is

```
CIRCLE (X,Y),R,C,HW,START,END
```

In this format, *START* and *END* specify the start and end of the arc, as shown in Figure 6.15. The circle is marked off in values from 0 through 1. The 0 point is at "three o'clock" (90 degrees geographic, 0 degrees trigonometric), the .25 point is six o'clock (180, –90), the .5 point is nine o'clock (270, –180), the .75 point is twelve o'clock (360, –270), and the 1 point is back to the start. When the circle is drawn, it is drawn clockwise from the starting point.

You know from using CIRCLE in previous examples that when the *START* and *END* are not specified, a complete circle or ellipse is drawn. To specify any segment (arc) of a circle or ellipse, put in the proper *START* and *END* and execute the circle command.

To convert from geographic degrees (compass coordinates from a map) to CIRCLE *START* and *END* values, take the number of degrees,

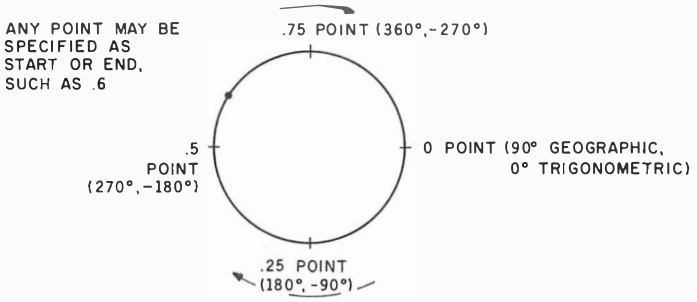


Figure 6.15: CIRCLE arc format.

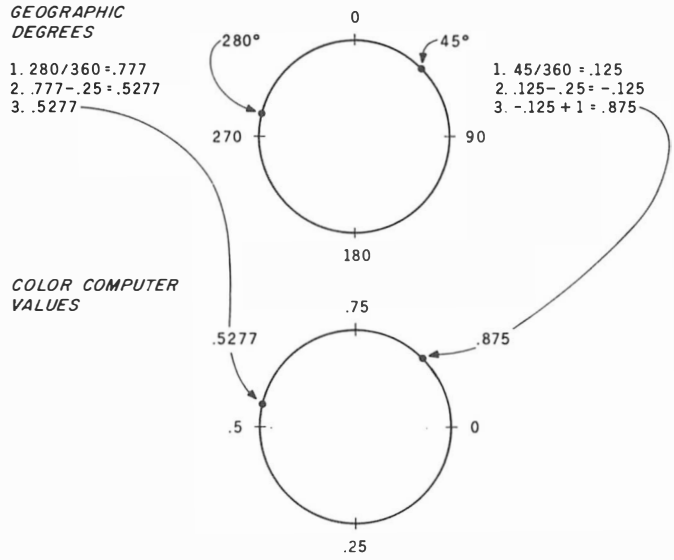


Figure 6.16: Converting from geographic to CIRCLE coordinates.

divide by 360, then subtract .25. If the result is negative, add 1. A compass heading of 180 degrees (due south), for example, would be 180/360 – .25, or .5 – .25 = .25. A compass heading of 45 degrees (northeast) would be 45/360 – .25 = .125 – .25 = –.125; the result is negative, so adding one gives us .75. Figure 6.16 shows the process.

In trigonometric representation, a circle starts at 0 degrees in the three o'clock position and proceeds counterclockwise through 90 degrees (twelve o'clock), 180 degrees (nine o'clock), 270 degrees (six o'clock), and back to 360 degrees or 0. The angular displacement may also proceed clockwise from 0 degrees in a negative sense : –90 degrees (six o'clock), –180 degrees (nine o'clock), –270 degrees (twelve o'clock), and back to –360 or 0 degrees. In many cases it may keep on going through greater and greater angular displacements — 720 degrees, 1080 degrees, and so forth. Figure 6.17 shows the scheme.

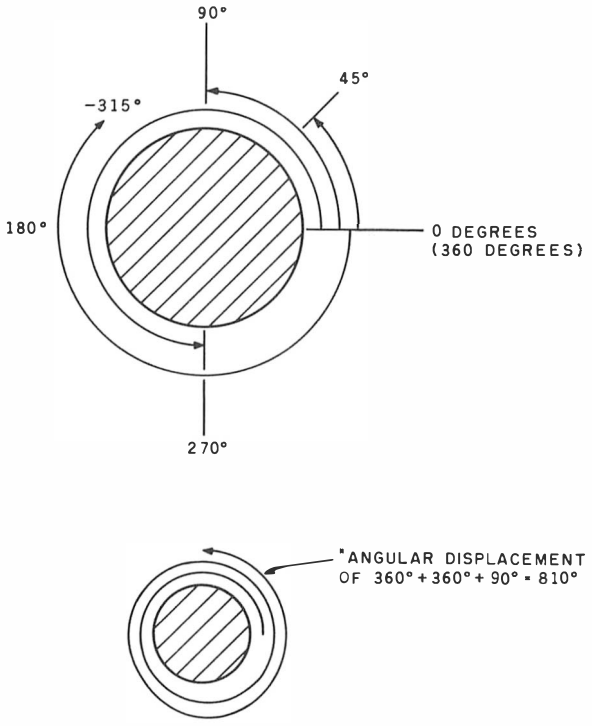


Figure 6.17: Trigonometric representation.

To convert from positive trigonometric degrees (less than or equal to 360) to *START* and *END* representation, take the trigonometric degrees and divide by 360. Make this number negative. Add one to it. A trigonometric value of +90 degrees, for example, is –(90/360) + 1 = –.25 + 1 = .75. A trigonometric value of +270 degrees is –(270/360) + 1 = –.75 + 1 = .25. To convert from negative trigonometric degrees (less than or equal to 360), change to the equivalent positive

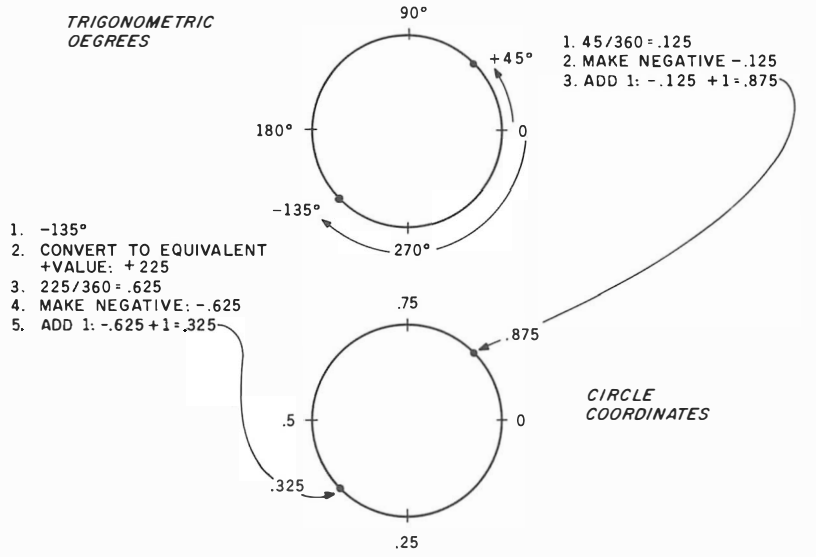


Figure 6.18: Converting from trigonometric to CIRCLE coordinates.

value and then convert as above (–90 = +270; –180 = +180; –270 = +90; –360 = +360). To convert from large angular displacements (greater than 360), subtract 360 until the result is less than or equal to 360, and then convert as above. Figure 6.18 shows the conversion method.

Now that we know how to convert into *START* and *END* values, let's plot some arcs. We'll use one-quarter arcs of a circle stepping from 0 (three o'clock) to .75 (twelve o'clock), as shown in Figure 6.19.

```
100 REM PLOT ARCS
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 ST=0
150 FOR X=35 TO 215 STEP 60
160 CIRCLE (X,96),20,4,1,ST,ST+.25
170 ST=ST+.25
180 NEXT X
190 GOTO 190
```

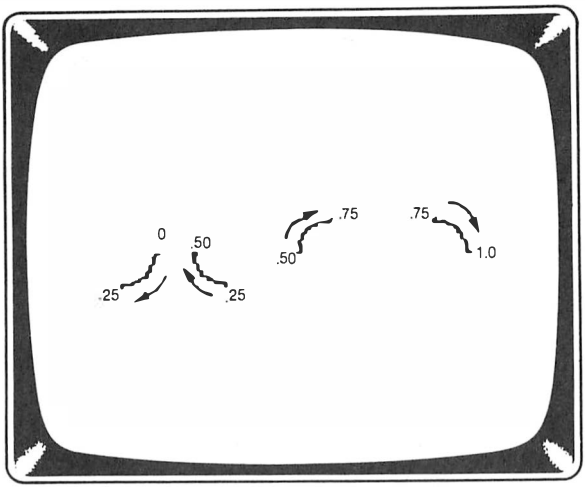


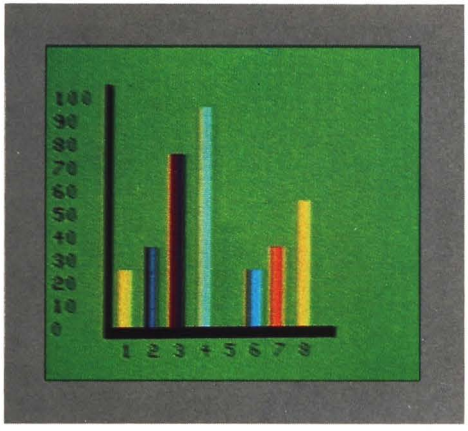
Figure 6.19: Arc example.

The CIRCLE statement above can be broken down as follows: The centers for the four circles are at X,96, or 35,96; 95,96; 155,96; and 215,96. The radius is fixed at 20. The color is red (background is green from the PCLS), the *HW* ratio is 1 (perfect circle). The *STARTs* are 0, .25, .5, and .75. The *ENDs* are .25, .5, .75, and 1.

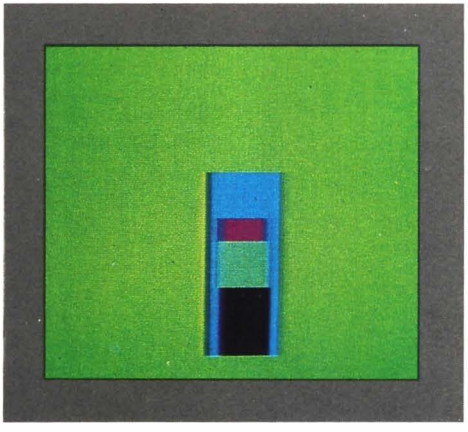
Arcs can also be drawn for ellipses. In this case the HW ratio would define the eccentricity of the ellipse. The program below draws a series of arcs from the six o'clock point to the nine o'clock point for *HW* values of .25 through 3 in steps of .25. The result is shown in Figure 6.20.

```
100 REM ARCS WITH VARYING HW RATIOS
110 PMODE 4,1
120 SCREEN 1,0
130 PCLS
140 FOR HW=.25 TO 3 STEP .25
150 CIRCLE (128,96),50,1,HW,.25,.5
160 NEXT HW
170 GOTO 170
```

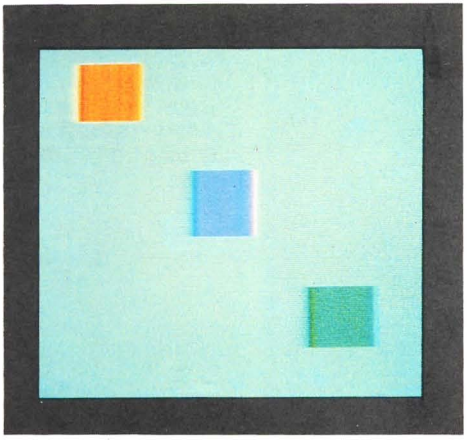
1 SEMIGRAPHIC 4 MODE
An example of the Color BASIC mode of 64 pixels horizontally by 32 pixels vertically, this photograph shows available colors. (Chapter 1)



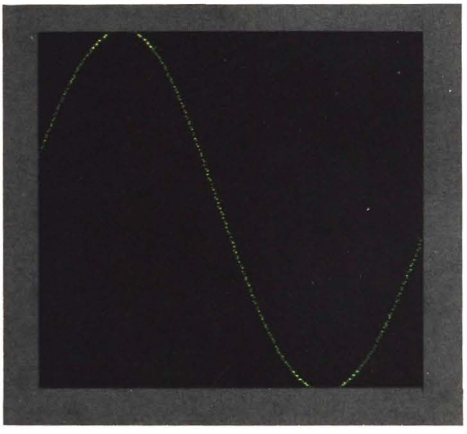
2 BAR GRAPH PLOT
Here we are using SET/RESET to have up to eight colors on a background, with text as well. (Chapter 3)



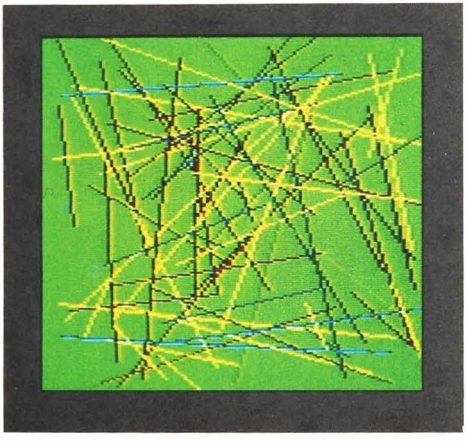
3 ANIMATION USING CHR$ STRINGS
This program utilizes graphics strings in animation. The figure shown is a cylinder and piston of a two-stroke engine. It fires with a red explosion and the piston moves up and down. (Chapter 3)



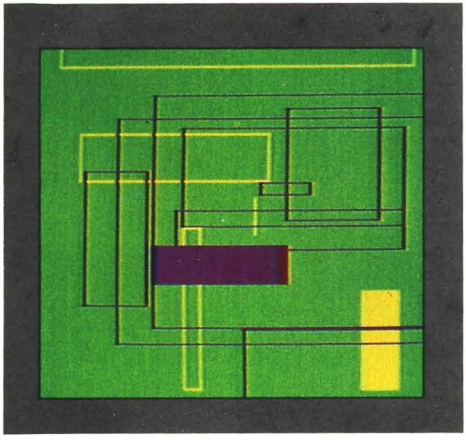
4 USE OF THE COLOR COMMAND
This example of the COLOR command paints three rectangles on a buff background. (Chapter 4)



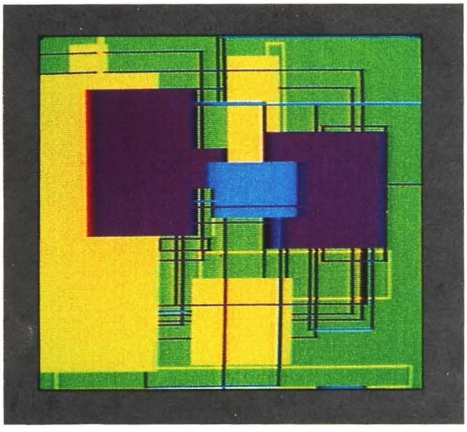
5 PLOT A SINE WAVE
Using PSET to plot a sine wave, we are also converting from Cartesian coordinates to Color Computer coordinates. (Chapter 5)



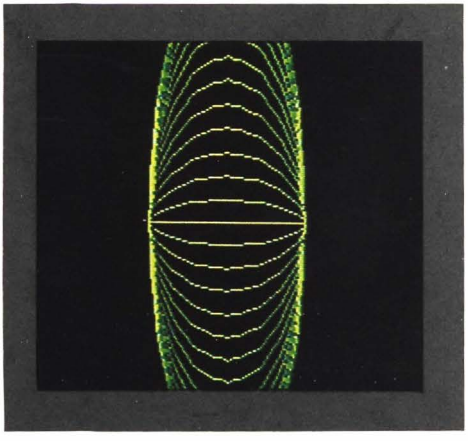
6 LINE DEMONSTRATION
Here we select any two points at random and use the LINE command to connect them, using a random color. (Chapter 6)



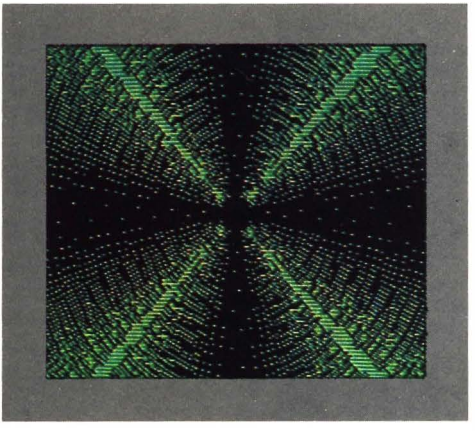
7a BOX DEMONSTRATION
The LINE command can also be used to generate boxes and filled-in boxes.
(Chapter 6)



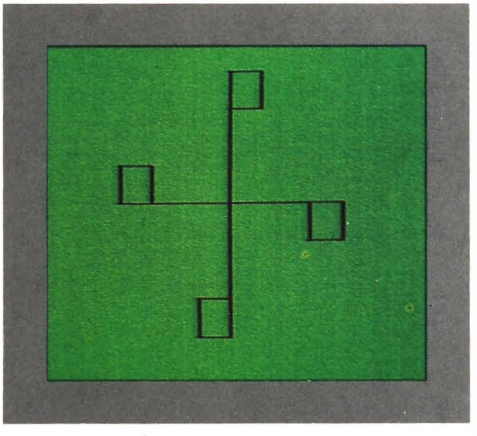
7b BOX DEMONSTRATION continued
This shows the BOX program at a later stage.
(Chapter 6)



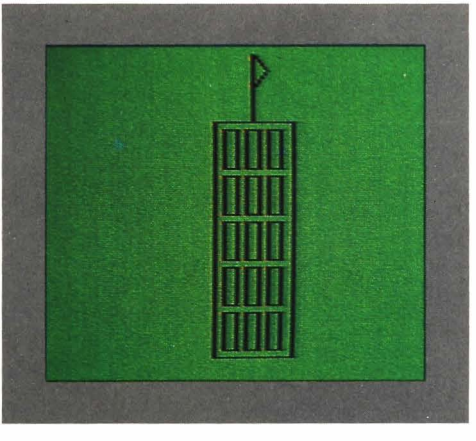
8 HEIGHT: WIDTH RATIO FOR CIRCLES
Varying the height to width ratio in the CIRCLE command results in ellipses.
(Chapter 6)



9 FILLED-IN CIRCLE DRAWER
Here we fill in a circle without using the PAINT command. (Chapter 6)



10 ROTATE PROGRAM
Using the DRAW commands, including the Angle command, we can draw a figure, then rotate and draw it again. (Chapter 7)



11 DRAW THE TANDY TOWER
This is an example of using strings and substrings with the DRAW commands to copy a repetitive shape. (Chapter 7)

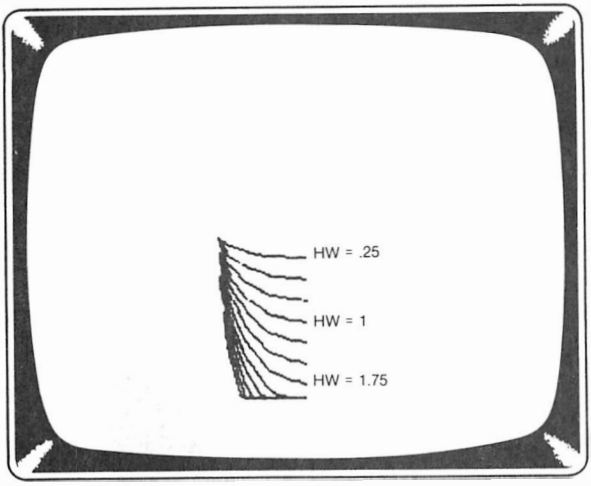


Figure 6.20: Arcs for ellipses example.

CIRCLE Defaults

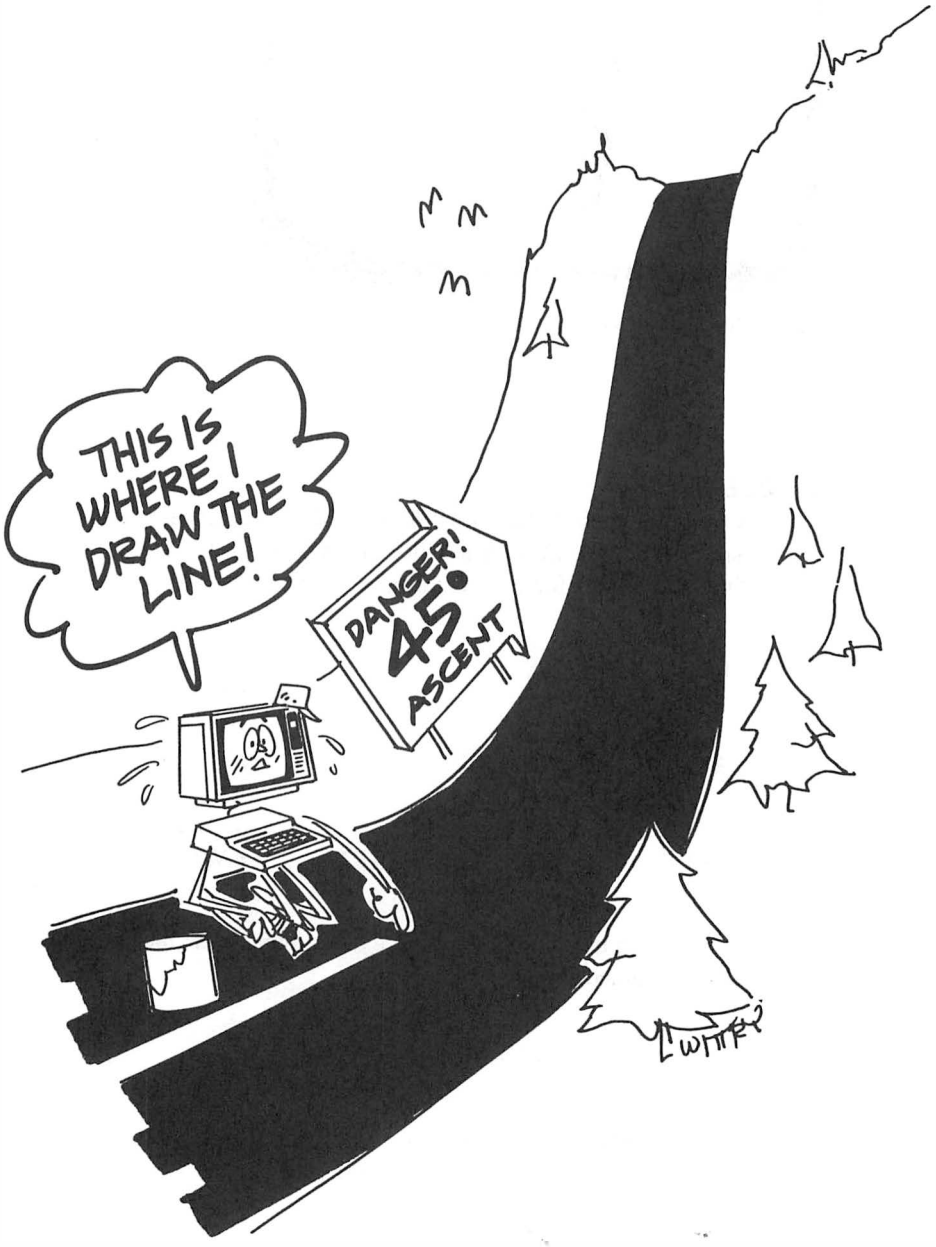
As we've discussed, the format of the CIRCLE command varies from a simple CIRCLE (*X*,*Y*),*R* to CIRCLE (*X*,*Y*),*R*,*C*,*HW*,*START*,*END*. The default parameters for various formats are shown below:

`CIRCLE (X,Y),R`	*C* = foreground, *HW* = 1, *START* = 0, *END* = 1
`CIRCLE (X,Y),R,C`	*HW* = 1, *START* = 0, *END* = 1
`CIRCLE (X,Y),R,C,HW`	*START* = 0, *END* = 1
`CIRCLE (X,Y),R,C,HW,START`	*END* = 1

If a comma is used in place of specifying a parameter, the default parameter is assumed. The statement

```
CIRCLE (X,Y),R,,HW,START,END
```

for example, would generate a circle with the foreground color. The *HW* ratio must be specified for *START* and *END*.


