Graphics Workshop
Block Shapes Part 1

by Robert R. Devine
P.O. Box 10
Adona, Arkansas 72001

If you're like me, you can't help but be impressed by some of the fantastic graphics that are appearing in many of today's computer games. When I see a program with 15 or 20 spaceships, not to mention a fancy background, all moving smoothly and quickly around the Hi-Res screen, I can't help but envy the knowledge those programmers must have. Part of my reason for preparing this series is to help you and me come a few steps closer to being able to design graphics with that same kind of excellence.

THREE GRAPHICS METHODS

There are three basic methods for creating and manipulating shapes for use with Hi-Res graphics, not including the rather inelegant method of writing scores of HPLOT statements.

Most of us start out with the **VECTOR shape method**, which is built into your Apple and animates shapes with the DRAW and XDRAW commands. Vector shapes are very easy to animate, but they are quite slow, they are difficult to create and modify, and they only work well with reasonably small shapes.

The next method is the **HPLOT shape method**, which is quite an improvement over the vector method. (See Nibble Vol. 4/No. 2 for a complete discussion of HPLOT shapes.) Hplot shape tables are easier to create, and will move much faster, especially where large shapes without a great deal of detail are required. In fact, if the shape is large enough, the HPLOT shape may be the fastest method available.

The third method, and the one most often used in those high speed animation games that we all love, is the **BLOCK or BYTE shape method**. It is this method of shape creation and animation that we will deal with in this series. BLOCK shapes are the easiest of all shapes to create, they will move the fastest, and complexity of detail does not affect execution speed. As an added bonus, a BLOCK shape can, in many cases, neatly run over background areas and restore the background as it leaves the neighborhood.

The drawback to block shapes is that some rather complex programming logic is needed to manipulate them. Since we will be dealing with specific bytes of Hi-Res memory, we'll also need a quick and easy method of determining Hi-Res screen memory addresses.

We will attempt, in this series, to develop and explain specific machine language routines that will accomplish these tasks. It won't be necessary for you to be an assembly language programmer to make use of these routines. We'll try to establish how the routines work, and how you can make use of them from within your own BASIC programs.

CREDIT WHERE IT'S DUE

Many of the approaches that we'll be looking at came to me after reading the HI-RES SECRETS package published by Avant Garde Creations. If you're really serious about Hi-Res graphics, this package might be something to look into. As a certain publisher said to me about this package, "The information about Hi-Res graphics **IS** there, but you're going to need to spend a lot of time, and make a real effort to find it." He was right.

LET'S EXAMINE THE HI-RES SCREENS

Both of the Hi-Res screens or pages in the Apple are 8192 bytes long, and illuminate up to 53,760 individual dots on each screen. **Both pages can be thought of as rectangular blocks of memory, 192 bytes high by 40 bytes wide. Page 1 (HGR) begins at memory address 8192 ($2000) and extends upward to address 16383 ($3FFF). Page 2 (HGR2) begins at 16384 ($4000) and extends upwards to 24575 ($5FFF).**

The 192 high vertical stack of bytes make up lines 0-191 of the Y-axis. Each horizontal row of bytes (40 on each row) make up the 280 dot resolution of the X-axis. So you ask, how do 40 bytes make 280 dots? Each byte is made up of 8 BITS, 7 of which (BITS 0-6) are each responsible for 1 dot on the screen. In other words, 40 BYTES (numbered 0-39) × 7 dots (per byte) = 280 dots. The 8th BIT (bit #7) is called the color bit, and though it is not displayed on the screen, it will determine the colors which are displayed by the other 7 bits.

HOW THE BITS ARE ORGANIZED

The normal convention in describing a byte is to arrange the bits, counting from right to left.

For example, a normal Hi-Res byte containing the number 03 . . . Bit number 7 6 5 4 3 2 1 0
Bit status 0 0 0 0 0 0 1 1

A displayed byte with the value 03 will have only bits 0 and 1 lit, both of which will be white, due to the fact that bit 7 (the color bit) is a 0.

However, just to make things confusing, a byte with the value 03 will appear on the screen as . . . W W B B B B B. (W=white B=black). The reason for this is that the contents of a Hi-Res byte is displayed on the screen in reverse order.