THIS IS WHERE I DRAW THE LINE!
DANGER! 45° ASCENT

CHAPTER 7

Extended Color BASIC: Using the DRAW Command

The DRAW command is somewhat related to the LINE command. It draws a line from point 1 to point 2. The line may be drawn in an up direction, at 45 degrees, to the right, at 135 degrees, down, at 225 degrees, or left, at 315 degrees, as shown in Figure 7.1.

The length of the line may be any number of units, from 0 to hundreds.

The DRAW command also includes the capability to move to a designated spot on the screen, specified in *X,Y* coordinates or in relative coordinates.

DRAW also has the provision of drawing a blank line (!) or changing the color.

The DRAW command uses a string of commands to indicate how the line should be drawn. A typical line might be defined as DRAW "up 25 units," "right 20 units," and so forth. Within the commands, DRAW may reference a substring of commands, a powerful feature that enables the user to build little modules of commands.

Another powerful DRAW feature is the ability to scale DRAW commands. This means that with a single command, DRAW may produce displays that are 1/4 scale, 1/2 scale, on up to 62/4 scale.

The DRAW command is really an order of magnitude more powerful than LINE, which was an order of magnitude more powerful than PSET! (Which was an order of magnitude more powerful than SET, which was ...) We'll discuss all of the ramifications of DRAW in this chapter.

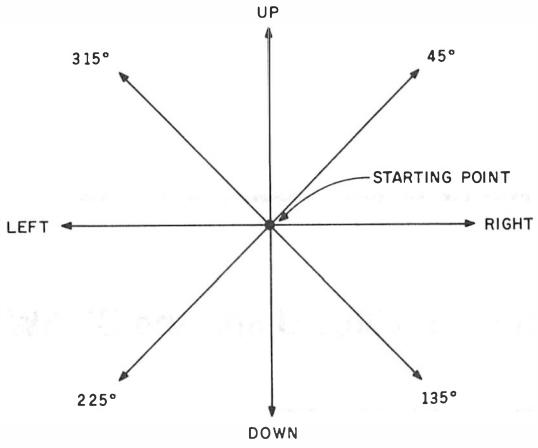


Figure 7.1: DRAW directions.

DRAW FORMAT

The format of DRAW is

```
DRAW "string"
```

or

```
DRAW A$
```

In the first case, "string" represents a string of DRAW commands; the string may be any number from one to dozens. A typical DRAW string might be

```
DRAW "D20; R25; U23"
```

where the string represents a line drawn from the current screen position, Down 20, Right 25, and then Up 23 positions. Note that the individual commands are separated by semicolons and that the entire line of command is a string with double quotation marks enclosing the line.

In the second case, A$ represents a general string variable, which might be A$, B$, CC$, YY$, or some other legitimate name. This string variable in turn would define a string of valid DRAW commands.

DRAW COMMANDS

DRAW commands are shown in Table 7.1. They include *motion* commands which result in a line segment being generated, *mode* commands which change the color, angle, or scaling, and two options, "no update" and "blank."

Motion	
M	= Move the cursor
U	= Move Up
D	= Move Down
L	= Move Left
R	= Move Right
E	= Move 45 degrees
F	= Move 135 degrees
G	= Move 225 degrees
H	= Move 315 degrees
X	= EXecute a substring
Mode	
C	= Change Color
A	= Change Angle (rotate)
S	= Change Scale
Options	
N	= No Update after draw action
B	= Draw Blank line (move, do not display)

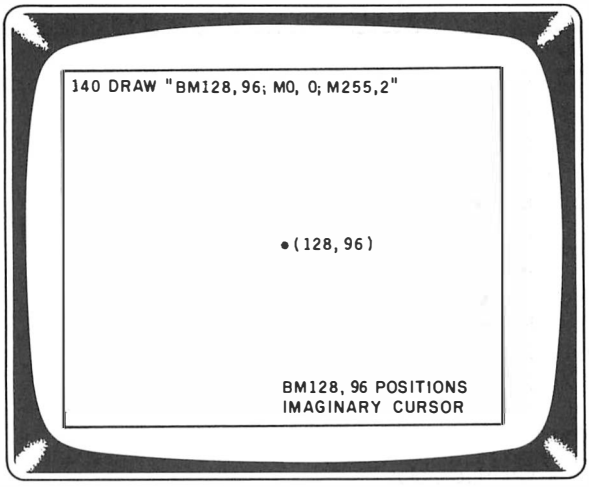
Table 7.1: DRAW Commands.

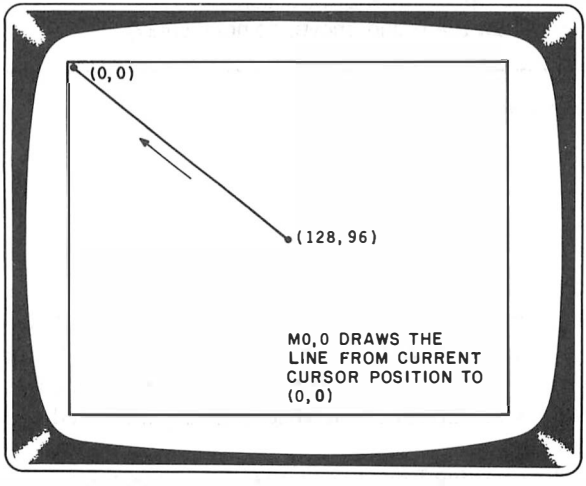
M Command

The M command moves the imaginary cursor for the DRAW to a specified spot on the screen. Initially, this is done with the "B" (or Blank) option so that wherever the cursor is to start off, no line is generated in the move. Thereafter, an M command can be used with the B prefix or without.

To draw a line starting from 128,96 to 0,0 and then to 255,2, for example, the line is first positioned by BM128,96 and then M0,0 and M255,2 are done to draw the lines. (See Figure 7.2).

```
100 REM DRAW EXAMPLE 1
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 DRAW "BM128,96;M0,0;M255,2"
150 GOTO 150
```





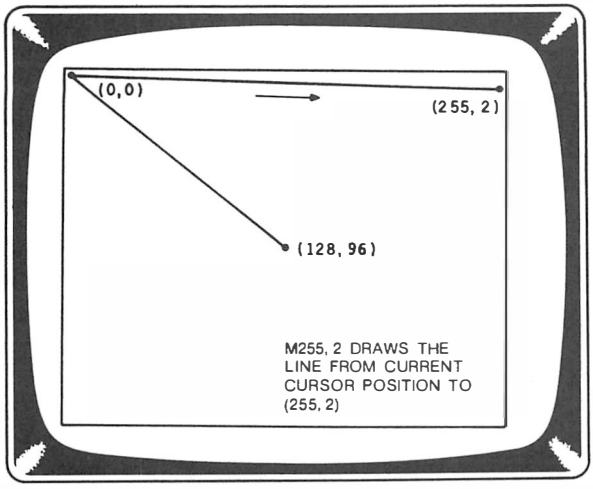


Figure 7.2: M command using absolute motion.

The M command, therefore, is used to position the cursor or to draw a line between any two specified screen points (just as the LINE command was used). M will generate any angled line and draw the line in any direction depending upon the relationship of the start point to the ending point. Note that the screen points are explicitly identified in the format (B)MX,Y, where (B) indicates a Blank option.

Another form of the M command uses relative rather than absolute values for *X* and *Y*. After the first M command is executed, the graphics cursor is in a known position. A Move may be done with a displacement from that known position to any other position. The displacement may be either negative or positive. The program above may be rewritten as

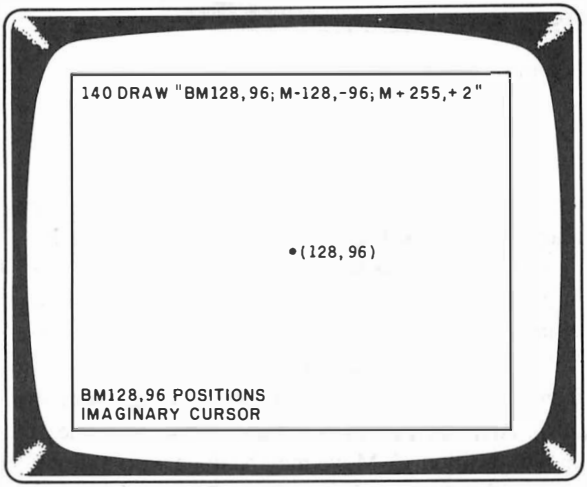
```
100 REM DRAW EXAMPLE 1
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 DRAW "BM128,96;M-128,-96;M+255,+2"
150 GOTO 150
```

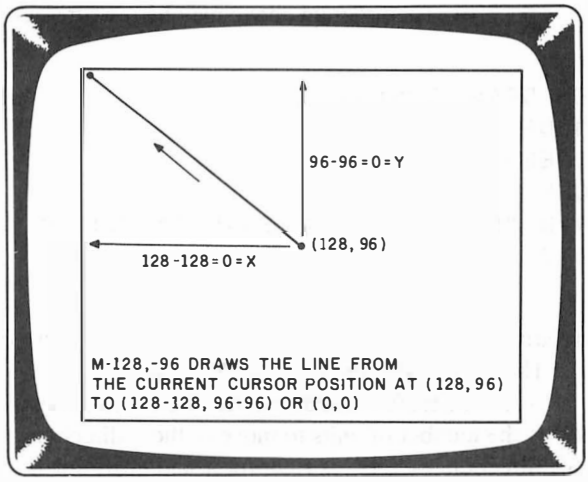
This is a relative move. (Please, no jokes about the proximity of mothers-in-law.) The format for this type of relative move is (B)M + *–XD*,+ *–YD*, where *XD* is the number of units to move in the *X* direction and *YD* is the number of units to move in the *Y* direction. The sign indicates the direction — "+" is right for *X* and down for *Y* and

"–" is left for *X* and up for *Y*. The displacement value will be added to the current cursor position, and the result will define the new point. The *Y* positive sign is optional in the above format; it's probably best, however, to always use a plus or negative sign when working in this format to avoid confusion.

In the above program, BM128,96 moved to 128,96 without writing a line. The next Move was M –128,–96 which gave the new value 128 – 128, 96 – 96, or 0, 0. The next Move was M +255 ,+2 which gave the new value 0 + 255, 0 + 2, or 255, 2. You can look at the relative displacements, then, as either being added to the current cursor position, or moving in a "+" direction or "–" direction for *X* or *Y*.

Figure 7.3 shows the movements.





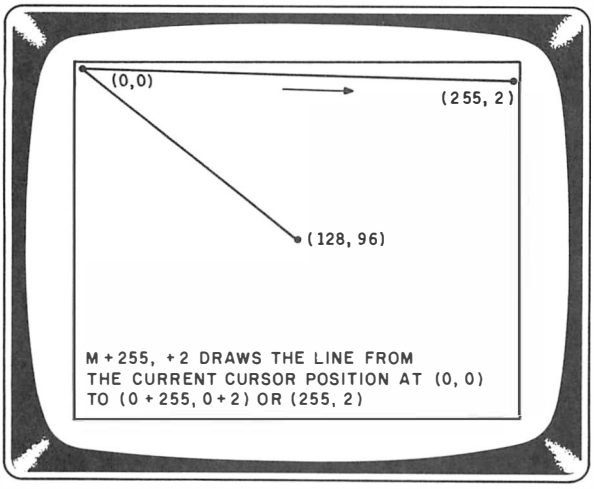


Figure 7.3: M command using relative motion.

Other Motion Commands

The other motion commands are all relative commands that specify a relative displacement from the current cursor position. (Remember that the cursor is imaginary — it doesn't show up and we're using it only for convenience.) Up, Down, Right, and Left are easy to remember, as the first letter of each direction is used as an abbreviation for the command. E, F, G, and H fill in the gaps between these directions as shown in Figure 7.4.

These commands allows us to easily specify movement in eight directions. Since much graphics will involve drawing lines in this fashion, they are much more convenient to use than specifying a new Move command for each line segment. Suppose, for example, that we wanted to draw a large letter "M" in the center of the screen, as shown in Figure 7.5. We could draw it very easily by using the DRAW motion commands:

```
100 REM DRAW EXAMPLE 3
110 MODE 3,1
120 SCREEN 1,0
130 PCLS
140 DRAW "BM128,96; E15; R10; D30; L12;
    U18; G13; H13; D18; L12; U30; R10;
    F15"
150 GOTO 150
```

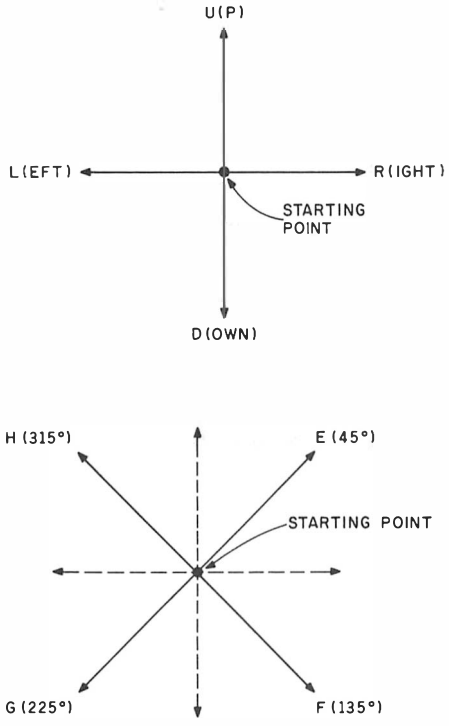


Figure 7.4: Other motion commands.

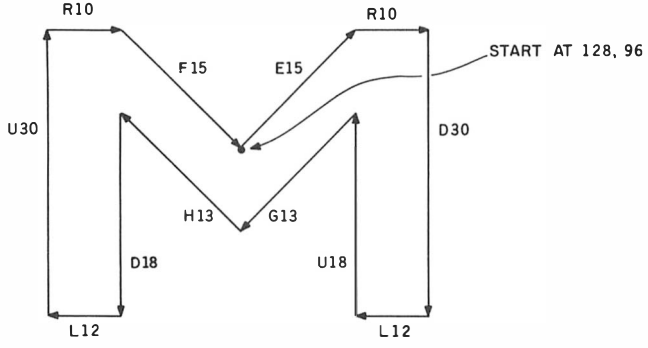


Figure 7.5: Using the motion commands.

About the only difficulty in using the DRAW motion commands is that the commands that specify motion in acute angles are referenced to the number of pixel units displaced horizontally or vertically. Remember that the number of units of length for a line drawn at a 45 degree angle is approximately 1.42 times the length of a vertical or horizontal line with the same displacement along the *Y* or *X* direction. See Figure 7.6.

The M command can easily be used to connect to any horizontal or vertical line, since you need only to know the horizontal or vertical displacement rather than the actual length of the angled line.

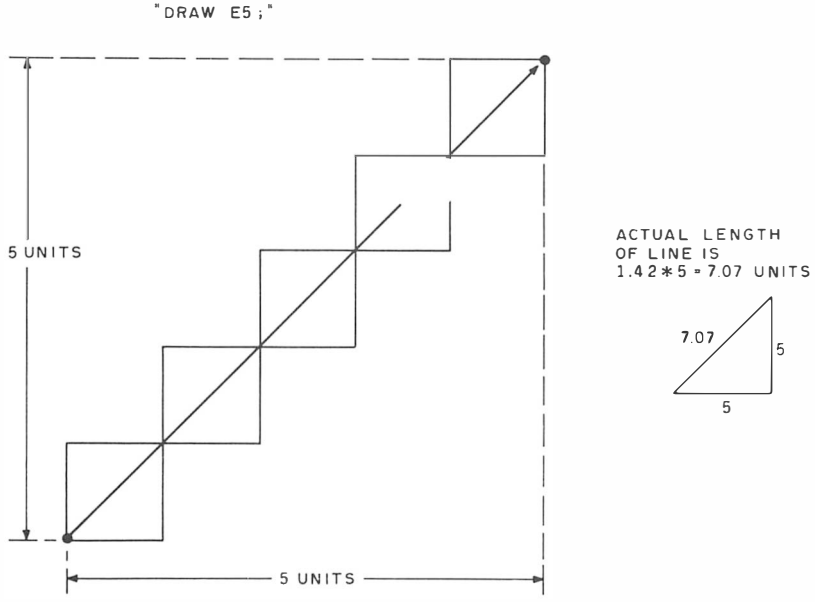


Figure 7.6: Acute angle displacements.

The Blank Command

We saw the use of the Blank command used as a prefix for the Move command to effect a move without drawing a line. The "B" prefix may also be used before any motion command to blank out a line. Simply tack it on before the command as in "BH15;" in the above program. The line will still be drawn, but it will be blank. (Some of you will be able to relate this to the philosophical question about a tree falling in the woods with no one around. If a line is DRAWn with a Blank prefix, is it really drawn?)

An alternative way to use the Blank command is to use the B command separately, without prefixing another command. An example from above is

```
140 DRAW "BM128,96; E15; R10; D30; L12;
    U18; G13; B; H13; D18; L12; U30; R10;
    F15"
```

The Color Command

The C command is used at any time in the DRAW string to change the color of the following line. The format of C is

C*x*

where *x* is a standard color code from 0 through 8. The C*x* command is embedded in the string just as the other commands are, with a semicolon following the C*x*.

Using the M program from above, you can alternate colors on the line segments quite easily. As with all graphics, you have only the four colors of the color set to play with in PMODE 1 or PMODE 3.

```
100 REM DRAW EXAMPLE 3
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 DRAW "BM128,96; C2; E15; R10; C1;
    D30; L12; C2; U18; G13; C3; H13; D18;
    C4; L12; U30; C2; R10; F15"
150 GOTO 150
```

The colors for the M are as shown in Figure 7.7.

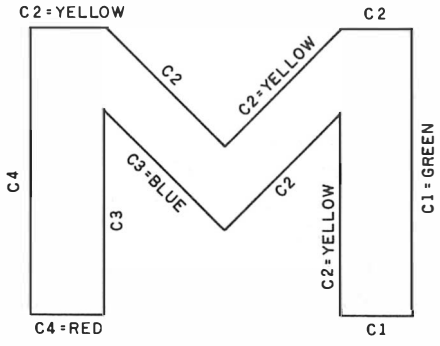


Figure 7.7: Using the color command.

Once a color is specified by the C command, it will remain in force until the C command is encountered.

The No Update Command

The N, or No Update command, is another optional command that may be used similarly to the B command, either as a prefix to a motion command or as a separate command delimited by a semicolon.

The N command allows the next motion command to be executed but prevents it from updating the cursor position. In other words, the next line is drawn, but the cursor remains at the beginning of the line after the line is drawn, as shown in Figure 7.8.

```
110   DRAW "U40; R30"
120   DRAW "ND20;"
130   DRAW "R30;"
```

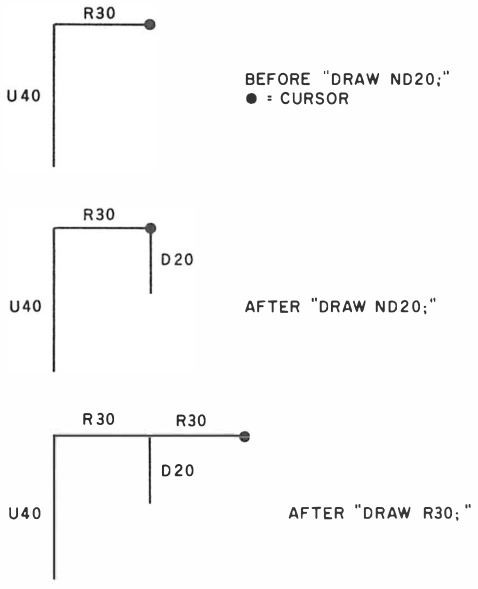


Figure 7.8: Using the N command.

We can see how the N command works in displaying a giant asterisk in the center of the screen, as shown in Figure 7.9. Both forms of the N command, the prefix and separate command, are used.

```
100 REM DRAW EXAMPLE 3
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 DRAW "BM128,96; NU40; NR28; NR40;
    NF28; ND40; NG28; NL40; NH28"
150 GOTO 150
```

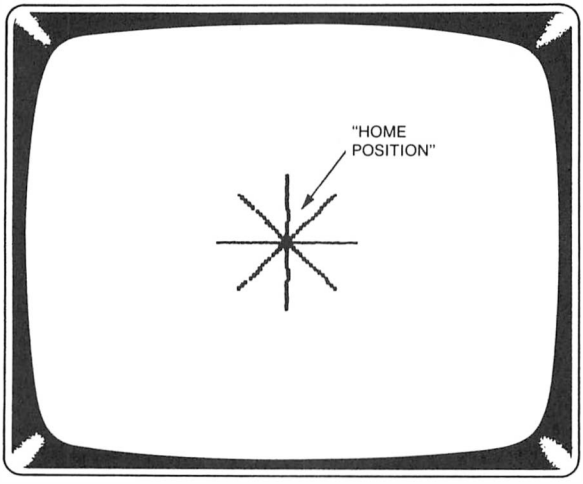


Figure 7.9: N command example.

The Angle Command

The A, or Angle command, is somewhat of a misnomer. It is really a rotate command. The format of the command is

A*x*

where *x* is 0, 1, 2, or 3, specifying rotations of 0 degrees, 90 degrees, 180 degrees, or 270 degrees. When the command is invoked, all subsequent lines will be rotated to the new position.

To see how this works, look at the program below. The figure in the normal position is shown in Figure 7.10. It is drawn by "BM128,96; U70; R20; D20; L20". The program draws four versions of the figure by using four A commands. The first A command, "A0", displays the

figure in the 0 degree position. The "A1" rotates the figure 90 degrees clockwise and displays it. The "A2" rotates the figure 180 degrees and displays it. The "A3" command rotates the figure 270 degrees and displays it. The final display is shown in Figure 7.11.

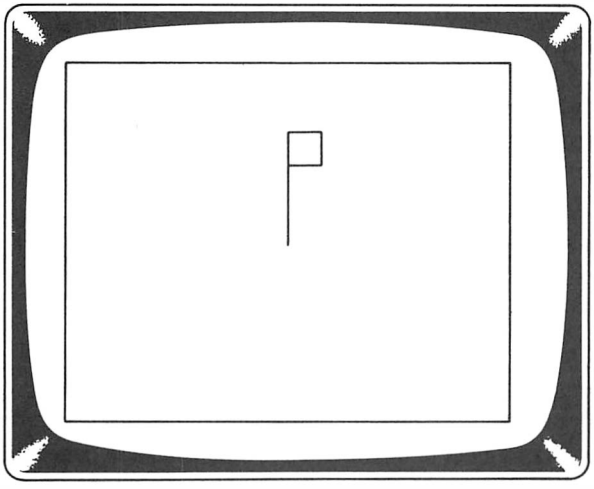


Figure 7.10: Angle command example 1.

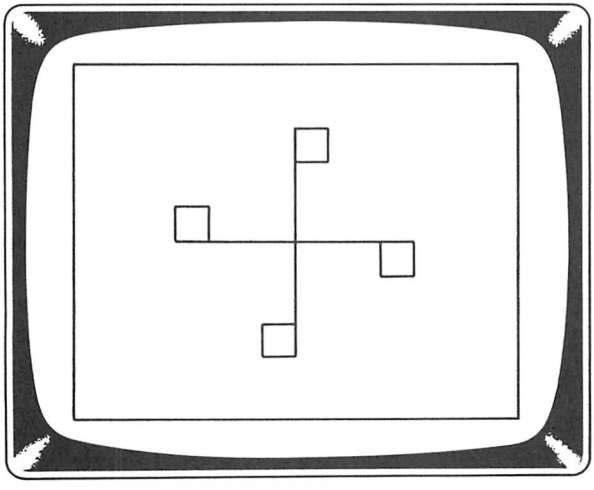


Figure 7.11: Angle command example 2.

```
100 REM ROTATE PROGRAM
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 A$="BM128,96;U70;R20;D20;L20"
150 DRAW "A0"+A$
160 GOSUB 1000
170 DRAW "A1"+A$
180 GOSUB 1000
190 DRAW "A2"+A$
200 GOSUB 1000
210 DRAW "A3"+A$
220 GOSUB 1000
230 GOTO 230
1000 FOR I=0 TO 1000:NEXT I
1010 RETURN
```

The subroutine in 1000 causes a slight delay between each rotation. Each DRAW command is a combination of a fixed string A$ defining the figure and an A command in a string. The two strings are concatenated to form one string that is executed by each DRAW command.

The last angle used in the A command is in force until the next A command is executed. An A0 will reset the display to its normal 0 degree orientation.

STRINGS AND SUBSTRINGS

Before we discuss the next command, the X command, let's review what we know about strings and their use. A string is nothing more than a collection of bytes. Many times these bytes represent text characters in ASCII format as in the string "THIS IS A STRING". In other cases, the string may represent non-ASCII codes by using the CHR$ command, as in A$=CHR$(23)+CHR$(25), which sets string A$ equal to two bytes made up of 23 and 25. We won't be using the second format of strings in the DRAW commands, where all of the data will be character data representing the DRAW commands found in Table 7.1.

A string defined by a statement such as

```
100 A$="THIS IS A STRING"
```

is a constant string. Any time A$ is referenced, it will contain the same data.

Strings can be concatenated, which is a fancy term for appending one string to the end of another string. When this is done, a new string is created. This new string may become a constant string or may simply be temporarily stored. C$ below is a constant string of "THESE ARE THE PROGRAMS THAT TRY MEN'S SOULS", while the temporary string of "FIFTY FOUR FORTY OR FIGHT" disappears after it is displayed.

```
100 A$="THESE ARE THE PROGRAMS "
110 B$="THAT TRY MEN'S SOULS"
120 C$=A$+B$
130 D$="FIFTY-FOUR FORTY "
140 E$="OR FIGHT"
150 PRINT C$, D$+E$
```

A *substring* in reference to the DRAW commands (a DRAW string?) means any constant string that has been defined previously and can be denoted by a string label, such as A$, B$, or Z1$. A substring for DRAW may move to a new position, draw a series of lines, or change the color, just as we have seen in the above examples.

A substring may be called from within a DRAW string by means of the X command, execute substring. This means that one string can reference another string which can reference another string which can reference another string, and so on. When these strings define graphics figures, a whole series of modules may be made up to perform various graphics functions. We'll show you what we mean in the next program.

The program below draws a modest skyscraper, shown in Figure 7.12. It uses several substrings to perform the graphics. String A$ defines a "window draw" of right 8, down 20, left 8, and up 20, as shown in the figure. String B$ executes the window draw by "XA$" and then moves the cursor over to the right 14 positions in preparation for the next window. String C$ executes string B$ by "XB$" three times, so it draws three windows and then moves over left 24 positions and down 24 positions in preparation for the next floor. String D$ executes string C$ to draw a floor of windows five times to draw the entire five floors of windows.

```
100 REM DRAW THE TANDY TOWER
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 A$="R8; D20; L8; U20"
150 B$="XA$; BR14"
160 C$="XB$; XB$; XB$; BL42; BD24"
```

```
170 D$="XC$; XC$; XC$; XC$; XC$;"
180 DRAW "BM94,36; R50; D124; L50; U124"
190 DRAW "BM100,40; XD$;"
200 DRAW "BM94,36; BR25; U60; F9; G9;"
210 GOTO 210
```

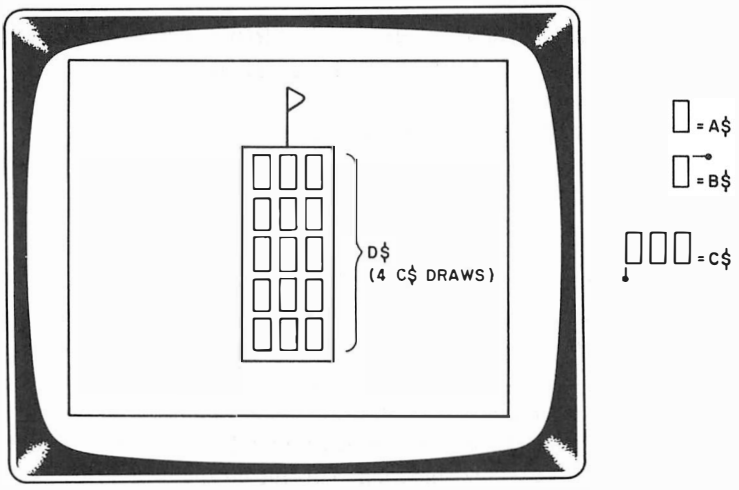


Figure 7.12: Using strings and substrings.

The actual drawing starts with line 170, which draws the building outline. Line 180 moves the cursor to 100,40 and then executes string XD$ to draw all of the windows. Line 190 adds a flag to the top of the building.

There were five levels of graphics here! This simple example doesn't do the X command justice. Graphics can be shortened drastically whenever repetitive designs have to be drawn; they can be defined by a substring and executed at any time. Submodules of designs can easily be defined and used as building blocks to draw composite pictures. A very powerful command!

SCALING UP AND DOWN

We're leaving the best commands until last. Scaling is another very powerful feature of the DRAW command. It's implemented by the S option. The format of S is (another complicated one)

S*x*

where *x* is a value from 1 to 62 indicating the scale factor as shown in Figure 7.13.

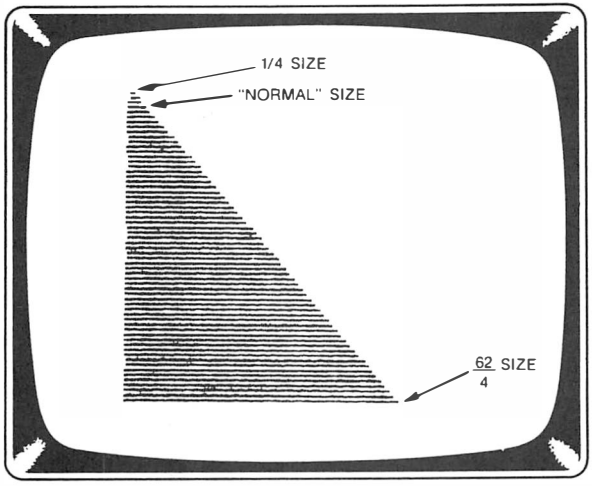


Figure 7.13: Scale factor.

As you can see from the figure, scale factors from 1/4 to 62/4 are possible. Scale factors under 4/4 result in displays that are less than the dimensions as defined in the DRAW; scale factors greater than 4/4 result in enlarged displays.

To see how this works, look at the program below. It generates an oddly shaped figure (see Figure 7.14) vaguely reminiscent of a Space Shuttle in need of maintenance. The figure is defined by the A$ string, which assumes a starting point in the midpoint of the figure. The actual DRAW command is in line 160, which uses a blue color (''C3'') and starts at 128,136. The figure is drawn by executing the B$ string followed by XA$.

```
100 REM ANIMATION BY S COMMAND
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 A$="BL8; R16; D1; L16; U1; BR6; U4;
    R2; U4; D4; R2; D4"
150 FOR S=1 TO 62
160 B$="S"+STR$(S)
170 DRAW "C3; BM128,136; XB$; XA$;"
```

```
180 FOR I=1 TO 99/S: NEXT I
190 DRAW "C1; BM128,136; XB$; XA$;"
200 NEXT S
210 GOTO 210
```

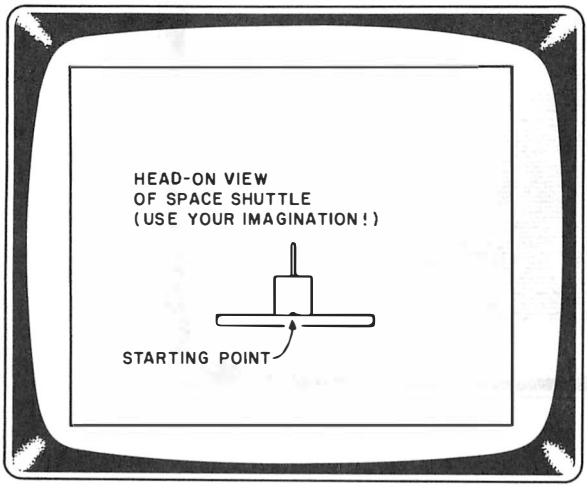


Figure 7.14: Scale factor example.

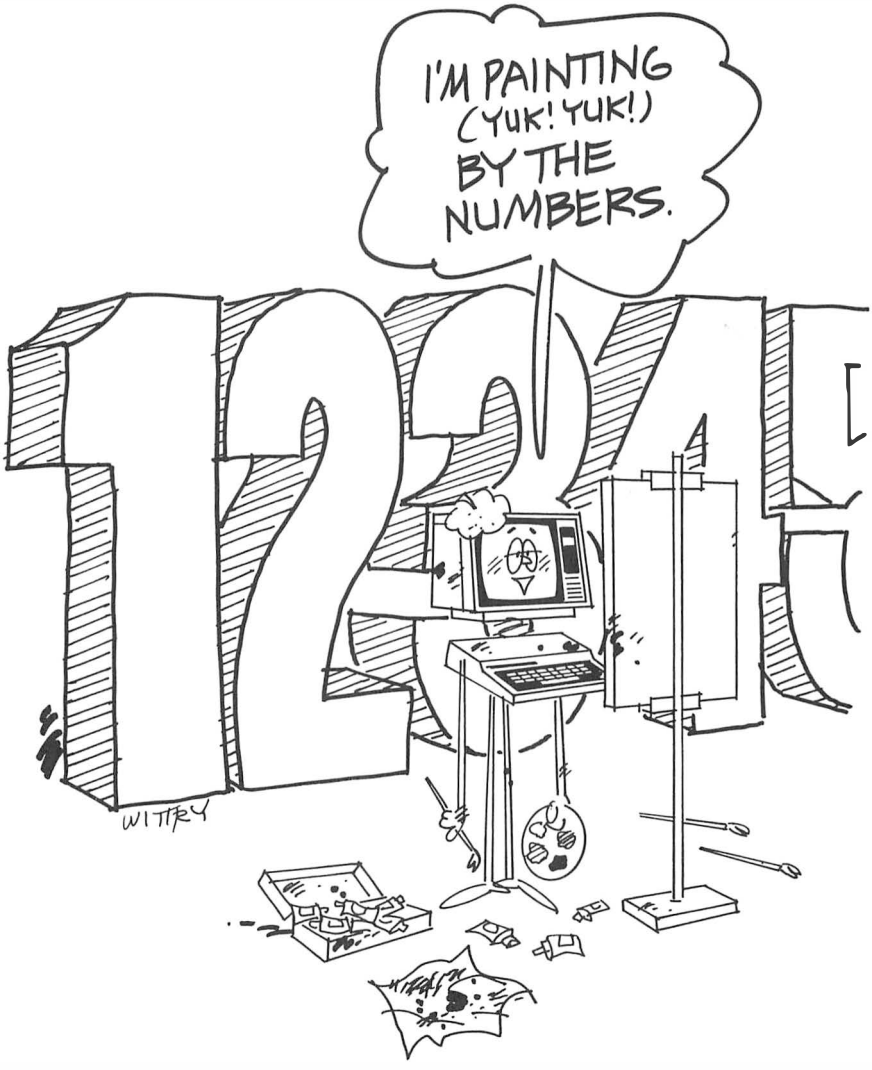
The B$ string is a constant string made up of "S" and the equivalent string value of the numeric variable *S*. The STR$ function is used to convert the numeric variable *S* to a string which is then concatenated with "S" to form "S 1", "S 2", and so forth.

The B$ string sets the scale factor which ranges from 1 to 62 in the loop from line 140 through 190.

After the figure is drawn in statement 160, a short time delay that decreases with larger scale factors is done in line 170. Larger scale factors generate larger figures that take longer times to generate. After the time delay, the figure is erased by a second DRAW line that uses the background color ("C1"). The effect is of a figure getting larger and larger on the screen.

The S command can be used any time that a figure must be reduced or expanded — animation is just one use.

NOTES


I'M PAINTING (YUK! YUK!) BY THE NUMBERS.
WITTEY

CHAPTER 8

Extended Color BASIC: Using the PAINT and GET/PUT Commands

In this chapter we'll discuss the last three Extended Color BASIC graphics commands: PAINT, GET, and PUT.

PAINT is used to fill in areas with a specific color very similar to a paint-by-numbers technique. PAINT will color an area with any color in the current color set (two or four colors) up to a boundary of another specified color.

GET/PUT are two associated commands. They must be used together. GET moves any screen area from the screen memory into an array. After the move, the array holds the graphics data from the screen area for as long as the user desires. The graphics screen may now be used for other displays.

At some later time, the PUT is used to move the array data back onto the screen at any user-specified area. As the areas may be different, GET/PUT can be used for animation and other graphics techniques.

THE PAINT COMMAND

The PAINT command format is

```
PAINT (X,Y),C,B
```

where (X,Y) is a standard graphics coordinate with $X = 0$ through 255 and $Y = 0$ through 191. C is the color to be painted; B is the boundary color for the PAINT.

The operation of PAINT is shown in Figure 8.1, where a figure has been drawn in PMODE 3 with one color. The graphics in this case were generated with a series of DRAW commands, but similar figures could have been generated with LINE, CIRCLE, or other commands. All of these commands, with the exception of LINE BF, produce an outline of a figure, but not a filled-in color. PAINT can be used to fill in any area.

The area to be filled in is first defined by means of the (*X*,*Y*) coordinate. This coordinate can be anywhere within the confines of the area to be PAINTed. The *C* value specifies the color for the PAINT. This

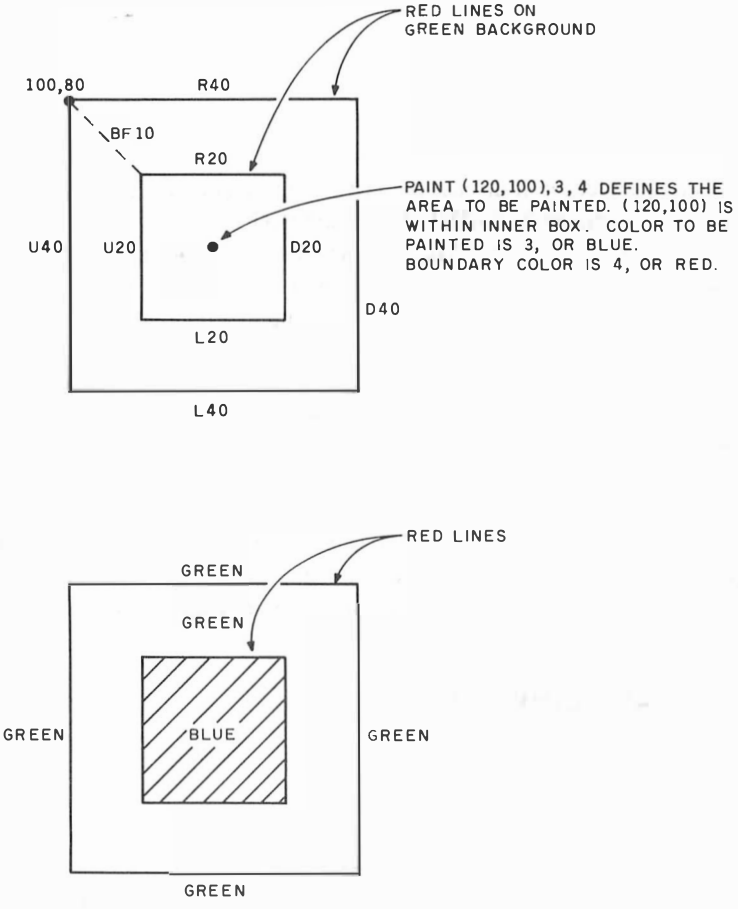


Figure 8.1: PAINT example.

color can be any one in the current color set. In the four-color mode this will be one set of four colors; in the other PMODES, this will be one set of two colors. As with other graphics commands, a color out of the color set can be specified, but the color value will be converted to a color in the current set.

The *B* value is the boundary color. Here again, this must be a color in the current color set. It may be the same as the color value to be PAINTed.

The program below PAINTs one area in the figure.

```
100 REM PAINT EXAMPLE
110 PMODE 3,1
120 SCREEN 1,0
130 PCLS
140 DRAW "BM100,80; R40; D40; L40; U40;
    BF10; R20; D20; L20; U20;
150 PAINT (120,100),3,4
160 GOTO 160
```

In the example above, the foreground color default was red on a green background and the two concentric boxes were drawn with a red color. The boundary color chosen in the PAINT command was also red (4). The color to be painted was blue (3). As the area to be painted was small, the PAINT seemed almost instantaneous.

Suppose that a PAINT was done with a gap in the boundary, as shown in Figure 8.2. Here, the paint leaks out of the boundary, first into the outer box area and then into the area beyond the outer box. Finally, the entire screen is PAINTed! The point here is that the boundary area must be well defined with no gaps or discontinuities or this type of problem will occur. (Best to check the insides of your TV if you see this occurring — video paint can be extremely corrosive.)

PAINT will leak in from outside a boundary also. If the PAINT start point were specified out of both boxes in Figure 8.2, say at 1,1, the result would have been the same.

Another thing about the PAINT of Figure 8.2 is that it takes a much greater time than other Extended BASIC commands due to the filling-in nature of the command. The time required to PAINT an area of one-half the screen is about 4 seconds, a considerable time for any processing operation.

Note also in Figure 8.2 that a PAINT proceeds in segments. In this case the PAINT was done in about six or seven segments, each with an unpredictable direction. (Actually, with a very predictable direction in terms of a computer algorithm, but not very predictable in terms of using the PAINT command.)

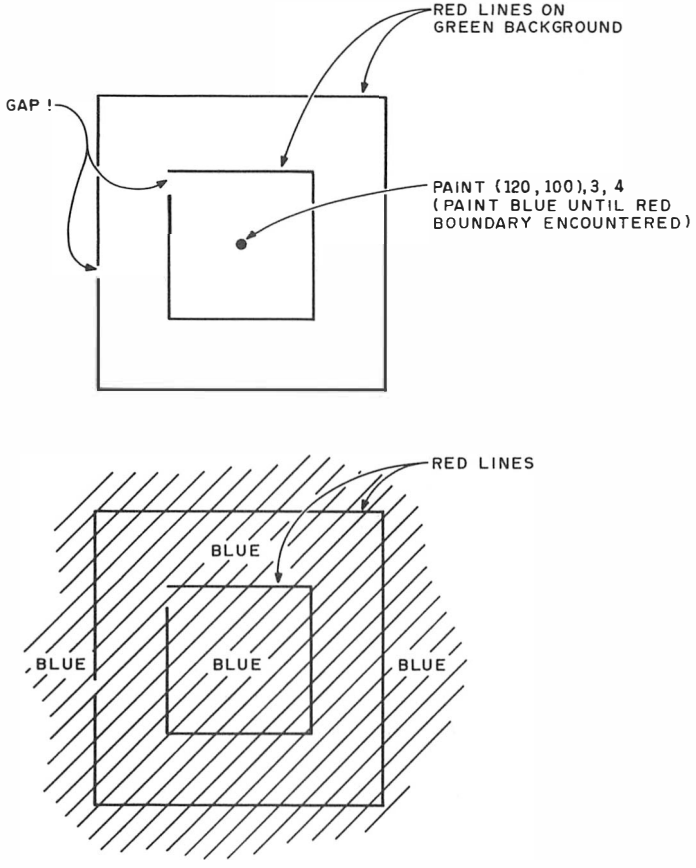


Figure 8.2: Leaking PAINT.