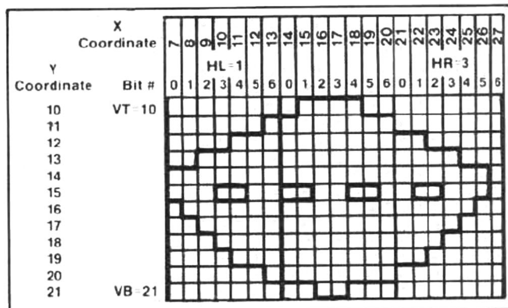
For the balance of our discussion, we will, due to this reversing effect, think of the bits in a byte as being numbered from left to right . . . 0 1 2 3 4 5 6 7, with only the **7 leftmost bits** as the ones which appear on the screen. Don't let all the technical jargon scare you off. While all this stuff will need to be understood if you want to become a real expert at Hi-Res animation, you will be quite able to use the routines in this series without understanding all the details.

A PICTURE IS WORTH 1000 WORDS

But then that's what this is all about . . . isn't it? To make things a bit easier to visualize, let's define a graphics shape, and look at all the information you could possibly want to know about it. You won't really need all of this data, but it will be helpful to see how things fit together. We'll refer to this "picture" frequently in our discussion.

FIGURE 1

HGR-Page 1 Leftmost byte Memory address Decimal	Hex	HGR2-Page 2 Leftmost byte Memory address Decimal	Hex
10368	$2880	18560	$4880
11392	$2C80	19584	$4C80
12416	$3080	20608	$5080
13440	$3480	21632	$5480
14464	$3880	22656	$5880
15488	$3C80	23680	$5C80
8448	$2100	16640	$4100
9472	$2500	17664	$4500
10496	$2900	18688	$4900
11520	$2D00	19712	$4D00
12544	$3100	20736	$5100
13568	$3500	21760	$5500



Shape bytes as they would appear in Hi-Res memory on Page 1.

The shape bytes would appear exactly the same on Page 2, except you would need to add $2000 to each of the referenced memory addresses.

		HL		HR
$2881-2883	VT	00	1E	00
$2C81-2C83		40	7F	00
$3081-3083		70	7F	03
$3481-3483		7C	7F	0F
$3881-3883		7F	7F	3F
$3C81-3C83		67	4C	39
$2101-2103		7E	7F	1F
$2501-2503		7C	7F	0F
$2901-2903		78	7F	07
$2D01=2D03		70	7F	03
$3101-3103		40	7F	00
$3501-3503	VB	00	0C	00

Block shape table as it would appear in memory:

00 0C 00 00 7F 40 03 7F 70 07 7F 78 0F 7F 7C 1F 7F 7E 39 4C 67 3F 7F 7F 0F 7F 7C 03 7F 70 00 7F 40 00 1E 00

For our discussion we will be dealing with the alien spaceship described in our example. Those of you who read the article on HPLOT SHAPES will recognize this as the same shape we used in that method of shape creation. So you might ask at this point . . . what's different about BLOCK shapes? The answer is quite simple . . . absolutely nothing!

Our spaceship is located on the Hi-Res screen from horizontal (X) coordinates 7 through 26, and vertical (Y) coordinates 10 through 21. It makes no difference **HOW** the shape got there. What is important is that once the shape **IS** there, everything in our example will be valid.

All our shape really consists of is 36 bytes of data with different BIT patterns. In other words, the information in our example is not concerned with the creation method used, but rather with the results after the shape has been placed on the Hi-Res screen. The information in our example would be the same if we'd drawn the shape with the game paddles or a bunch of HPLOT statements, loaded it from disk, used an HPLOT or vector shape table, or whatever.

WHAT IS A BLOCK SHAPE?

A block shape is simply a rectangular block of Hi-res memory bytes, and the bit patterns contained in these bytes. In our example, the block shape is 12 bytes high and 3 bytes wide, for a total of 36 bytes.

A block **shape table** contains only the data described in the shape bytes, with no other information about the shape being included. Those of you who dealt with this same shape in the HPLOT article will remember that it took 97 bytes to describe this shape, while the same shape is described here with only 36 bytes. Another nice thing about BLOCK shapes is that you never need to set color. In fact a block shape can be multi-colored, since it's the color bit and the location of the other bits containing **1's** that determine what color will be displayed.

DEFINING A BLOCK SHAPE

In order to deal with a block shape, there are 4 pieces of data that we'll need for a shape definition.

The first thing we'll need is **VT, the vertical top Y-coordinate**.

Next we'll need **VB, the vertical bottom Y-coordinate**.

Finally we'll need to know **HL horizontal left**, and **HR horizontal right**.

HL and HR refer to the leftmost and rightmost **BYTE** that is involved in the shape. Since our screen is only 40 bytes wide, HL and HR will always be in the range of 0-39, with byte 0 (zero) on the leftmost side of the screen.