As with all Extended Color BASIC commands, PAINT works only with the colors in the current color set. If other color codes are specified, they are converted to a code in the current set. Specifying an 8 for the PAINT color in the programs above would have had the same result as specifying a 4.

Another condition to watch for in PAINT is that the start point is in the proper area to be PAINTed. Suppose that we have the same figure as Figure 8.1 but that we had specified an inside boundary color of yellow (C2). The PAINT would proceed from the inside of the inner box out and paint over the yellow boundary, as shown in Figure 8.3. Of course, if this is planned, it's fine, but if not, the PAINT will obliterate all areas except the one beyond the designated boundary.

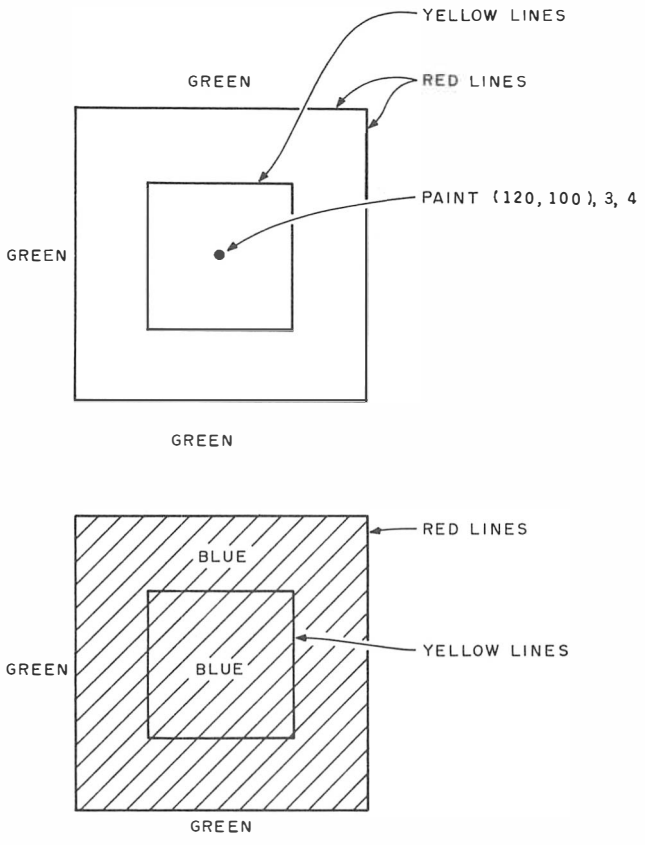


Figure 8.3: PAINT boundaries.

THE GET AND PUT COMMANDS

The last two commands are probably the most powerful (and most complicated) in the Extended Color BASIC set. GET and PUT must be used together. GET defines an area of the graphics screen and stores it in an array while PUT retrieves the data from the array and puts in onto a designated area of the screen.

The GET/PUT action works in the simplest case as shown in Figure 8.4. The screen data in the GET rectangle is stored in array Z. During the execution of the PUT statement, the data is taken from array Z and displayed in the PUT area.

GET and PUT can be used to replicate figures in different places on the screen or to produce animation effects. Before we get into the details

of GET/PUT, let's run a simple example.

The program below replicates (reproduces) the upper left-hand figure at three other screen areas by one GET and three PUT statements, as shown in Figure 8.5.

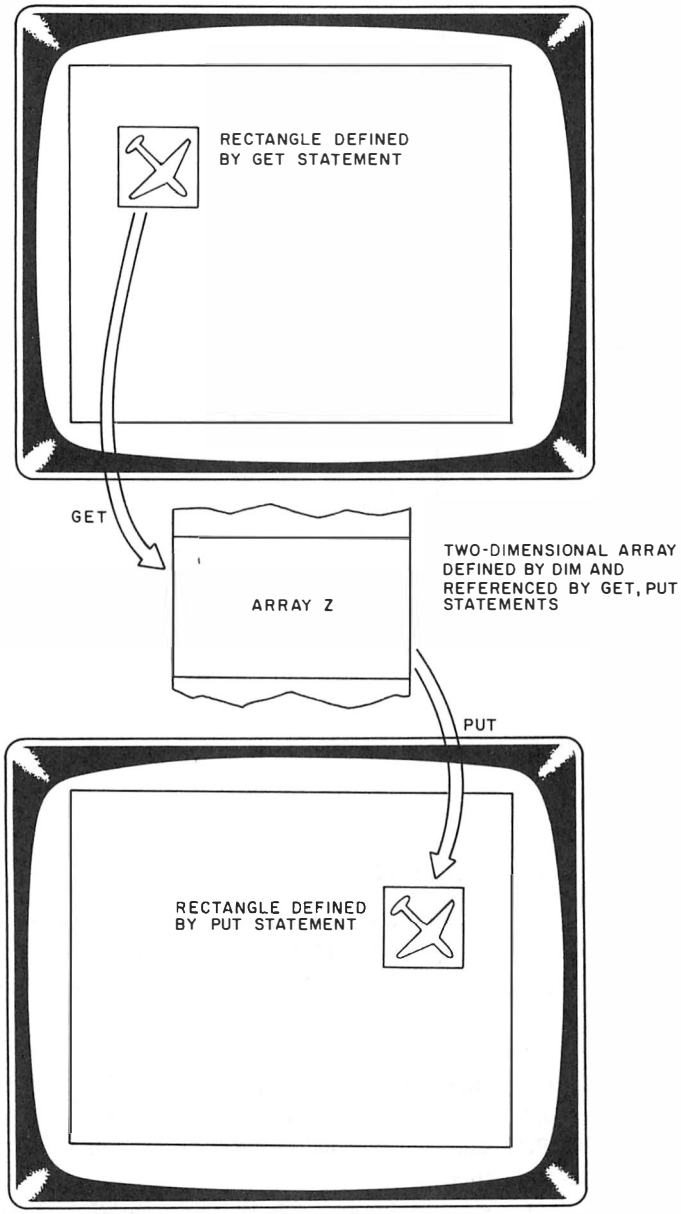


Figure 8.4: GET/PUT action.

```
100 REM SIMPLE USE OF GET/PUT
110 DIM A(19,39)
120 PMODE 3,1
130 SCREEN 1,0
140 PCLS
150 DRAW "BM64,40; E10; F10; D29; L20;
    U29"
160 GET (64,30)-(83,69),A
170 PUT (196,30)-(215,69),A
180 PUT (64,122)-(83,161),A
190 PUT (196,122)-(215,161),A
200 GOTO 200
```

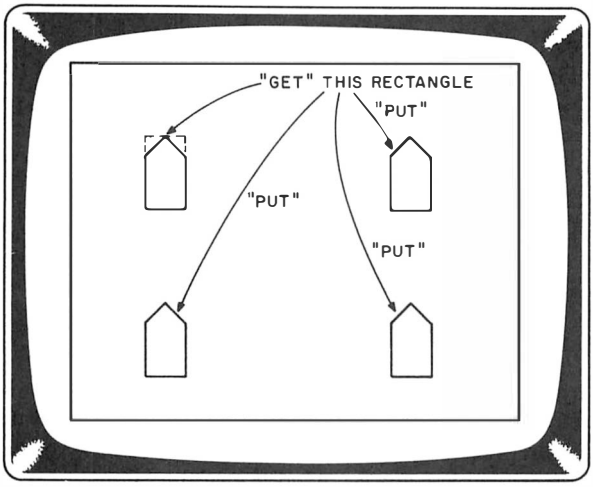


Figure 8.5: GET/PUT example.

The GET command defines a rectangle in the same fashion as LINE. The upper left-hand corner is (64,30) and the lower right-hand corner is (83,69). The GET command stores the contents of the rectangle in two-dimensional array A, previously defined by the DIM A(19,39) array. The three PUTs read the array into the three rectangles defined.

Arrays

To understand the GET/PUT statements, we need to know a little more about arrays. Arrays are structures to hold data. Data within the array is referenced by an element number within the array. A one-dimensional array holds each piece of data one after another in list

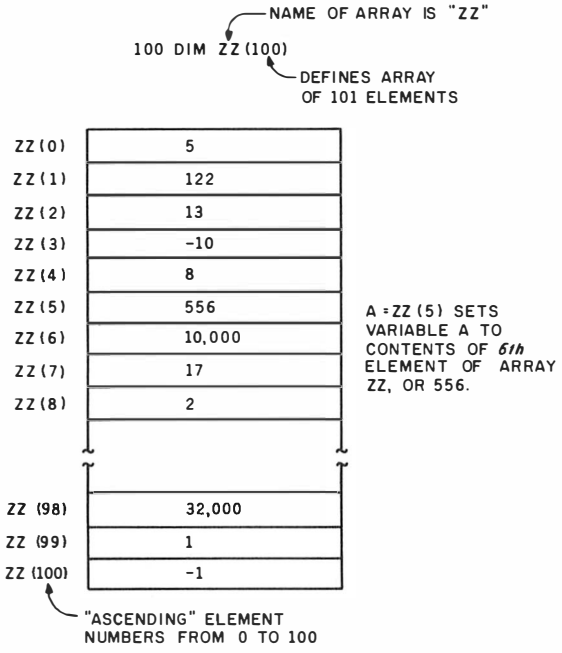


Figure 8.6: One-dimensional array.

form. An element within the array is referenced by one element number, as shown in Figure 8.6.

A two-dimensional array is arranged in two directions, as in a chess board. Elements within the array are referenced by two element numbers, as shown in Figure 8.7. Arrays of three or more dimensions are possible.

Arrays are set up by a DIMension statement. The DIM statement simply allocates storage for the array — it sets aside a block of RAM memory locations based upon the dimensions of the array.

Nothing is put into the elements of the array after the execution of the DIM statement — the BASIC interpreter simply knows where to find the array when references are made to it. The form of the DIM statement for a two-dimensional array is

```
DIM X(N,M)
```

where *N* is one less than the "X" dimension of the array, and *M* is one less than the "Y" dimension of the array. "X" is the name of the array,

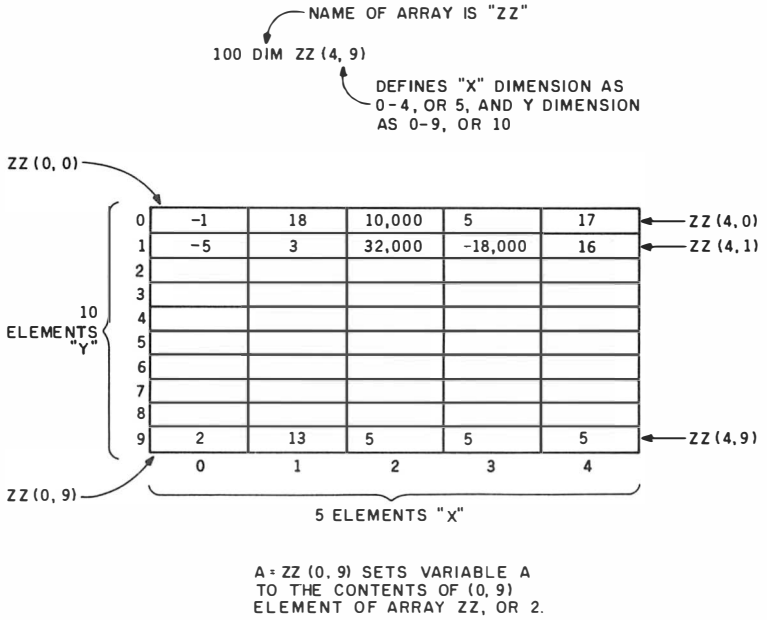


Figure 8.7: Two-dimensional array.

a one- or two-character name identical to a BASIC variable-name format.

An array that represents a graphics screen rectangle of width 15 by height 10 could be set up by

```
DIM AA(14,9)
```

Arrays are referenced by element number or numbers, depending upon the number of dimensions. To store a 55 in the first element of the AA array, we could do

```
AA(0,0)=55
```

Note that the index numbers for the array start with 0 and run through to the last dimension number. In the case above we'd have element numbers from 0 through 14 for X and 0 through 9 for Y. To put numeric values of 0 into the entire array, we'd have

```
100 REM SETUP AND CLEAR ARRAY
110 DIM AA(14,9)
120 FOR X=0 TO 14
130 FOR Y=0 TO 9
140 AA(X,Y)=0
150 NEXT Y
160 NEXT X
```

One of the reasons arrays are used is to be able to represent one-, two-, or three-dimensional arrangements of data that occur in the real-world in computer form. Two-dimensional arrays are almost ideal for processing the *X* and *Y* coordinates of the graphics screen. One element number of the array represents the *X* coordinate, while the other represents the *Y* coordinate.

We say almost ideal because Color Computer BASIC arrays have one fault in regard to storing graphics data. Each element in an array takes 5 bytes of storage space in RAM. An array of 15 by 10 elements, then, has 15 × 10, or 150 total elements, each one occupying 5 bytes. The total number of bytes is 150 × 5 or 750 bytes, all to store 150 graphics elements.

On the other hand, utilizing an array in this fashion can be done rapidly. We're really sacrificing storage for speed, and graphics speed is probably most important.

You can see how much storage is allocated by an array or other variable types by using the MEM command. The MEM command returns the number of free RAM bytes available at any time. After SYSTEM initialization and before any BASIC program is entered, for example, the MEM command can be used like so

```
OK
PRINT MEM
8487
```

The MEM command returned the number of free RAM bytes for a 16K-RAM system with Extended Color BASIC. After adding some BASIC code we get something like the following:

```
OK
100 REM TEST AVAILABLE RAM
110 DIM AA(14,9)
PRINT MEM
 8447
```

The difference between the two values of available RAM is 8487-8447, or 40 bytes. This difference is equal to the storage required for statements 100 and 110. The statements are stored in a compressed form for BASIC commands with most of the text data intact. Some space is required for line numbers and pointers. Note that at this point we haven't *executed* any BASIC statements yet and that the DIM statement for array AA hasn't been processed. At this point the BASIC interpreter is unaware of the array requirements. Now we'll execute the two statements:

```
100 REM TEST AVAILABLE RAM
110 DIM AA(14,9)
RUN
OK
PRINT MEM
 7688
OK
```

What we've done at this point is to execute the REM and DIM statements. The REM statement had no effect on available RAM. The DIM allocated space for array AA. The difference between 8447 and 7688 is the space allocated, plus a few bytes for array variable name and pointers. The difference is 759 bytes, 750 bytes of space for the array proper and 9 bytes for name and pointer.

The absolute maximum RAM space we have available in a 16K system with four graphics pages (default value for PCLEAR) is 8487 bytes. (PCLEAR 1 to 3 will provide additional RAM space.) This represents array storage of 8487/5 or 1699 elements. With the proper array allocation, up to 64,000 points can be stored in an array during GET and PUT.

THE GET STATEMENT

The format of the GET statement is

```
GET (X1,Y1)-(X2,Y2),A,G
```

The first two sets of parameters define a rectangle in a manner similar to the LINE command. The upper left corner is specified by (*X1*,*Y1*) and the lower right corner is specified by (*X2*,*Y2*). The *A* parameter is the name of the array into which the graphics data is to be

stored. This array must have previously been specified with a DIM statement. The *G* parameter specifies full graphic detail and is optional.

When the GET is executed, the graphics data in the display rectangle is moved into the array named in the GET. It remains there forever, unless the array is changed in the program (which should not be done). The graphics data is stored in compressed form. There are either 4, 8, or 16 graphic elements in each byte of the array. We used an array with the same dimensions as the graphic block in an earlier example, but, in fact, we could have cut down on the array requirements considerably.

To make the most efficient use of array storage for GET, allocate an array using a DIM statement of

```
DIM A(Ø,X)
```

where *A* is the array name and *X* is a number based on the number of graphic elements, PMODE, and "G" option. To find *X*,

1. Find the number of elements in the graphic GET statement. If the statement is GET (20,40)–(40,60), for example, the number of graphic elements is 21 × 21 or 442.

2. If you have specified the "G" option, divide this number by 8 if in PMODE 3 or 4, by 16 if in PMODE 1 or 2, or by 32 if in PMODE 0. The result, rounded to the next higher integer, is the number of bytes required. If you aren't using "G", divide by 16 if in PMODE 3 or 4, 8 if in PMODE 1 or 2, or 32 if in PMODE 0. The result, rounded to the next higher integer, is the number of bytes required. If we had been in PMODE 3 with a "G" option for GET (20,40)–(40,60), for example, we'd have 442/8 = 52 6/8. This number rounded to the next higher integer is 53. A total of 53 bytes is required.

3. Now take the number of bytes required and divide by 5. This number, rounded up to the next higher integer, is the *X* to put in the DIM statement. In the example we've been working with we'd have 53/5 = 10 3/5, rounded up is 11. The DIM statement would therefore be DIM A(0,11). That's quite reasonable in terms of storage compared to DIM A(20,20), which would take up 1210 bytes!

Of course, you can leave the DIM statement at the size of the graphics area, but why waste space? The steps in figuring out the value for the DIM statement are shown in Figure 8.8. It's worth the trouble. The array allocated is a two-dimensional array that really looks like a one-dimensional array (say,what?), as the *X* width is 1 (0 + 1).

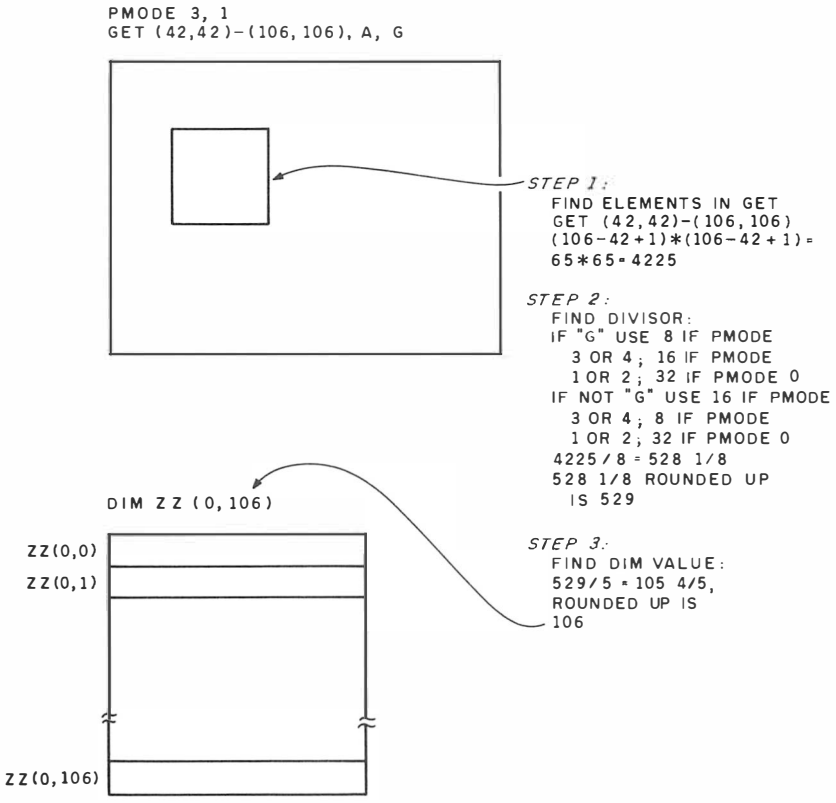


Figure 8.8: Calculation of minimum array for GET/PUT.

THE PUT STATEMENT

The PUT statement format is somewhat similar to the GET statement

```
PUT (X1,Y1)-(X2,Y2),A,ACTION
```

where *X1*,*Y1* is the upper left corner of the destination rectangle and *X2*,*Y2* is the lower right corner of the destination rectangle. *A* is the name of the array used in the GET statement. "ACTION" is an optional action to be taken if the "G" option has been used in the GET statement. We'll discuss the ACTION operations in a moment.

The PUT accesses the array used in the GET (there may be more than one GET, each with a different array) and stores the graphics data

found there in the area specified by the PUT. There must be enough room in the PUT area to store all of the elements from the GET. In general, the PUT should specify the same size rectangle as GET, although it's possible to play some games here and get partial data.

ACTION Items

If you've used a "G" option in the GET, then you must use one of the ACTION items in the PUT; otherwise you may have garbage on the screen.

The PSET (sound familiar?) action items transfers the rectangle defined in the GET with all colors and points set in the same way. An example is shown in Figure 8.9; the program below duplicates the block in the figure.

```
100 REM PSET ACTION ITEM
110 DIM AA(0,66)                    '51*51/8/5
120 PMODE 4,1
130 SCREEN 1,1
140 PCLS
150 LINE (0,0)-(40,40),PSET,BF
160 LINE (0,0)-(10,10),PRESET,BF
170 GET (0,0)-(50,50),AA,G
180 PUT (205,141)-(255,191),AA,PSET
190 GOTO 190
```

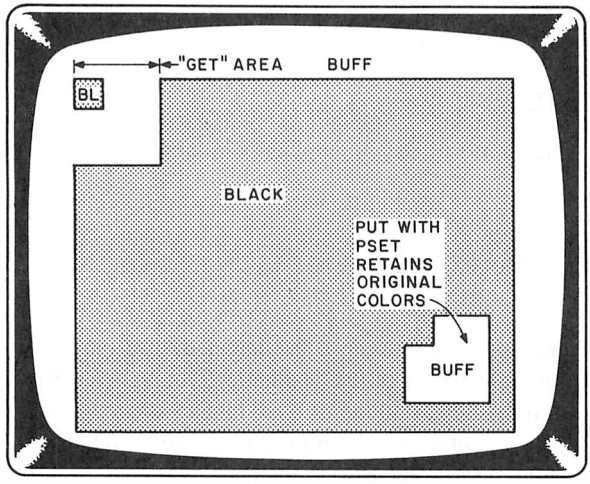


Figure 8.9: PSET with GET/PUT.

The PRESET option transfer the data but resets the points that were set in the GET statement area. Adding statement 165 PUT (205,141) – (255,191),PRESET resets the PUT area directly after the PSET, as shown in Figure 8.10. The PRESET can be used to erase areas on the screen.

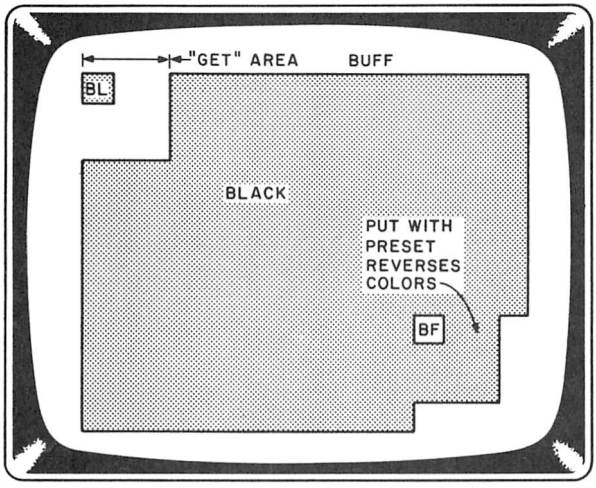


Figure 8.10: PRESET with GET/PUT.

The AND option performs an AND operation on the points of the original rectangle and the points in the destination area. If both the original and destination points are set, the result will be set; if either point is reset, the result will be reset.

The AND can be used to mask out certain areas of graphics.

The OR option performs an OR operation on the points of the original rectangle and the points in the destination area. If one or the other of the points is set, the result will be set. If neither is set the result will be reset.

The OR can be used to guarantee that certain areas of graphics are set to the foreground color.

The NOT option reverses the state of each point in the destination rectangle; the contents of the array do not affect the operation but only define the area of the destination rectangle to be processed.

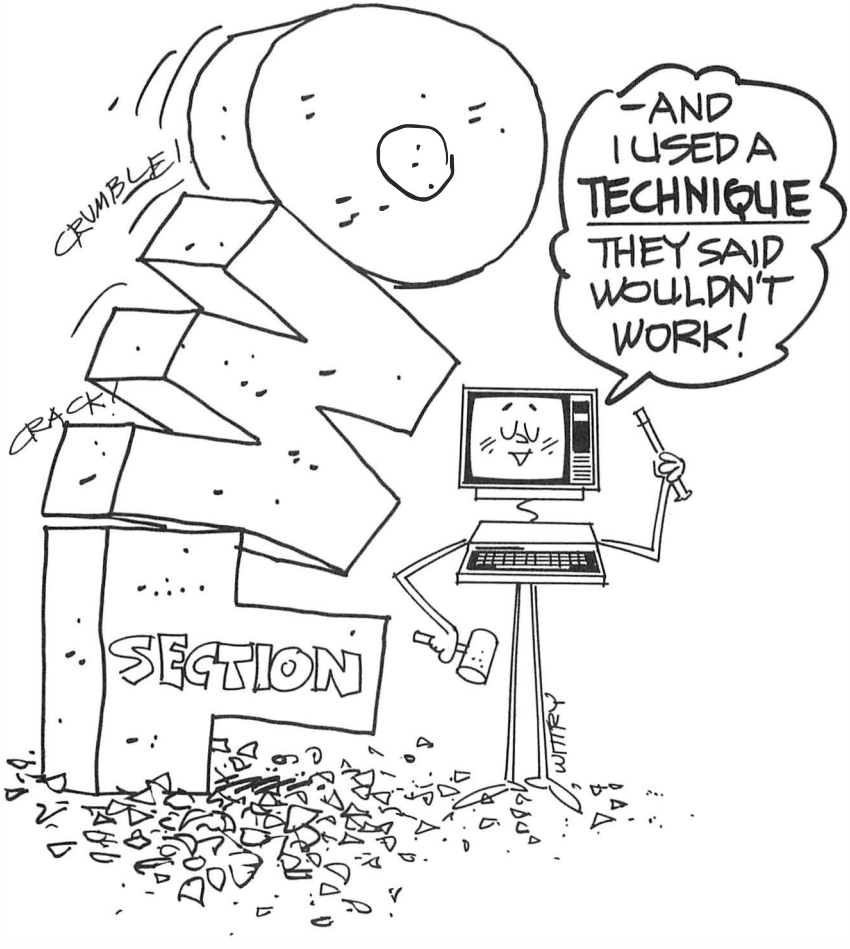
Using the GET/PUT Commands

Any number of GET commands, up to the limits of memory, may be used by defining new arrays to hold the GET data. PUT commands

may reference any of the arrays in any area of the screen desired, provided that the destination areas do not go across screen boundaries. The PMODEs for the GETs and PUTs must correspond, otherwise garbage may appear on the screen.

NOTES

NOTES


CRUMBLE!
CRACK
SECTION
-AND I USED A TECHNIQUE THEY SAID WOULDN'T WORK!
WITTEY

SECTION II

Programming Techniques for Color Computer Graphics

The material in this section is designed as an applications workbook. It is a compendium of graphics techniques for the Color Computer and includes applications techniques for both Color Computer BASIC and Color Computer Extended BASIC. It includes short notes on such things as drawing lines, squares, rectangles, triangles, filled-in figures, gray shades, circles, ellipses, arcs, animation, and others.

The code in the applications should be used as a guide for your own applications. It is not always "idiot-proof" and may not work with invalid arguments. In many cases the speed of the code may be increased by combining statements in a single BASIC line; single statements lines are used here for clarity.

The applications are presented in alphabetical order for easy reference. The application notes are meant to be used as a quick guide on how to implement a graphics problem, but they are not a replacement for the more comprehensive material in earlier sections.

List of Applications

1

Alphanumeric Characters in Graphics Modes

Description

Although Extended Color BASIC graphics modes do not permit internally generated text characters, it is relatively easy to define and generate character sets. One popular way of defining alphabetic, numeric, and special characters is by a 5-by-7 dot matrix, shown in Figure 9.ALP.1. Characters are defined by dots of the matrix.

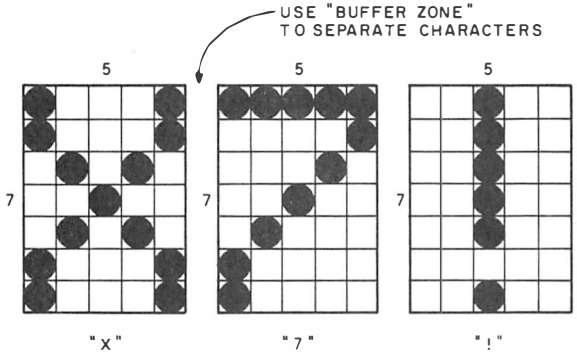


Figure 9.ALP.1: Five-by-seven characters.

This concept can be used with the DRAW command to create text fonts that may be displayed along with graphics patterns in Extended Color BASIC graphics modes. Any number of character sets may be created, with any patterns, such as Japanese Kata-Kana characters or other foreign or special characters.

When 5-by-7 dot-matrix characters are used, up to 42 characters may be displayed horizontally (5 dots plus a buffer of one "off" dot) and 24 characters vertically. Larger matrices, such as 6 by 8, will allow a reasonable number of characters with better resolution.

When the characters are defined properly by a DRAW string, they may be executed at any spot on the screen and may be increased or diminished in size at will.

Sequence

Draw the character to be represented as a dot matrix. Now define the character as a series of DRAW commands. Assume some constant starting position, such as the upper left-hand corner position. Use the smallest number of dots that will be displayed (e.g., 5-by-7 dots).The DRAW string can now be executed by a DRAW command.

Example

Define and draw an "X" using a 5-by-7 dot matrix. The "X" is shown in Figure 9.ALP.1. It is defined by

```
X$="D1;F4;D1;BL4;U1;E4;U1;"
```

The X$ string can now be used in a DRAW command, together with such commands as Execute, Color, and Scale.

```
100 PMODE 4,1
110 SCREEN 1,0
120 X$="D1;F4;D1;BL4;U1;E4;U1;"
130 PCLS 1
140 DRAW "C0;S4;"+X$
150 GOTO 150
```

Notes

1. Use PMODE 4 unless you are working in large scale characters.
2. Black on green is a good foreground/background color. Other colors cause convergence problems and characters show up in multiple colors.

2

Arc, Drawing

Description

This application draws an arc between any two points by using the CIRCLE command of Extended Color BASIC. The arc drawn will represent a hemisphere between the two points as shown in Figure 9.ARC.1. The arc can be specified "concave up/concave left" or "concave down/concave right".

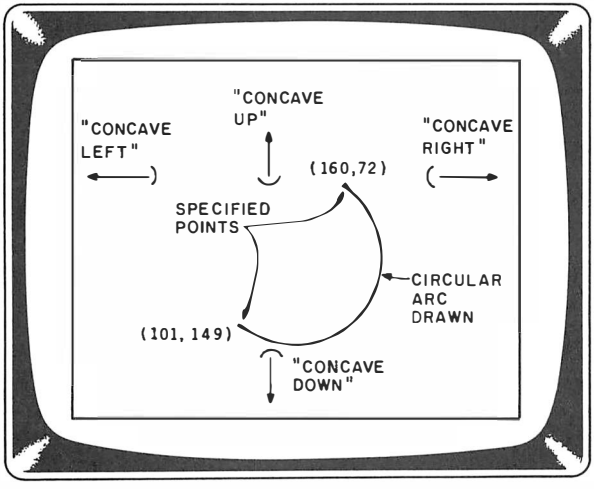


Figure 9.ARC.1: ARC action.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Set *C* to the color to be used. Set *X1,Y1* and *X2,Y2* to the coordinates of the two points. Set *D* to 0 for concave up/left or to 1 for concave down/right.

Use these commands:

```
100 XN=ABS(X2-X1)/2 : YN=ABS(Y2-Y1)/2
110 IF X2-X1<0 THEN XM=XN+X2 ELSE XM=XN+X1
```

```
120 IF Y2-Y1<0 THEN YM=YN+Y2 ELSE YM=YN+Y1
130 XX=0 : IF X2=X1 THEN XX=90 : GOTO 150
140 XX=ATN((Y1-Y2)/(X2-X1))*57.2958
150 XX=1-XX/360
160 IF XX+.5>1 THEN XY=XX-.5 ELSE XY=XX+.5
170 IF D=0 THEN CIRCLE
    (XM,YM),SQR (XM*XM+YM*YM),C,1,XX,XY ELSE
    CIRCLE (XM,YM),SQR(XM*XM+YM*YM),C,1,XY,
    XX
180 GOTO 180
```

Example

Draw an arc between the points 50,100 and 100,50.

```
90 PMODE 3,1
92 SCREEN 1,0
94 PCLS
96 X1=50 : Y1=100 : X2=100 : Y2=50
100 XN=ABS(X2-X1)/2 : YN=ABS(Y2-Y1)/2
110 IF X2-X1<0 THEN XM=XN+X2 ELSE XM=XN+X1
120 IF Y2-Y1<0 THEN YM=YN+Y2 ELSE YM=YN+Y1
130 XX=0: IF X2=X1 THEN XX=90 : GOTO 150
140 XX=ATN((Y1-Y2)/(X2-X1))*57.2958
150 XX=1-XX/360
160 IF XX+.5>1 THEN XY=XX-.5 ELSE XY=XX+.5
170 IF D=0 THEN CIRCLE
    (XM,YM),SQR(XM*XM+YM*YM),C,1,XX,XY ELSE
    CIRCLE (XM,YM),SQR (XM*XM+YM*YM),C,1,
    XY,XX
180 GOTO 180
```

Notes

1. An error will occur if the center of the circle falls outside of the screen limits. That is, if a full circle with the given points could not be drawn, then the arc can not be drawn either.

2. The arc will be flattened if the center of the circle defining the arc is within the screen limits, but a portion of the arc is out of the screen limits.

3

Border Outline, Drawing

Description

This application draws a border outline by using the CIRCLE command of Extended Color BASIC. The border is drawn in the foreground color. It is assumed that the background color is green (color set 0) or buff (color set 1) so that the border outline is visible.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. PCLS the screen to a green or buff color by PCLS 1 or PCLS 5.

Use this CIRCLE command:

```
CIRCLE (128,96),200,C
```

The color used is the color specified in *C*.

Example

Draw a border in color set 0 with green background and red outline.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 CIRCLE (128,96),200,4
140 GOTO 140
```

4

Circle, Drawing

Description

This application draws a circle by using the CIRCLE command of Extended Color BASIC. The center of the circle, the radius, and color code are defined in the command. The circle will be a complete circle and not an arc.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Use this CIRCLE command:

```
CIRCLE (X,Y),R,C
```

The circle will be drawn with center at *X*,*Y*, a radius of *R*, and a color of *C*.

Example

Draw a circle of radius 20 at the center of the screen.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 CIRCLE (128,96),20,4
140 GOTO 140
```

5

Ellipse, Drawing

Description

This application draws an ellipse by using the CIRCLE command of Extended Color BASIC. The center of the circle, the radius, and color code are defined in the command. The radius will determine the width of the ellipse; the height will be determined by the height:width ratio.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Use this CIRCLE command:

```
CIRCLE (X,Y),R,C,HW
```

The ellipse will be drawn with center at X,Y, a color of C, a width of $2 \times R$, and a height of $2 \times R \times HW$.

Example

Draw an ellipse of width 100 and height of 50 at the center of the screen.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 CIRCLE (128,96),50,4,.5
140 GOTO 140
```

6

Intermixing Text and Graphics

Description

Alphanumerics and text can be intermixed only while in the normal text display mode of Color BASIC or Extended Color BASIC. The graphics display mode uses a separate graphics page that will not allow internal generation of text characters. The text display mode allows both internally generated text characters and limited graphics (64 elements horizontally by 32 elements vertically).

Because the four graphics elements of a character position are contained in 1 byte in the text display mode, four graphics elements may be defined by a single 8-bit value. This value may be treated as a single character in a character string, or a series of these graphics characters may be joined together in a longer string.

Alternatively, the values defining the four elements of graphics may be POKEd into the text display area from 1024 through 1535.

Sequence

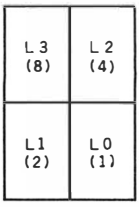
For each four graphics elements in a character position, do the following:

1. Determine the color for the elements. All elements in one character position must be the same color, or black.
2. Compute the value for the character position by

$$V=128+(C-1)*16+L3+L2+L1+L0$$

where C is the color code of 1 through 8 (green through orange) and $L3$, $L2$, $L1$, and $L0$ correspond to the graphics element position as shown in Figure 9.INT.1. If $L3$ is to be on, $L3=8$; if off, 0. If $L2$ is to be on, $L2=4$; if off, 0. If $L1$ is to be on, $L1=2$; if off, 0. If $L0$ is to be on, $L0=1$; if off, 0. If a black color (0) is to be used, all elements are off and $V=128$.

3. POKE the value V into the proper byte of the text display buffer. The address for the POKE will be




```
V = 128 + (C - 1) * 16 + L3 + L2 + L1 + L0

C = 0  BLACK
    1  GREEN
    2  YELLOW
    3  BLUE
    4  RED
    5  BUFF
    6  CYAN
    7  MAGENTA
    8  ORANGE
```

Figure 9.INT.1: Graphics elements coding.

$$1024+L*32+CP,$$

where L is the line number of 0 through 15, and CP is the character position of 0 through 31.

4. Alternatively, set a one-character string equal to CHR$(V). The resulting string can then be PRINTed @ or used in any other way a string is used.

Example

Display "TEXT" bracketed on the left by a bar of blue and on the right by a bar of red, as shown in figure 9.INT.2.

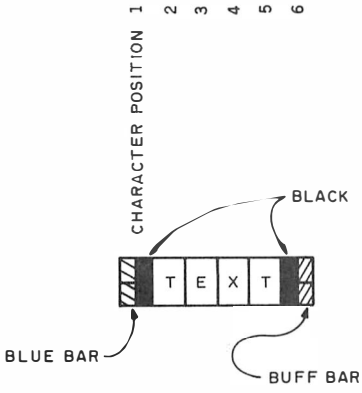


Figure 9.INT.2: Intermix of text and graphics example.

The graphics character position on the left is

```
V1=128+2*16+8+2
```

The graphics character position on the right is

```
V2=128+4*16+4+1
```

The 6 characters can now be PRINTed by

```
100 PRINT CHR$(V1)+"TEXT"+CHR$(V2)
```

or displayed by

```
100 PRINT @ 256+16,"TEXT";
110 POKE 1024+256+15,V1:POKE 1024+
    256+20,V2
120 GOTO 120
```

7

Line, Any Angle, Drawing (Method 1)

Description

In this method of drawing a line using the LINE command of Extended Color BASIC, a start and end point for the line are determined. The *X* and *Y* coordinates of the line are based upon the maximum graphic resolution of 255 horizontal by 192 vertical elements. The two points must be within the screen boundaries.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Use the LINE command in the following format:

```
LINE (X1,Y1)-(X2,Y2),PSET
```

This draws a line between points *X1*,*Y1* and *X2*,*Y2*. The start and end points are included in the line. The color used is the current foreground color.

Example

Draw a line between 24,30 and 128,55.

```
100 PMODE 4,1
110 SCREEN 1,0
120 PCLS
130 LINE (24,30)-(128,55),PSET
140 GOTO 140
```

Notes

1. This method is best for lines which may have any angle. Consider DRAW for angles of 0, 45, 90, 135, 180, 225, 270, and 315 degrees.

8

Lines, Horizontal, Vertical, or Diagonal, Drawing

Description

This method of drawing a line uses the DRAW command of Extended Color BASIC. A starting point, direction, and length for the line are determined. The *X* and *Y* coordinates of the starting point are based upon the maximum graphic resolution of 255 horizontal by 192 vertical elements. The starting point must be within the screen boundaries.

The length of the line is expressed in high-resolution units. Diagonal lines are expressed in the number of units moved in a horizontal or vertical position.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Set *X1*, *Y1* to the starting point for the line. Use the DRAW command in the following format:

```
DRAW "BMX1,Y1; #L"
```

The "#" sign represents the DRAW command as follows: U = up, D = down, R = right, L = left, E = 45 degrees, F = 135 degrees, G = 225 degrees, H = 315 degrees. The L parameter is the length of the line to be drawn. The color used is the current foreground color. Any color in the current PMODE may be used by using the format

```
DRAW "BMX1,Y1; C&; #L"
```

where & is the color code from 1 to 8.

Example

Draw a diagonal red line from 128,92 to 148,72.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 DRAW "BM128,92; C4; E20"
140 GOTO 140
```

Notes

1. This method is best for lines with angles of 0, 45, 90, 135, 180, 225, 270, and 315 degrees.

2. The LINE command may be used instead of DRAW. In LINE, however, the color cannot be specified as in DRAW.

9

Mode of Graphics, Changing

Description

This BASIC program provides an easy way to change the graphics mode to one not implemented in either Color BASIC or Extended Color BASIC. Changing the mode can be used as an initialization to doing graphics work in the video display generator (VDG) modes that are not implemented in the BASIC interpreters.

Sequence

Set variable *P* to the mode to be used. Set variable *C* to the color set, 0 or 1.

For *P*: Values of 0 through 4 correspond to PMODE 0 through 4 (even if Extended BASIC is not installed). Values of –6, –8, –12, and –24 correspond to the semigraphics modes 6, 8, 12, and 24 respectively. Values of 5 through 7 correspond to the 64 × 64 four-color mode, the 128 × 64 two-color mode, and the 128 × 64 four-color mode respectively.

Use these statements:

```
1000 IF P>7 THEN GOTO 1500
     ELSE IF P=-6 THEN GOTO 1100
     ELSE IF P=-8 THEN GOTO 1100
     ELSE IF P=-12 THEN GOTO 1100
     ELSE IF P=-24 THEN GOTO 1100
     ELSE IF P<0 THEN GOTO 1500
1010 IF P>4 THEN T=P+3 ELSE T=P+11
1020 A=(PEEK(65314) AND 7) : POKE
     65314,A+T*16+C*8
1030 IF P=5 THEN P=6
1040 IF P>4 THEN P=P-5 ELSE P=P+3
1050 IF P=7 THEN P=6
1060 IF (P AND 4)=4 THEN POKE &HFFC5,1
     ELSE POKE &HFFC4,0
1070 IF (P AND 2)=2 THEN POKE &HFFC3,1
     ELSE POKE &HFFC2,0
1080 IF (P AND 1)=1 THEN POKE &HFFC1,1
     ELSE POKE &HFFC0,0
```

```
1090 GOTO 1500
1100 IF P=-6 THEN P=0
     ELSE IF P=-8 THEN P=2
     ELSE IF P=-12 THEN P=4
     ELSE IF P=-24 THEN P=6
1110 A=(PEEK(65314) AND 7) :
     IF P=0 THEN POKE 65314,A+16+C*8
     ELSE POKE 65314,A
1120 GOTO 1060
1500 RETURN
```

Example

To set PMODE 4 and color set 1 in a Color BASIC machine, do the following:

```
100 P=4: C=1
110 GOSUB 1000
120 .
130 .
140 GOTO 140
1000 (subroutine above)
```

Notes

The area displayed will be from the current starting address — 1024 if in Color BASIC or Extended Color BASIC text display, or a graphics page start if in Extended Color BASIC graphics display. Change the starting address as described in "Start of Video Memory, Changing" if desired.

10

Moving Figures, by GET,PUT

Description

This application shows how figures can be moved on the screen to produce animation or other effects. The technique used here is by the use of the Extended Color BASIC GET/PUT commands. The figure is drawn at a given starting point by a standard Extended Color BASIC command or commands. The figure is then moved into an array by the GET command and redrawn at a new screen position by the PUT command. The process is repeated as often as required. Indexing (see "Moving Figures by Indexing") can be used to facilitate the PUTs.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Clear the screen by PCLS, or leave the current screen contents in place.

Establish a two-dimensional array for the GET. The array should be large enough to hold the area of the screen to be stored. In the simplest case, the dimensions of the array should match the area dimensions. However, a great deal of memory may be saved and the speed will be increased by using the techniques of Chapter 8.

Draw the figure to be moved by normal commands. Perform a GET on the screen area to move the entire area into the array. Initiate a FOR...TO...SET loop from 0 to an end index value. Draw the figure by performing a PUT to the next screen area, using the index value to determine the location. Continue through the end of the loop.