You should also note at this point that if our sample shape was shifted to the right as little as 2 dots (to X-coordinates 9-28), it would then be 4 bytes wide, and 33% larger. This is a good thing to keep in mind, as a simple shift into an extra byte can slow execution time.

As you look at the shape in our example, you will see that **not all of the bytes involved in the block shape are actually parts of the shape itself. In fact there are 6 bytes that have nothing at all to do with the shape. They are however parts of the rectangular block of space that the shape occupies at the moment, and are therefore necessary parts of the block shape.**

WORKING THROUGH A BLOCK SHAPE

As we SCAN, DRAW, or manipulate our block shapes, we will always **begin at the bottom rightmost corner (HR/VB), and finish at the top leftmost corner (HL/VT).**

In our example, we would start at memory address $3503, and then move through $3502, $3501, $3103, $3102, etc. until completing our shape at $2881. Basically all we're doing is getting a shape byte from our table, putting it into a Hi-Res memory address, getting the next byte from our table, putting it into the next memory address, and so on until we're done. We won't need to worry about using cumbersome DRAW or HPLOT routines, or any of that other garbage. Sounds real neat and easy, doesn't it?

Before you answer that question, let's see if you can answer this one: What is the memory address of byte 17 on vertical line 156? If you can't answer that right away, look at our example and find the address of the first byte (byte 0) for vertical line 22. It should be easy; we've already shown the address for line 21.

FINDING HI-RES MEMORY ADDRESSES

If you haven't already noticed that Hi-Res memory addresses are all mixed up, now is as good a time as any for an indoctrination. Take a second look at the leftmost four columns in our example, and you'll see that the addresses are one big mess. There is a pattern to address locations, but unless your Apple is installed inside your skull, it's not likely that you'll be able to find an address without some effort. This problem is one of the major reasons why DRAW and HPLOT are rather slow. These commands make use of, and are constantly referring to another routine that's built into your Apple which resides at memory location hex $F411. This routine is called HPOSN, and among other things, it calculates Hi-Res memory addresses.

At this point you may be wondering how block shapes can be so much faster when we're dealing exclusively with memory addresses. The answer is simple. First, a BLOCK shape table normally uses fewer bytes than any other type of table. Next, we will only need to find a Hi-Res memory address **once** for each Y-coordinate (in our sample only 12 times). Finally, we won't use HPOSN to find our addresses.

USING THE YTABLE TO GET MEMORY ADDRESSES

If you have the space in memory, you can speed up the way your Apple finds Hi-Res screen addresses by setting up an address table. Let's face it, YTABLE and its access routine will burn-up 623 bytes of memory, but if you have the room, the speed increase (about 20%), will be well worth the price.

All of the routines that we use will make use of YTABLE instead of HPOSN. However if you really get into a pinch and need the memory, you can still use the routines with HPOSN by making the following changes to the routines which we'll be using:

Remove JSR YADDR wherever you find it, and replace it with . . .

```
A2 00      LDX #$00
A0 00      LDY #$00
20 11 F4   JSR $F411 (HPOSN)
```

You will need to re-assemble the routines, move them to different memory locations, and change your entry points if you change to HPOSN.

Listing 1 shows the HEX bytes for YTABLE, as well as the YADDR and SETUP routines. To enter them into memory, you'll need to be in the monitor (which is entered with CALL-151). Now that you're in the monitor, with the * cursor, enter 9391:A4 06 B1 CE etc. until you've filled up about 4 lines on the screen. Then press RETURN, enter another (:) and fill up another 4 lines. Your first piece of data must always immediately follow the colon (:). Continue the process until the entire hex listing is entered. Once it's all entered, save it to disk with BSAVE YTABLE,A$9391,L623.

HOW THE YTABLE WORKS

The actual table of memory addresses runs from addresses $93C0 through $95FF. Addresses $9391 through $93BF are really a machine language routine which accesses the table and a routine to setup the table pointers. Let's disassemble these routines to see how they work. **(See Listing 2).**

The sole function of YTABLE is to retrieve the memory address of byte 0 (the leftmost), as it relates to each of the 192 Y-coordinates.

We don't need to know any of the other addresses on the lines, as our other routines will handle the appropriate offset address from byte 0. YTABLE contains all of the data necessary to find this information on both Hi-Res pages.

If you look again at the first 4 columns of addresses in the example, you will note that the HEX addresses for any given Y-coordinate do have one thing in common: **The last 2 digits (Lo-byte) in each of the Hex addresses are the same, regardless of whether we're on page 1 or page 2.** This fact will hold true for all 192 Y-coordinates.

Bytes $9480 through $953F of YTABLE contain the lo-byte portion of the address that we're looking for.