Example

Move a circle centered at 20,20 with a radius of 3, diagonally down to 170,170.

```
100 PMODE 4,1
110 SCREEN 1,0
120 PCLS
130 DIM V(10,10)
140 CIRCLE (20,20),3
150 GET (15,15)-(25,25),V,G
```

```
160 FOR I=0 TO 150
170 PUT (15+I,15+I)-(25+I,25+I),V,PSET
180 FOR J=0 TO 20: NEXT J
190 NEXT I
200 GOTO 200
```

This example produces a succession of circles, as shown in 9.MFG.1. Each is drawn with a center at 20+I,20+I and then erased. The overhead of the PUT command allows the figure to remain long enough to create an animation effect.

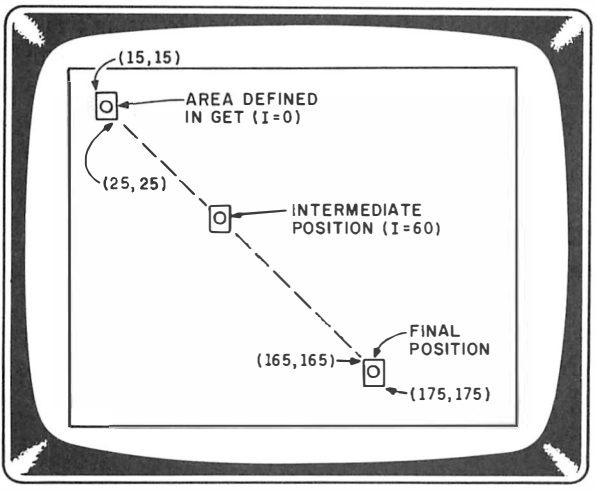


Figure 9.MFG.1: Moving figures by GET/PUT example.

Notes

1. This example moves a circle along the diagonal in about 13 seconds.
2. Delete the FOR J=0... loop for a faster move (6 seconds).
3. The PUTs effect an automatic erase as the array block is rewritten. More elaborate figures may leave garbage unless a proper erase (by a PUT with PRESET or other means) is done.
4. Replicating the block will cut a swath along the path and possibly overwrite other figures that fall into the block area.
5. Use a larger STEP size for slower moves.

11

Moving Figures, by Indexing

Description

This application shows how figures can be moved on the screen to produce animation or other effects. The technique used here is *indexing*. The figure is drawn at a given starting point by a standard Extended Color BASIC command. The figure is then erased by using a PCLS or by overwriting the figure with the background color. The figure is then redrawn at a new screen position and the process is repeated. The process continues until the end of the path is reached.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Clear the screen by PCLS, or leave the current screen contents in place. Initiate a FOR...TO...STEP loop from 0 to an end index value.

Draw the figure at the screen start plus the index value. Erase the figure by a PCLS or by redrawing the figure with the background color. Continue until the last index is reached.

Example

Move a circle centered at 20,20 with a radius of 3, diagonally down to 170,170.

```
100 PMODE4,1
110 SCREEN1,0
120 PCLS
130 FOR I=0 TO 150
140 CIRCLE (20+I,20+I),3
150 CIRCLE (20+I,20+I),3,0
160 NEXT I
170 GOTO 170
```

This example produces a succession of circles as shown in 9.MFI.1. Each is drawn with a center at 20+I,20+I and then erased. The overhead of the CIRCLE command allows the figure to remain long enough to create an animation effect.

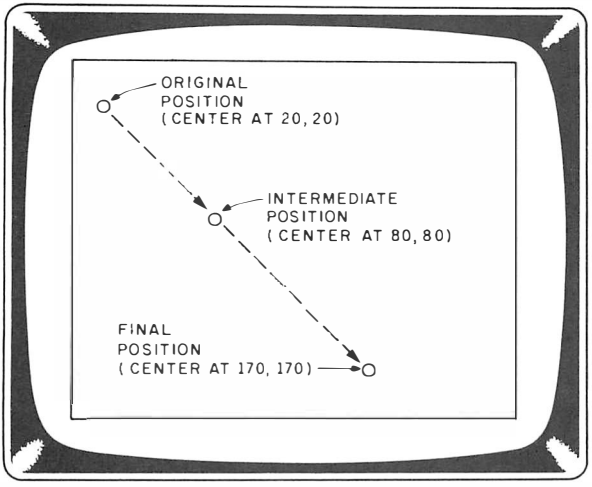


Figure 9.MFI.1: Moving figures by indexing example.

Notes

1. This example moves a circle along the diagonal in about 31 seconds.
2. Use a larger STEP size for a faster move.
3. Substitute a PCLS for the second move if desired. The speed will be about the same.
4. Vary *X* and/or *Y* with the index variable to move vertically, horizontally, or diagonally.
5. Use two index variables for other angles, although this will slow down the move considerably.

12

Octagon, Drawing

Description

This application draws an octagon, as shown in Figure 9.OCT.1. The octagon is drawn with the midpoint at any given *X*,*Y* location. It may be any of 32 sizes — width of 3, 6, 9, 12, 15, etc., up to a base of 96. The entire octagon must be within the screen boundaries or an imperfect octagon will result.

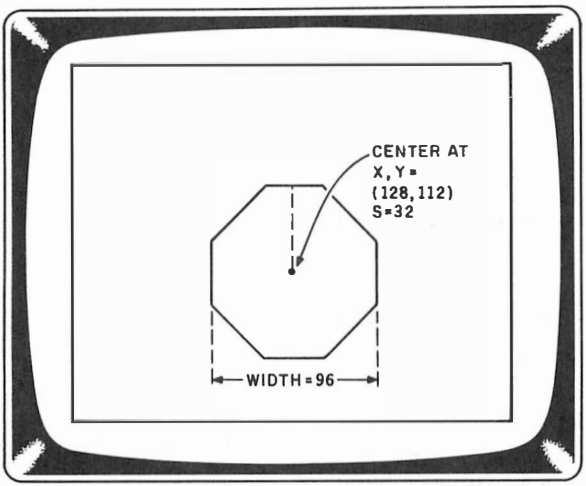


Figure 9.OCT.1: Octagon action.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Set *X*,*Y* to the midpoint of the octagon. Set *S*, scale factor, to a value of 1 through 32. The width of the octagon will be 3 times S.

Use the following BASIC line to draw the octagon:

```
100 DRAW"BM"+STR$(X)+","+STR$(Y)+"S"+
    STR$(S)+";"+"BM+0,-6;R2;F4;D4;G4;L4;
    H4;U4;E4;R2;BM+0,+6"
```

Example

Draw an octagon centered at 128,96 with a width of 33.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 X=129 : Y=96 : S=11
140 DRAW"BM"+STR$(X)+","+STR$(Y)+"S"+
    STR$(S)+";"+"BM+0,-6;R2;F4;D4;G4;L4;
    H4;U4;E4;R2;BM+0,+6"
150 GOTO 150
```

Notes

1. The cursor is centered at the midpoint of the octagon after the line has been executed. Further DRAWs will occur from this point.

2. If the octagon will not fit on the screen, the DRAW will still draw a figure, but one or more sides will be the wrong lengths.

13

Other Colors in PMODE 4

Description

This application provides a gray color in addition to green and black in color set 0, or a blue color in addition to buff and black in color set 1. It uses the PSET command to alternate black and green stripes in a rectangular area. The same technique can be used in a non-rectangular area with some modification.

Sequence

Set PMODE to 4 and page number. Set SCREEN to graphics and color set 0 or 1. Reverse the foreground/background colors by COLOR 0,1. PCLS the screen to a green or buff color by PCLS.

Define the rectangle to be colored gray or blue by setting *X1,Y1* to the upper left-hand corner and *X2,Y2* to the lower right-hand corner. Perform the following loop:

```
130 FOR X=X1 TO X2 STEP 2
140 FOR Y=Y1 TO Y2
150 PSET (X,Y)
160 NEXT Y : NEXT X
```

Example

Color the rectangle defined by 20,20 to 100,100 gray.

```
90 PMODE 4,1
100 SCREEN 1,0
110 COLOR 0,1
120 PCLS : X1=20 : Y1=20 : X2=100 : Y2=100
130 FOR X= X1 TO X2 STEP 2
140 FOR Y=Y1 TO Y2
150 PSET (X,Y)
160 NEXT Y : NEXT X
170 GOTO 170
```

Notes

Other STEP sizes in color set 1 produce additional effects, including rainbow striping.

14

Rectangle, Filled-in, Color BASIC

Description

POKE graphics may be used for Color BASIC to draw filled-in rectangles. This method is fast, allows eight colors, but works only with the 64-by-64 resolution of Color BASIC.

The rectangles may be any size in multiples of character positions. Normal graphics in Color BASIC has a resolution of 64 by 32 with each character position defining four graphics elements. The character position is 4 pixels wide by 6 pixels high. The smallest rectangle that can be defined in this mode is one character position (4 by 6, or two X units by two Y units). Other rectangles will be drawn to the next smallest character position. For example, if a 12-wide by 15-high rectangle was called for, the rectangle would be 12/2 = 6 character positions wide by 15/2 = 7 character positions high. The rectangle is specified in Color BASIC graphic coordinates where $X=0$ to 63 and $Y=0$ to 31.

Sequence

Set *C* to the color to be used, 0 through 9. Set (*X1*,*Y1*) to the coordinates of one corner of the rectangle. Set (*X2*,*Y2*) to the coordinates of the opposite corner of the rectangle.

Use these statements to draw the rectangle:

```
100 IF X2<X1 THEN XL=X2 ELSE XL=X1
110 IF X2<X1 THEN XR=X1 ELSE XR=X2
120 IF Y2<Y1 THEN YT=Y2 ELSE YT=Y1
130 IF Y2<Y1 THEN YB=Y1 ELSE YB=Y2
140 XL=INT(XL/2) : XR=INT(XR/2) :
    XD=XR-XL+1
150 YT=INT(YT/2) : YB=INT(YB/2) :
    YD=YB-YT+1
160 IF C=0 THEN L=0 ELSE L=15
170 IF C<>0 THEN C=C-1
180 PK=128+C*16+L
190 FOR I=1 TO YD
200 FOR J=1 TO XD
210 POKE 1024+XL+J-1+YT*32+(I-1)*32,PK
220 NEXT J
230 NEXT I
```

The sequence above first finds the "left" *X*, *XL*, the "right" *X*, *XR*, the "top" *Y*, *YT*, and the "bottom" *Y*, *YB*.

It then finds the coordinates in character positions by dividing the *X* coordinates by 2 and the *Y* coordinates by 2. The *X* and *Y* distances are then computed — *XD* and *YD*.

If the color is black, all elements will be off and variable *L* set to 0; otherwise *L* is set to all on (1111 binary or 15 decimal). The color code for POKEs is then found by subtracting 1 from variable *C*. The proper code to be POKEd for each character position is $128 + C \times 16 + L$.

Next, two loops are performed. The innermost "J" loop draws a row of colors. The outermost "I" loop increments over YD rows. Each POKE stores one character position of the rectangle.

Example

Draw a blue rectangle with an upper left corner at 32,16 (center of screen) and a lower right corner at 48,24.

```
90 C=3 : X1=32 : Y1=16 : X2=48 : Y2=24
100 IF X2<X1 THEN XL=X2 ELSE XL=X1
110 IF X2<X1 THEN XR=X1 ELSE XR=X2
120 IF Y2<Y1 THEN YT=Y2 ELSE YT=Y1
130 IF Y2<Y1 THEN YB=Y1 ELSE YB=Y2
140 XL=INT(XL/2) : XR=INT(XR/2) :
    XD=XR-XL+1
150 YT=INT(YT/2) : YB=INT(YB/2) :
    YD=YB-YT+1
160 IF C=0 THEN L=0 ELSE L=15
170 IF C<>0 THEN C=C-1
180 PK=128+C*16+L
190 FOR I=1 TO YD
200 FOR J=1 TO XD
210 POKE 1024+XL+J-1+YT*32+(I-1)*32,PK
220 NEXT J
230 NEXT I
240 GOTO 240
```

Notes

This code is not "idiot-proof." Using invalid *X* and *Y* coordinates or colors will yield improper results!

15

Rectangle, Filled-in, Drawing (Method 1)

Description

In this method of drawing a filled-in rectangle using the LINE command of Extended BASIC, a diagonal of the rectangle defines the size and position. The *X* and *Y* coordinates of the diagonal are based upon the maximum graphic resolution of 255 horizontal by 192 vertical elements. The two points must be within the screen boundaries.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Use the LINE command in the following format:

```
LINE (X1,Y1)-(X2,Y2),PSET,BF
```

This draws a rectangle outline. The upper left corner of the rectangle is defined by *X1*,*Y1*. The lower right corner of the rectangle is defined by *X2*,*Y2*. The start and end points are included in the outline. The color used is the current foreground color.

Example

Draw a rectangle whose upper left corner is at 1,1 and whose lower right corner is at 127,91.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 LINE (1,1)-(127,91),PSET,BF
140 GOTO 140
```

Notes

The method is faster than creating a filled-in rectangle with DRAW and PAINT. Compare the example with "Drawing a Filled-in Rectangle (Method 2)."

16

Rectangle, Filled-in, Drawing (Method 2)

Description

This method of drawing a rectangle uses the DRAW and PAINT commands of Extended BASIC. In this method, a DRAW string defines the sides of the rectangle and the starting point. This method has some advantages over LINE with the B option. A color other than the foreground color may be specified and the rectangle may be scaled up or rotated. The increments for the DRAW are based upon the maximum graphic resolution of 255 horizontal by 192 vertical elements. The rectangle may not go beyond the boundaries.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Use the DRAW command in the following format:

```
DRAW "BMX1,Y1; B; RL1; DL2; LL1; UL2"
```

In this format *X1* and *Y1* are the starting coordinates of the rectangle where *X1* = 0 to 255 and *Y1* = 0 to 191. *L1* is the width of the rectangle minus 1, (1 – 255) and *L2* is the height of the rectangle minus 1, (1 – 191). *B* is the border color for the rectangle.

PAINT the rectangle by

```
PAINT (X1+1,Y1+1),C,B
```

where *C* is the color for painting and *B* is the border color used in the DRAW command. *B* and *C* may be equal if the rectangle is to be one solid color. If *B* and *C* are not equal, there will be a narrow outline around the rectangle.

Example

Draw a rectangle whose upper left corner is at 1,1 and whose lower right corner is at 127,91. The width is therefore 127 and the height 91. Color the rectangle blue.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 DRAW "BM1,1; C3; R126; D90; L126; U90"
140 PAINT (2,2),3,3
150 GOTO 150
```

Notes

The cursor for the draw is positioned back at 1,1 after the DRAW.

17

Rectangle Outline, Drawing (Method 1)

Description

In this method of drawing a rectangle using the LINE command of Extended BASIC, a diagonal of the rectangle defines the size and position. The *X* and *Y* coordinates of the diagonal are based upon the maximum graphic resolution of 255 horizontal by 192 vertical elements. The two points must be within the screen boundaries.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Use the LINE command in the following format:

```
LINE (X1,Y1)-(X2,Y2),PSET, B
```

This draws a rectangle outline. The upper left corner of the rectangle is defined by *X1*,*Y1*. The lower right corner of the rectangle is defined by *X2*,*Y2*. The start and end points are included in the outline. The color used is the current foreground color.

Example

Draw a rectangle whose upper left corner is at 1,1 and whose lower right corner is at 127,91.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 LINE (1,1)-(127,91),PSET,B
140 GOTO 140
```

Notes

The method is faster than a rectangle drawn with the DRAW command. Compare the example with "Rectangle Outline, Drawing (Method 2)."

18

Rectangle Outline, Drawing (Method 2)

Description

In this method of drawing a rectangle using the DRAW command of Extended BASIC, a DRAW string defines the sides of the rectangle and the starting point. This method has some advantages over LINE with the B option. The color of the sides may be changed easily and the rectangle may be scaled up or rotated. The increments for the DRAW are based upon the maximum graphic resolution of 255 horizontal by 192 vertical elements. The rectangle may not go beyond the boundaries.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Use the DRAW command in the following format:

```
DRAW "BMX1,Y1; R L1; D L2; L L1; U L2"
```

In this format *X1* and *Y1* are the starting coordinates of the rectangle where *X1* = 0 to 255 and *Y1* = 0 to 191. *L1* is the width of the rectangle minus 1 (1 – 255), and *L2* is the height of the rectangle minus 1 (1 – 191). The color used is the current foreground color.

Example

Draw a rectangle whose upper left corner is at 1,1 and whose lower right corner is at 127,91. The width is therefore 127 and the height 91.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 DRAW "BM1,1; R126; D90; L126; U90"
140 GOTO 140
```

Notes

1. The cursor for the draw is positioned back at 1,1 after the DRAW.

2. The C command can be used at any time to change the color of a side. Example: "R126; C2; D90;..."

19

Rotations, of Figures

Description

The best way to rotate figure outlines is with the Angle option of the DRAW command. Angle lets you displace any line drawn by 0, 90, 180, or 270 degrees, in effect a rotation. The format of Angle is

```
"AX"
```

where *X* is 0,1,2, or 3, corresponding to 0 degrees, 90 degrees, 180 degrees, or 270 degrees.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Define your figure by a DRAW string.

Define the angle command as a separate string, such as A$. This string can be concatenated onto the DRAW string for the figure. Draw the figure by performing a DRAW A$ + " (draw string) ," where A$ is the Angle option.

Example

Rotate the *pentomino* (figure made up of five squares) defined by "R4;D4;R4;D4;L4;D4;L8;U4;R4;U8" (see Figure 9.ROT.1) through 0, 90, 180, 270, and 0 degrees.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 B$="R4;D4;R4;D4;L4;D4;L8;U4;R4;U8;"
140 DRAW "A0"+B$
150 GOSUB 1000
160 DRAW "A1"+B$
170 GOSUB 1000
180 DRAW "A2"+B$
190 GOSUB 1000
```

```
200 DRAW "A3"+B$
210 GOSUB 1000
220 DRAW "A0"+B$
230 GOTO 230
1000 FOR I=0 TO 1000 : NEXT I
1010 PCLS
1020 RETURN
```

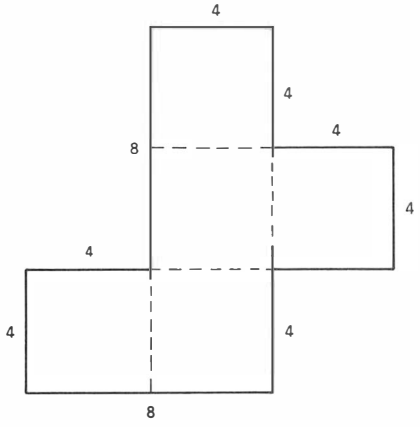


Figure 9.ROT.1: Pentomino.

Notes

1. The default Angle value is 0.
2. Once an Angle option is executed, it stays in force for all subsequent DRAWs.

20

Semigraphics 8, SET/RESET

Description

This application allows a set or reset of one graphic element in the semigraphics 8 mode while in either a Color BASIC or Extended Color BASIC machine. The semigraphics 8 mode is not supported in either BASIC version. It allows 8 colors in a 64-by-64 matrix with a black border.

Sequence

Prior to the set or reset, semigraphics 8 mode must be set by the instructions in "Mode of Graphics, Changing." Also, the start of the display area must have been defined by the procedures in "Start of Video Memory, Changing."

Set variable *X* to the horizontal coordinate, 0 through 63. Set variable *Y* to the vertical coordinate, 0 through 63. Set *C* to the color, 1 through 8. 0 (black) is not a valid color. Set *SR* to 0 if the element is to be set, or to 1 if the element is to be reset. Variable *ST* should be set to the start of the display area (this may be done only once, after setting the start).

Use the following statements:

```
100 IF SR=0 THEN M=128+(C-1)*16 :
    IF (X AND 1)=0 THEN M=M+10 ELSE M=M+5
110 IF SR=1 THEN M=128+(C-1)*16 :
    IF (X AND 1)=0 THEN N=5 ELSE N=10
120 AD=ST+Y*32+INT(X/2)
130 IF SR=0 THEN A=(PEEK(AD) AND 15) :
    A=A OR M : POKE AD,A
140 IF SR=1 THEN A=(PEEK(AD) AND N) :
    A=A OR M : POKE AD,A
```

Example

Set element 63,63 (lower right-hand corner) to blue.

```
90 C=3 : X=63 : Y=63 : SR=0 : ST=14000
100 (statements above)
150 GOTO 150
```

Notes

1. When working in graphics modes that require more buffer than is allocated (more than 512 in Color BASIC), set the display start to high memory and protect this area with a CLEAR.

2. Before processing graphics in this mode, reset all 4096 elements (2048 bytes). This can be done by the statements described here or by 2048 POKEs of 128 + (C-1)*16.

21

Semigraphics 12, SET/RESET

Description

This application allows a set or reset of one graphic element in the semigraphics 12 mode, while in either a Color BASIC or Extended Color BASIC machine. The semigraphics 12 mode is not supported in either BASIC version. The semigraphics 12 mode allows 8 colors in a 64-by-96 matrix with a black border.

Sequence

Prior to the set or reset, semigraphics 12 mode must be set by the instructions in "Mode of Graphics, Changing." Also, the start of the display area must have been defined by the procedures in "Start of Memory, Changing."

Set variable *X* to the horizontal coordinate, 0 through 63. Set variable *Y* to the vertical coordinate, 0 through 95. Set *C* to the color, 1 through 8. Black (0) is not a valid color. Set *SR* to 0 if the element is to be set or to 1 if the element is to be reset. Variable *ST* should be set to the start of the display area (this may be done only once, after setting the start).

Use the following statements:

```
100 IF SR=0 THEN M=128+(C-1)*16 : IF (X
    AND 1)=0 THEN M=M+10 ELSE M=M+5
110 IF SR=1 THEN M=128+(C-1)*16 : IF (X
    AND 1)=0 THEN N=5 ELSE N=10
120 AD=ST+Y*32+INT(X/2)
130 IF SR=0 THEN A=(PEEK(AD) AND 15) :
    A=A OR M : POKE AD,A
140 IF SR=1 THEN A=(PEEK(AD) AND N) : A=A
    OR M : POKE AD,A
```

Example

Set element 63,95 (lower right-hand corner) to red.

```
90 C=4 : X=63 : Y=95 : SR=0: ST=14000
100 (statements above)
150 GOTO 150
```

Notes

1. When working in graphics modes that require more buffer than is allocated (more than 512 in Color BASIC), set the display start to high memory and protect this area with a CLEAR.

2. Before processing graphics in this mode, reset all 6144 elements (3072 bytes). This can be done by the statements described here or by 3072 POKEs of 128 + (C–1)*16.

22

Semigraphics 24, SET/RESET

Description

This application allows a set or reset of one graphic element in the semigraphics 24 mode while in either a Color BASIC or Extended Color BASIC machine. The semigraphics 24 mode is not supported in either BASIC version. The semigraphics 24 mode allows 8 colors in a 64-by-192 matrix with a black border.

Sequence

Prior to the set or reset, semigraphics 24 mode must be set by the instructions in "Mode of Graphics, Changing." Also, the start of the display area must have been defined by the procedures in "Start of Video Memory, Changing."

Set variable *X* to the horizontal coordinate, 0 through 63. Set variable *Y* to the vertical coordinate, 0 through 191. Set *C* to the color, 1 through 8. Black (0) is not a valid color. Set *SR* to 0 if the element is to be set or to 1 if the element is to be reset. Variable *ST* should be set to the start of the display area (this may be done only once, after setting the start).

Use the following statements:

```
100 IF SR=0 THEN M=128+(C-1)*16 : IF (X
    AND 1)=0 THEN M=M+10 ELSE M=M+5
110 IF SR=1 THEN M=128+(C-1)*16 : IF (X
    AND 1)=0 THEN N=5 ELSE N=10
120 AD=ST+Y*32+INT(X/2)
130 IF SR=0 THEN A=(PEEK(AD) AND 15) :
    A=A OR M : POKE AD,A
140 IF SR=1 THEN A=(PEEK(AD) AND N) : A=A
    OR M : POKE AD,A
```

Example

Set element 63,191 (lower right-hand corner) to yellow.

```
90 C=2 : X=63 : Y=191 : SR=0 : ST=14000
100 (statements above)
150 GOTO 150
```

Notes

1. When working in graphics modes that require more buffer than is allocated (more than 512 in Color BASIC), set the display start to high memory and protect this area with a CLEAR.

2. Before processing graphics in this mode, reset all 6144 elements (3072 bytes). This can be done by the statements described here or by 3072 POKEs of 128 + (C–1)*16.

23

Square, Drawing

Description

This application draws a square of any of 32 sizes, as shown in Figure 9.SQU.1. The square is drawn with the midpoint at any given X,Y location. The square may be any of 32 sizes — sides of 4, 8, 12, 16, 20, etc., up to sides of 128. The entire square must be within the screen boundaries or an imperfect square will result.

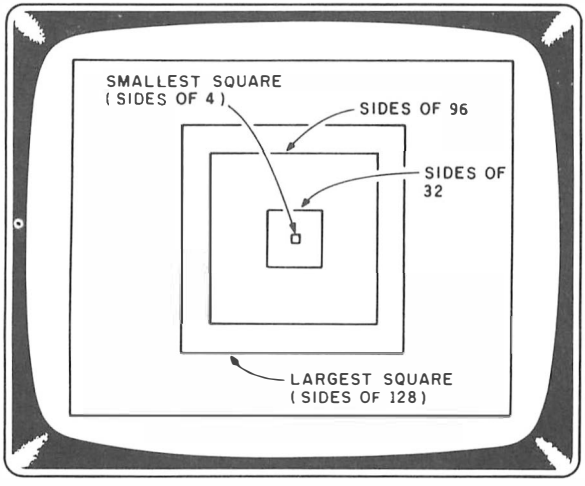


Figure 9.SQU.1: Square action.

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Set *X*,*Y* to the midpoint of the square. Set *S*, scale factor, to a value of 1 through 32. The size of the square's sides will be 4 times *S*.

Use the following BASIC line to draw the square:

```
100 DRAW"BM"+STR$(X)+","+STR$(Y)+
    "S"+STR$(S)+";"+"BM-8,-8;R16;D16;
    L16;U16;BM+8,+8"
```

Example

Draw a square centered at 128,96 with sides of 8.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 X=128 : Y=96 : S=32
140 DRAW"BM"+STR$(X)+","+STR$(Y)+
    "S"+STR$(S)+";"+"BM-8,-8;R16;D16;L16;
    U16;BM+8,+8"
150 GOTO 150
```

Notes

1. The cursor is centered at the midpoint of the square after the line has been executed. Further DRAWs will occur from this point.

2. If the square will not fit on the screen, the DRAW will draw a figure, but one or more sides will be the wrong lengths.

24

Start of Video Memory, Changing

Description

This BASIC program provides an easy method of changing the start of video memory in the hardware to any 512-byte boundary. The program can be used in both Color BASIC and Extended Color BASIC. Changing the start of video memory can be used to graphically display the BASIC interpreter or as initialization to doing graphics work in the VDG modes that are not implemented in the BASIC interpreters.

Sequence

Set variable *S* to the address to be used. The lower 9 bits of this address will be ignored if the address is other than a multiple of 512 — 0, 512, 1024, 1536, etc. The program will set the VDG start to the given address.

Use these statements:

```
1000 S=INT(S/512)
1010 FOR I=&HFFC6 TO &HFFD2 STEP 2
1020 R=S-INT(S/2)*2 : S=INT(S/2)
1030 POKE I+R,0
1040 NEXT I
1050 RETURN
```

Example

To set the VDG to display the area starting at location 2048:

```
100 S=2048
110 GOSUB 1000
120 .
130 .
140 GOTO 140
1000 (subroutine above)
```

Notes

The mode of display will be the one that is currently in force. If in Color BASIC or the text display of Extended Color BASIC, the mode will be alphanumeric and the area on the screen will be the start specified through start + 511. If in an Extended Color BASIC graphics mode, the mode will be the current PMODE and will display the number of bytes normally displayed in that PMODE.

25

Triangle, Drawing

Description

This application draws a triangle with the base down, as shown in Figure 9.TRI.1. The triangle is an isosceles triangle with a right angle at the apex. The triangle is drawn with the midpoint at any given *X*,*Y* location. The triangle may be any of 32 sizes — base of 4, 8, 12, 16, 20, etc., up to a base of 128. The entire triangle must be within the screen boundaries or an imperfect triangle will result.

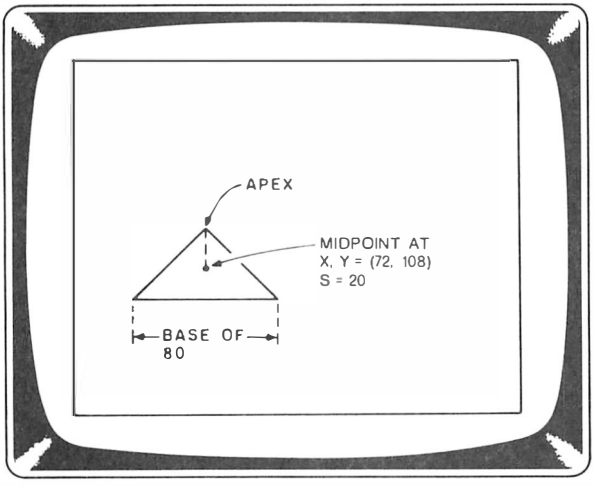


Figure 9.TRI.1: Triangle action.

Sequence

Set PMODE to proper graphics mode and page number. Set SCREEN to graphics and color set. Set *X*,*Y* to the midpoint of the triangle. Set *S*, scale factor, to a value of 1 through 32. The size of the triangle's base will be 4 times *S*.

Use the following BASIC line to draw the triangle:

```
100 DRAW "BM"+STR$(X)+","+STR$(Y)+
    "S"+STR$(S)+";"+"BM+0,-4;F8;L16;E8;
    BM+0,+4"
```

Example

Draw a triangle centered at 129,96 with a base of 32.

```
100 PMODE 3,1
110 SCREEN 1,0
120 PCLS
130 X=129 : Y=96 : S=8
140 DRAW "BM"+STR$(X)+","+STR$(Y)+
    "S"+STR$(S)+";"+"BM+0,-4;F8;L16;E8;
    BM+0,+4"
150 GOTO 150
```

Notes

1. The cursor is centered at the midpoint of the triangle after the line has been executed. Further DRAWs will occur from this point.

2. If the triangle will not fit on the screen, the DRAW will draw a figure, but one or more legs will be the wrong lengths.

Appendices

APPENDICES

Appendix I. Decimal/Binary Conversion Chart

DEC	BINARY
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
10	00001010
11	00001011
12	00001100
13	00001101
14	00001110
15	00001111
16	00010000
17	00010001
18	00010010
19	00010011
20	00010100
21	00010101
22	00010110
23	00010111
24	00011000
25	00011001
26	00011010
27	00011011
28	00011100
29	00011101
30	00011110
31	00011111
32	00100000
33	00100001
34	00100010
35	00100011
36	00100100
37	00100101
38	00100110
39	00100111
40	00101000
41	00101001
42	00101010
43	00101011
44	00101100
45	00101101
46	00101110
47	00101111
48	00110000
49	00110001
50	00110010
51	00110011
52	00110100
53	00110101
54	00110110
55	00110111
56	00111000
57	00111001
58	00111010
59	00111011
60	00111100
61	00111101
62	00111110
63	00111111
64	01000000
65	01000001
66	01000010
67	01000011
68	01000100
69	01000101
70	01000110
71	01000111
72	01001000
73	01001001
74	01001010
75	01001011
76	01001100
77	01001101
78	01001110
79	01001111
80	01010000
81	01010001
82	01010010
83	01010011
84	01010100
85	01010101
86	01010110
87	01010111
88	01011000
89	01011001
90	01011010
91	01011011
92	01011100
93	01011101
94	01011110
95	01011111
96	01100000
97	01100001
98	01100010
99	01100011
100	01100100
101	01100101
102	01100110
103	01100111
104	01101000
105	01101001
106	01101010

107	01101011	151	10010111	195	11000011
108	01101100	152	10011000	196	11000100
109	01101101	153	10011001	197	11000101
110	01101110	154	10011010	198	11000110
111	01101111	155	10011011	199	11000111
112	01110000	156	10011100	200	11001000
113	01110001	157	10011101	201	11001001
114	01110010	158	10011110	202	11001010
115	01110011	159	10011111	203	11001011
116	01110100	160	10100000	204	11001100
117	01110101	161	10100001	205	11001101
118	01110110	162	10100010	206	11001110
119	01110111	163	10100011	207	11001111
120	01111000	164	10100100	208	11010000
121	01111001	165	10100101	209	11010001
122	01111010	166	10100110	210	11010010
123	01111011	167	10100111	211	11010011
124	01111100	168	10101000	212	11010100
125	01111101	169	10101001	213	11010101
126	01111110	170	10101010	214	11010110
127	01111111	171	10101011	215	11010111
128	10000000	172	10101100	216	11011000
129	10000001	173	10101101	217	11011001
130	10000010	174	10101110	218	11011010
131	10000011	175	10101111	219	11011011
132	10000100	176	10110000	220	11011100
133	10000101	177	10110001	221	11011101
134	10000110	178	10110010	222	11011110
135	10000111	179	10110011	223	11011111
136	10001000	180	10110100	224	11100000
137	10001001	181	10110101	225	11100001
138	10001010	182	10110110	226	11100010
139	10001011	183	10110111	227	11100011
140	10001100	184	10111000	228	11100100
141	10001101	185	10111001	229	11100101
142	10001110	186	10111010	230	11100110
143	10001111	187	10111011	231	11100111
144	10010000	188	10111100	232	11101000
145	10010001	189	10111101	233	11101001
146	10010010	190	10111110	234	11101010
147	10010011	191	10111111	235	11101011
148	10010100	192	11000000	236	11101100
149	10010101	193	11000001	237	11101101
150	10010110	194	11000010	238	11101110

239	11101111
240	11110000
241	11110001
242	11110010
243	11110011
244	11110100
245	11110101
246	11110110
247	11110111
248	11111000
249	11111001
250	11111010
251	11111011
252	11111100
253	11111101
254	11111110
255	11111111

Appendix II. Color BASIC Graphics Commands and Actions

CLS (c)

c is a color code of 0-8. Clears all 512 bytes (64 by 32 pixels) of text screen to specified color.

```
100 CLS(4)      'clear screen to red
```

POINT (h,v)

h is the horizontal element number from 0 through 63. *v* is the vertical element number from 0 through 31. Returns an argument for the specified pixel at *h,v*. If the character position containing the pixel is in the alphanumeric (text) mode, returns –1. If the character position containing the pixel is in the graphics mode and the pixel is on, returns the color code (c) of 1 through 8. If the character position is in the graphics mode and the pixel is off (black), returns a 0.

```
100 IF POINT (32,15)=-1 THEN PRINT "NOT
    GRAPHICS!" ELSE PRINT "OK"
```

RESET (h,v)

h is the horizontal element number from 0 through 63. *v* is the vertical element number from 0 through 31. The pixel at *h,v* is turned off. A black color results.

```
100 RESET (0,0): RESET (63,31)
    'reset corner pixels
```

SET (h,v,c)

h is the horizontal element number from 0 through 63. *v* is the vertical element number from 0 through 31. *c* is the color code (see CLS). Sets the pixel at *h,v*. If background color is other than black, changes three adjacent pixels to the specified color. If background is black, only the specified pixel is set to the color.

```
100 SET (63,0) : SET (0,31)
    'set corner pixels
```

Color Codes (c)

0 = black
1 = green
2 = yellow
3 = blue
4 = red
5 = buff
6 = cyan
7 = magenta
8 = orange

Appendix III. Extended Color BASIC Graphics Commands and Actions

CIRCLE (X,Y),R,C,HW,S,E

(*X*,*Y*) are the coordinates of the center of the circle or ellipse or arc. *R* is the radius of the circle or ellipse or arc to be drawn. *C* is the color to be used in drawing the outline and is optional. If not specified, the current foreground color is used. *HW* is the ratio of height to width, from 0 to infinity. If not specified, 1 (circle) is used. *S* is the starting point for an arc and *E* is the ending point for an arc. *S* and *E* may be 0 to 1 and are optional. If not specified, a complete circle or ellipse is drawn.

```
100 CIRCLE (128,96),20
    'draw circle of radius 20 in center

110 CIRCLE (128,96),20,6
    'draw cyan circle

120 CIRCLE (128,96),40,,,.5
    'draw ellipse

130 CIRCLE (128,96),40,,2     'draw ellipse

140 CIRCLE (128,96),40,,,0,,.5
    'draw circular arc

150 CIRCLE (128,96),40,,2,0,,.5
    'draw elliptical arc
```

COLOR F,B

F is the foreground color from 0 to 8. *B* is background color from 0 to 8. Foreground and background colors are set accordingly. Colors must be in the current color set as defined by SCREEN.

```
100 COLOR 1,4
    'set foreground to green, background
     to red
```

DRAW "string"

The *string* is string of DRAW commands. See Chapter 7.

GET (X1,Y1)–(X2,Y2),A,G

(*X1*,*Y1*) is one corner of the screen area to be processed. (*X2*,*Y2*) is the opposite corner of the screen area to be processed. A is the name of the array to be used to store graphics data from the GET. A must be previously defined by a DIM statement. G is optional. When used, complete graphics detail will be stored in the array. GET stores graphics data from the defined screen area into an array for later use by PUT.

```
100 (50,50)-(60,60),ZZ,G
    'get graphics data
```

LINE (X1,Y1)–(X2,Y2),PSET or PRESET,B or BF

(*X1*,*Y1*) is one end point of line on screen. (*X2*,*Y2*) is the opposing end point. If PSET is used, the line, box, or filled-in box will be the current foreground color. If PRESET is used, the line, box, or filled-in box will be the current background color. B or BF are optional. If neither is used, a line will be drawn between *X1*,*Y1* and *X2*,*Y2*. If B is used, a box will be drawn with *X1*,*Y1* and *X2*,*Y2* treated as opposite corners. If BF is used, a filled-in box will be drawn with the color defined by PSET or PRESET.

```
100 LINE (50,50)-(90,80),PSET
    'draw line

110 LINE (50,50)-(90,80),PRESET
    'erase line

120 LINE (50,50)-(90,80),PSET,B
    'draw box outline

130 LINE (50,50)-(90,80),PRESET,B
    'erase box outline

140 LINE (50,50)-(90,80),PSET,BF
    'draw filled-in box
```

PAINT (X,Y),CP,CB

(*X*,*Y*) defines the point at which the color is to be started. *CP* is 0 to 8 and is the color for the PAINT. *CB* is 0 to 8 and is the border color at which the painting is to stop. PAINTing with color proceeds until the border color is encountered. The border must completely surround the area in which PAINTing is to occur.

```
100 PAINT (128,96),7,8
    'paint magenta until orange border
```

PCLEAR N

This reserves *N* graphics pages of 1536 bytes each. *N* is 1 to 8. If this command is never specified, the default number of pages reserved is 4. No data is stored in the pages. (The pages are not cleared.)

```
100 PCLEAR 5      'clear 5 pages
```

PCLS C

This clears the current graphics screen with specified color *C*. *C* is 1 through 8. *C* must be in the current color set.

```
100 PCLS 4      'clear screen with buff
```

PCOPY N TO M

This copies the contents of graphics page *N* to graphics page *M*. *N* and *M* are 1 to 8. The contents of *N* are not affected.

```
100 PCOPY 1 TO 5      'copy page 1 to 5
```

PMODE M,P

M is 0 to 4 and specifies the graphic mode to be selected. *P* is 1 to 8 and specifies the graphics page to be displayed. If PMODE is never specified, mode 0 and page 1 are used.

```
100 PMODE 3,2      'set PMODE 3 and page 2
```

PPOINT (X,Y)

This tests the point at (*X*,*Y*) and returns the color code of the point. The color code will be in the current color set.

```
100 A=PPOINT(129,96)   'test center point
```

PRESET (X,Y)

This resets the point at (*X*,*Y*). The current background color is used in the reset.

```
100 PRESET (128,96)   'reset center point
```

PSET (X,Y)

This sets the point at (*X*,*Y*). The current foreground color is used in the set.

```
100 PSET (128,96)  'set the center point
```

PUT (X1,Y1)–(X2,Y2),A,(options)

(*X1*,*Y1*) defines one corner of a screen area. (*X2*,*Y2*) defines the opposite corner. A defines an array previously used by a GET. See Chapter 8 for options. Graphics data previously stored in array A is PUT into defined screen area. Contents of array A are not affected.

```
100 PUT (80,80)-(90,90),ZZ
    'display graphics data
```

SCREEN T,S

This selects the text or graphics screen and color set. *T* is 0 if text screen to be displayed or 1 if graphics screen is to be displayed. *S* is color set, 0 or 1.

```
100 SCREEN 1,1
    'select graphics and color set 0
```

Notes

1. *X* (or *X1* or *X2*) is 0 to 255.
2. *Y* (or *Y1* or *Y2*) is 0 to 191.
3. *C* is 0 to 8, representing black, green, yellow, blue, red, buff, cyan, magenta, or orange, respectively.

Appendix IV. Color Computer Graphics Modes

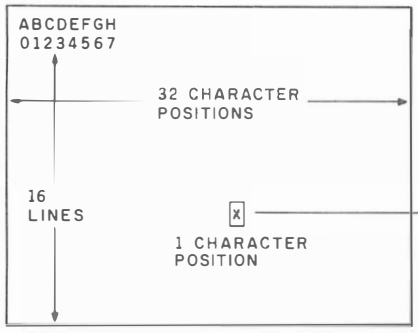
MODE *1*

ALPHANUMERIC

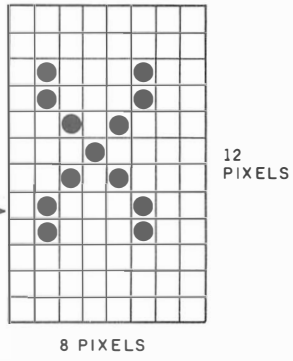
BASIC MODE

NORMAL TEXT

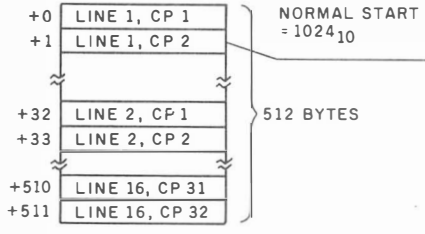
DISPLAY FORMAT



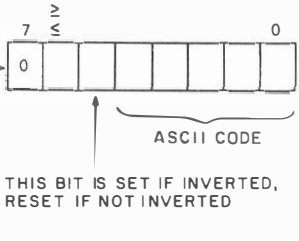
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



COLORS

BORDER BLACK

CHARACTER/BACKGROUND

IF CSS = 0 AND INV = 0, GREEN ON BLACK
IF CSS = 0 AND INV = 1, BLACK ON GREEN
IF CSS = 1 AND INV = 0, ORANGE ON BLACK
IF CSS = 1 AND INV = 1, BLACK ON ORANGE

USE

Color BASIC and Extended Color BASIC both reset to green on black.

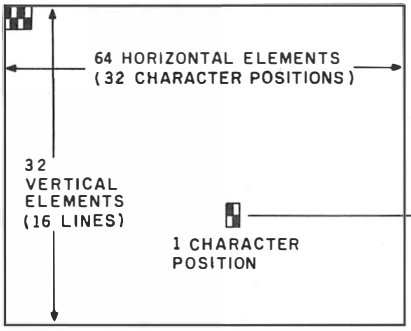
MODE 2

SEMIGRAPHIC 4

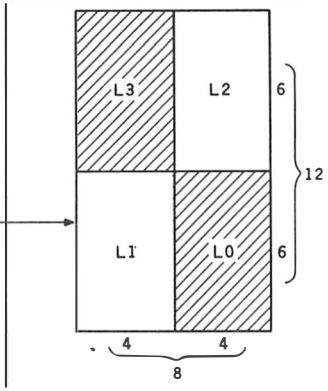
BASIC MODE

USED BY SET/RESET IN COLOR BASIC AND EXTENDED COLOR BASIC.

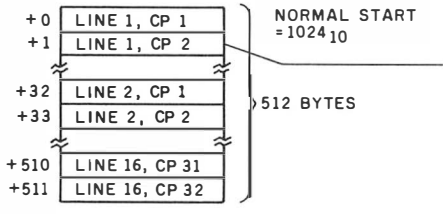
DISPLAY FORMAT



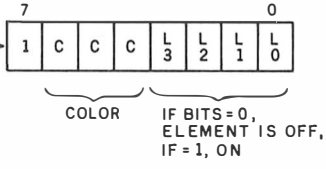
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



COLORS

BORDER BLACK

CHARACTER/BACKGROUND

CCC
000 GREEN
001 YELLOW
010 BLUE
011 RED
100 BUFF
101 CYAN
110 MAGENTA
111 ORANGE

USE

SET/RESET when in text display uses this mode in both Color BASIC and Extended Color BASIC. Alphanumeric mode above and this mode can be used together by storing proper bytes in text buffer. Graphics bytes will have bit 7 set.

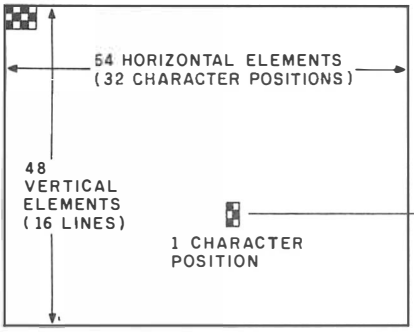
MODE 3

SEMIGRAPHIC 6

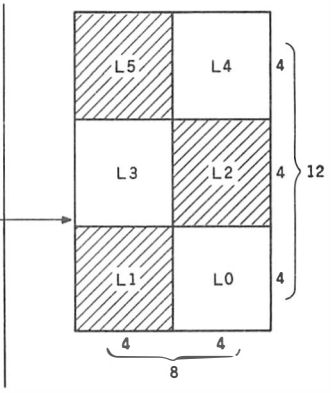
BASIC MODE

NOT IMPLEMENTED. MUST BE SET AS DESCRIBED UNDER "USE."

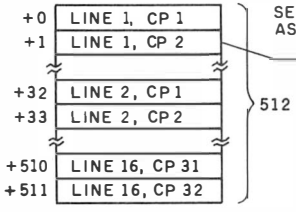
DISPLAY FORMAT



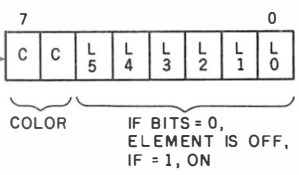
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



COLORS

BORDER BLACK

CHARACTER/BACKGROUND

CC		
00	GREEN	CSS = 0
01	YELLOW	
10	BLUE	
11	RED	
00	BUFF	CSS = 1
01	CYAN	
10	MAGENTA	
11	ORANGE	

USE

1. A = PEEK (65314) : POKE 65314, (A AND 7) + B + X where X = 0 for CSS 0 OR 16 for CSS 1.
2. POKE 65476, 0 = POKE 65474, 0 : POKE 65472, 0.
3. Set START by POKEs to 65,478–65,491. See Chapter 2.
4. Store bytes in 512-byte video display area as required.

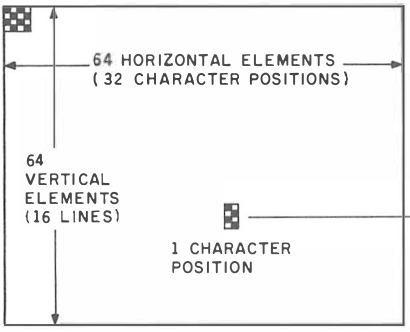
MODE 4

SEMIGRAPHIC 8

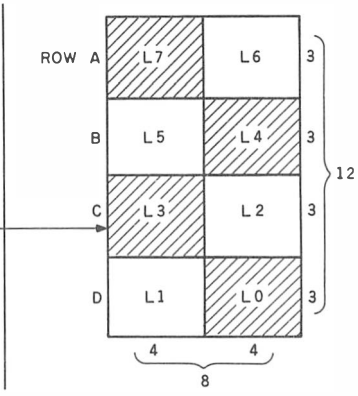
BASIC MODE

NOT IMPLEMENTED. MUST BE SET AS DESCRIBED UNDER "USE."

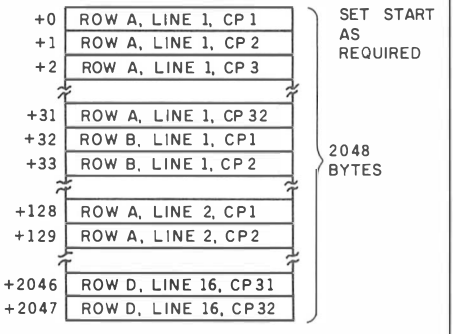
DISPLAY FORMAT



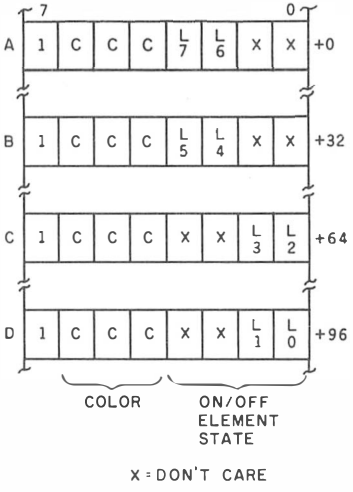
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



Semigraphic 8 continued

COLORS

BORDER BLACK

CHARACTER/BACKGROUND

CCC
000 GREEN
001 YELLOW
010 BLUE
011 RED
100 BUFF
101 CYAN
110 MAGENTA
111 ORANGE

USE

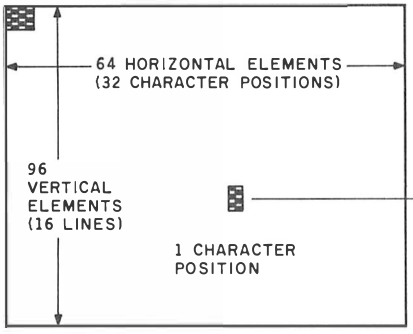
1. A = PEEK (65314) : POKE 65314, (A AND 7).
2. POKE 65475, 0 : POKE 65475, 1 : POKE 65472, 0.
3. Set START by POKEs to 65,478–65,491. See Chapter 2.
4. Store bytes in 2048-byte video display area as required.

MODE 5
SEMIGRAPHIC 12

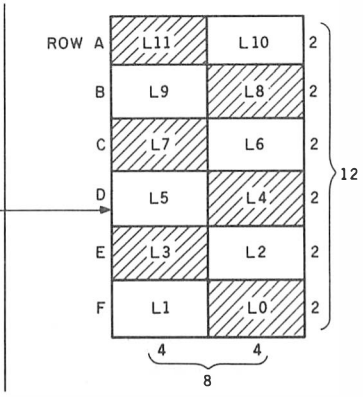
BASIC MODE

NOT IMPLEMENTED. MUST BE SET AS DESCRIBED UNDER "USE."

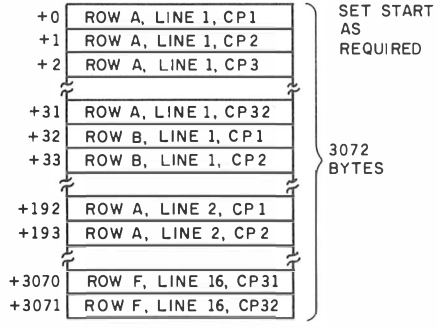
DISPLAY FORMAT



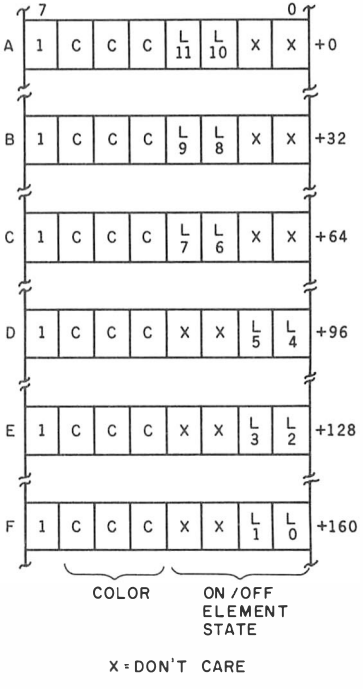
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



Semigraphic 12 continued

COLORS

BORDER BLACK

CHARACTER/BACKGROUND

CCC
000 GREEN
001 YELLOW
010 BLUE
011 RED
100 BUFF
101 CYAN
110 MAGENTA
111 ORANGE

USE

1. A = PEEK (65314) : POKE 65314, (A AND 7).
2. POKE 65477, 1 : POKE 65474, 0 : POKE 65472, 0.
3. Set START by POKEs to 65,478-65,491. See Chapter 2.
4. Store bytes in 3072-byte video display area as required.

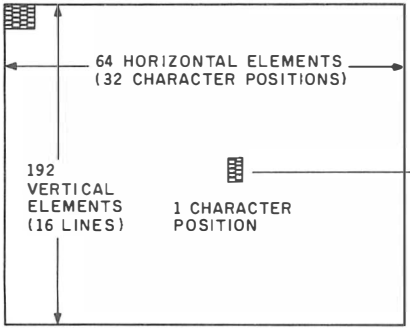
MODE 6

BASIC MODE

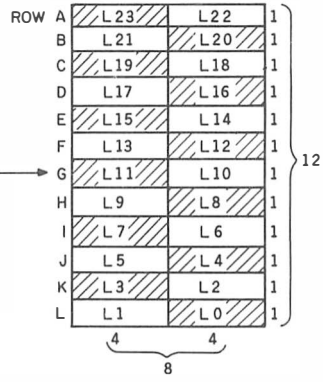
SEMIGRAPHIC 24

NOT IMPLEMENTED. MUST BE SET AS DESCRIBED UNDER "USE."

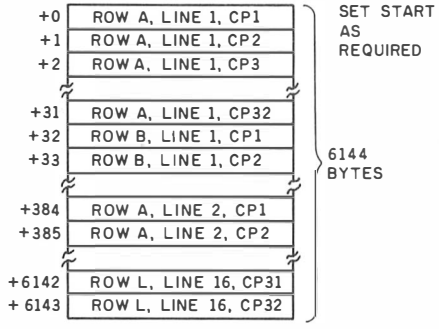
DISPLAY FORMAT



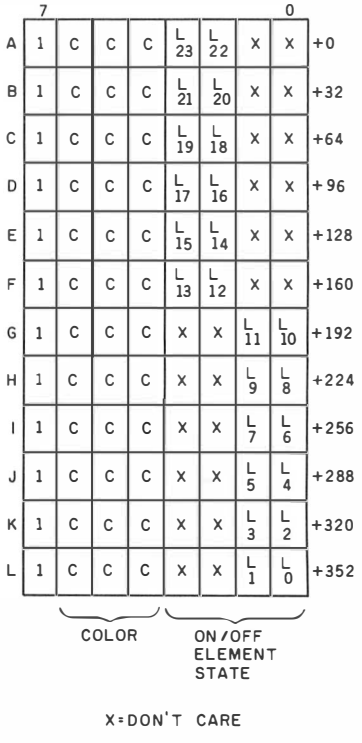
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



Semigraphic 24 continued

COLORS

BORDER BLACK

CHARACTER/BACKGROUND

CCC
000 GREEN
001 YELLOW
010 BLUE
011 RED
100 BUFF
101 CYAN
110 MAGENTA
111 ORANGE

USE

1. A = PEEK (65314) : POKE 65314, (A AND 7).
2. POKE 65477, 1 : POKE 65475, 1 : POKE 65472, 0.
3. Set START by POKEs to 65,478-65,491. See Chapter 2.
4. Store bytes in 6144-byte video display area as required.

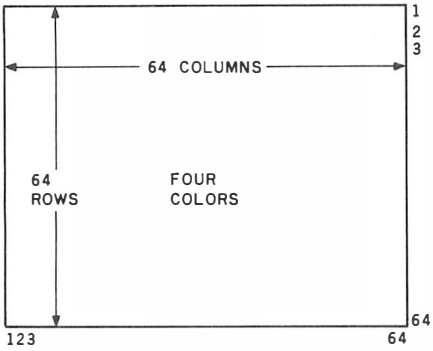
MODE 7

GRAPHICS 64 × 64 F

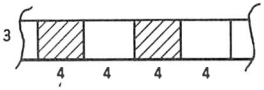
BASIC MODE

NOT IMPLEMENTED. MUST BE SET AS DESCRIBED UNDER "USE."

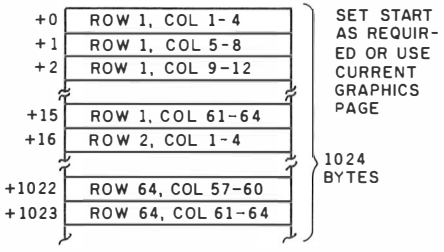
DISPLAY FORMAT



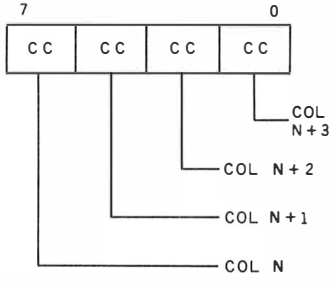
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



COLORS

BORDER

GREEN IF CSS = 0, BUFF IF CSS = 1

CHARACTER/BACKGROUND

CC		
00	GREEN	CSS = 0
01	YELLOW	
10	BLUE	
11	RED	
00	BUFF	CSS = 1
01	CYAN	
10	MAGENTA	
11	ORANGE	

Graphics 64 × 64 F continued

USE

1. A = PEEK (65314) : POKE 65314, (A AND 7) + 128 + C where C = 128 for CSS = 1 or 0 for CSS = 0.
2. POKE 65473, 1 : POKE 65474, 0 : POKE 65476, 0.
3. Set START by POKEs to 65,478–65,491 (see Chapter 2) or use current graphics page (Extended Color BASIC).
4. Store bytes in 1024-byte video display area as required.

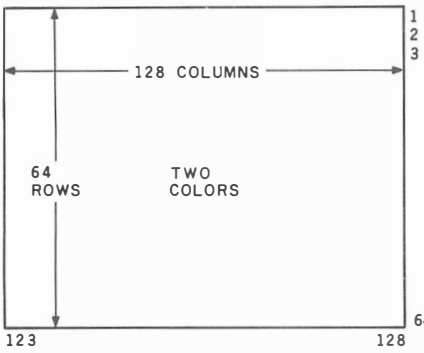
MODE 8

GRAPHICS 128 × 64 T

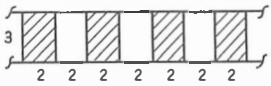
BASIC MODE

NOT IMPLEMENTED. MUST BE SET AS DESCRIBED UNDER "USE."

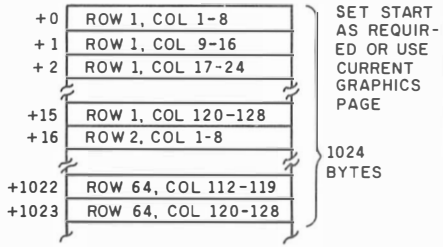
DISPLAY FORMAT



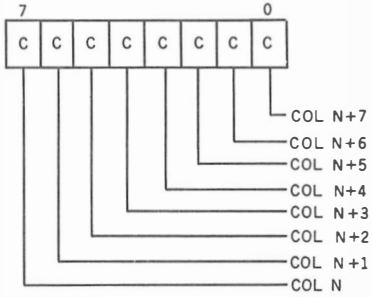
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



COLORS

BORDER

GREEN IF CSS = 0, BUFF IF CSS = 1

CHARACTER/BACKGROUND

C		
0	BLACK	CSS = 0
1	GREEN	
0	BLACK	CSS = 1
1	BUFF	

USE

1. A = PEEK (65314) : POKE 65314, (A AND 7) + 128 + 16 + C where C = 8 for CSS = 1 or 0 for CSS = 0.
2. POKE 65473, 1 : POKE 65474, 0 : POKE 65476, 0.
3. Set START by POKE to 65,478–65,491 (see Chapter 2) or use current graphics page (Extended Color BASIC).
4. Store bytes in 1024-byte video display area as required.

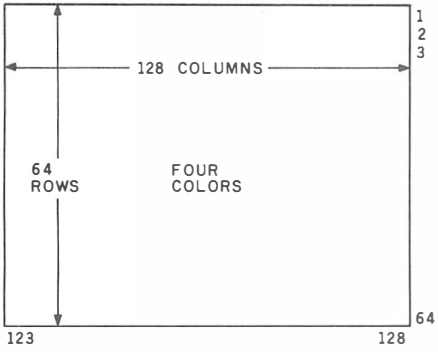
MODE 9

GRAPHICS 128 × 64 F

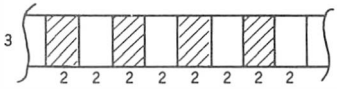
BASIC MODE

NOT IMPLEMENTED. MUST BE SET AS DESCRIBED UNDER "USE."

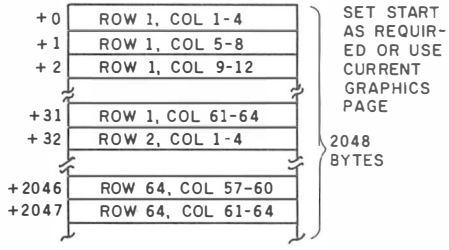
DISPLAY FORMAT



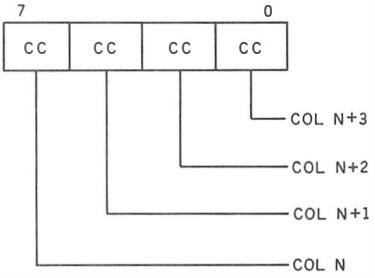
ELEMENT FORMAT



MEMORY MAPPING



BYTE MAPPING



COLORS

BORDER

GREEN IF CSS = 0, BUFF IF CSS = 1

CHARACTER/BACKGROUND

CC		
00	GREEN	CSS = 0
01	YELLOW	
10	BLUE	
11	RED	
00	BUFF	CSS = 1
01	CYAN	
10	MAGENTA	
11	ORANGE	

Graphics 128 x 64 F continued

USE

1. A = PEEK (65314) : POKE 65314, (A AND 7) + 128 + 32 + C where C = 8 FOR CSS = 1 or 0 for CSS = 0.
2. POKE 65472, 0 : POKE 65475, 0 : POKE 65476, 1.
3. Set START by POKEs to 65,478–65,491 (see Chapter 2) or use current graphics page (Extended Color BASIC).
4. Store bytes in 1024-byte video display area as required.

MODE *10*

GRAPHICS 128 × 96 T

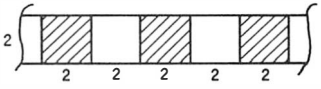
BASIC MODE

PMODE 0 (Ext) OR "USE."

DISPLAY FORMAT

128 COLUMNS BY 96 ROWS,
TWO COLORS

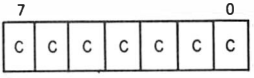
ELEMENT FORMAT



MEMORY MAPPING

1536 BYTES. EACH BYTE HOLDS 8
COLUMNS. SET START OR USE PAGE.

BYTE MAPPING



COLORS

BORDER

GREEN OR BUFF

CHARACTER/BACKGROUND

0 = BLK 1 = GRN IF CSS = 0
0 = BLK 1 = BUFF IF CSS = 1

USE

A = PEEK (65314) : POKE 65314, (A AND 7) + 128 + 32 + 16 + C
POKE 65476, 0 : POKE 65475, 1 = POKE 65473, 1

MODE *11*

GRAPHICS 128 × 96 F

BASIC MODE

PMODE 1 (Ext) OR "USE."

DISPLAY FORMAT

128 COLUMNS BY 96 ROWS,
FOUR COLORS

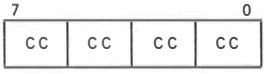
ELEMENT FORMAT

Same as Mode 10

MEMORY MAPPING

3072 BYTES. EACH BYTE HOLDS 4
COLUMNS. SET START OR USE PAGE.

BYTE MAPPING



COLORS

BORDER

GREEN OR BUFF

CHARACTER/BACKGROUND

00 = GREEN
01 = YELLOW
10 = BLUE
11 = RED
} (CSS 0)

00 = BUFF
01 = CYAN
10 = MAGENTA
11 = ORANGE
} (CSS 1)

USE

```
A = PEEK (65314) : POKE 65314, (A AND 7) + 128 + 64 + C
POKE 65477, 1 : POKE 65474, 0 : POKE 65472, 0
```

MODE *12*

GRAPHICS 128 × 192 T

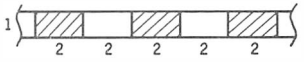
BASIC MODE

PMODE 2 (Ext) OR "USE."

DISPLAY FORMAT

128 COLUMNS BY 192 ROWS,
TWO COLORS

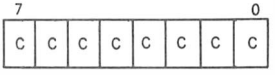
ELEMENT FORMAT



MEMORY MAPPING

3072 BYTES. EACH BYTE HOLDS 8
COLUMNS. SET START OR USE PAGE.

BYTE MAPPING



COLORS

BORDER

GREEN OR BUFF

CHARACTER/BACKGROUND

SAME AS 128 × 96 T

USE

```
A = PEEK (65314) : POKE 65314, (A AND 7) + 128 + 64 + 16 + C
POKE 65477, 1 : POKE 65474, 0 : POKE 65473, 1
```

MODE *13*

GRAPHICS 128 × 192 F

BASIC MODE

PMODE 3 (Ext) OR "USE."

DISPLAY FORMAT

128 COLUMNS BY 192 ROWS,
FOUR COLORS.

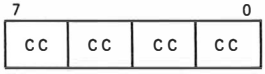
ELEMENT FORMAT

Same as Mode 12

MEMORY MAPPING

6144 BYTES. EACH BYTE HOLDS 4
COLUMNS. SET START OR USE PAGE.

BYTE MAPPING



COLORS

BORDER

GREEN OR BUFF

CHARACTER/BACKGROUND

SAME AS 128 × 96 F

USE

```
A = PEEK (65314) : POKE 65314, (A AND 7) + 128 + 64 + 32 + C
POKE 65477, 1 : POKE 65475, 1 : POKE 65472, 0
```

MODE *14*

GRAPHICS 256 × 192 T

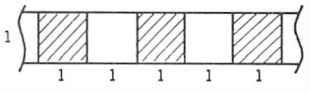
BASIC MODE

PMODE 4 (Ext) OR "USE."

DISPLAY FORMAT

256 COLUMNS BY 192 ROWS,
TWO COLORS.

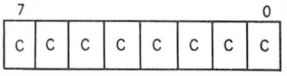
ELEMENT FORMAT



MEMORY MAPPING

6144 BYTES. EACH BYTE HOLDS 8
COLUMNS. SET START OR USE PAGE.

BYTE MAPPING



COLORS

BORDER

GREEN OR BUFF

CHARACTER/BACKGROUND

SAME AS 128 × 96 T

USE

```
A = PEEK (65314) : POKE 65314, (A AND 7) + 128 + 64 + 32 + 16 + C
POKE 65477, 1 : POKE 65475, 1 : POKE 65472, 0
```

Index

Radio Shack
A Division of Tandy Corp.