Bytes $93C0 through $947F contain the hi-byte portion of the address for each Y-coordinate on HGR2-Page 2.

Bytes $9540 through $95FF contain the hi-byte portion of the address for each Y-coordinate on HGR-page 1.

To use this routine you should first CALL or JSR the SETUP routine located at 37799 ($93A7). All this does is to put the starting addresses of each of the three table segments ($9480, $9540, and $93C0) into the proper zero-page pointers for use by YADDR. We could have done this with a series of POKEs, but it's a hassle trying to remember the proper POKEs. Once you've been through SETUP, you won't need to use it again, unless you somehow manage to damage these pointers.

To use YADDR, just POKE the Y-coordinate (for which you need an address) into memory location 6 (POKE 6,Y), and CALL 37777.

The routine first moves to the Yth element in the Lo-byte table, and stores what it finds in memory location 38 ($26). Next it checks memory location 230 ($E6) to find out which Hi-Res page we're on. If we're on **page 1**, it gets the Yth element from the HI-BYTE/Page 1 table and stores what it finds in memory location 39 ($27). If we're on **page 2** it goes to the HI-BYTE/Page 2 table and does the same thing.

Now if you wanted to **find the 0 byte address for line 22**, as we discussed, you could do the following:

First be sure that YTABLE is in memory and CALL 37799 to SETUP the pointers. Next **POKE 6,22** to tell the routine which **Y address** we want. If you want the address for **page 1**, enter **POKE 230,32**; if **page 2**, enter **POKE 230,64**. Finally **CALL 37777** to get the address. To recover the address, enter **ADDRESS=PEEK(38)+PEEK(39)*256:PRINT ADDRESS**, and the address, in decimal form, will appear.

LET'S TEST YTABLE

It's rather important to make sure that there aren't any errors in your table. If there are any mistakes, you could find your Apple displaying DOS commands, program lines, or whatever, as shapes on the Hi-Res screen. Or even

LISTING 1 YTABLE ADDRESSES

```
*9391.95FF
9391- A4 06 B1 CE 85 26 A5
9398- E6 C9 40 D0 05 B1 DE 85
93A0- 27 60 B1 EE 85 27 60 A9
93A8- 80 85 CE A9 94 85 CF A9
93B0- 40 85 EE A9 95 85 EF A9
93B8- C0 85 DE A9 93 85 DF 60
93C0- 40 44 48 4C 50 54 58 5C
93C8- 40 44 48 4C 50 54 58 5C
93D0- 41 45 49 4D 51 55 59 5D
93D8- 41 45 49 4D 51 55 59 5D
93E0- 42 46 4A 4E 52 56 5A 5E
93E8- 42 46 4A 4E 52 56 5A 5E
93F0- 43 47 4B 4F 53 57 5B 5F
93F8- 43 47 4B 4F 53 57 5B 5F
9400- 40 44 48 4C 50 54 58 5C
9408- 40 44 48 4C 50 54 58 5C
9410- 41 45 49 4D 51 55 59 5D
9418- 41 45 49 4D 51 55 59 5D
9420- 42 46 4A 4E 52 56 5A 5E
9428- 42 46 4A 4E 52 56 5A 5E
9430- 43 47 4B 4F 53 57 5B 5F
9438- 43 47 4B 4F 53 57 5B 5F
9440- 40 44 48 4C 50 54 58 5C
9448- 40 44 48 4C 50 54 58 5C
9450- 41 45 49 4D 51 55 59 5D
9458- 41 45 49 4D 51 55 59 5D
9460- 42 46 4A 4E 52 56 5A 5E
9468- 42 46 4A 4E 52 56 5A 5E
9470- 43 47 4B 4F 53 57 5B 5F
9478- 43 47 4B 4F 53 57 5B 5F
9480- 00 00 00 00 00 00 00 00
9488- 80 80 80 80 80 80 80 80
9490- 00 00 00 00 00 00 00 00
9498- 80 80 80 80 80 80 80 80
94A0- 00 00 00 00 00 00 00 00
94A8- 80 80 80 80 80 80 80 80
94B0- 00 00 00 00 00 00 00 00
94B8- 80 80 80 80 80 80 80 80
94C0- 28 28 28 28 28 28 28 28
94C8- A8 A8 A8 A8 A8 A8 A8 A8
94D0- 28 28 28 28 28 28 28 28
94D8- A8 A8 A8 A8 A8 A8 A8 A8
94E0- 28 28 28 28 28 28 28 28
94E8- A8 A8 A8 A8 A8 A8 A8 A8
94F0- 28 28 28 28 28 28 28 28
94F8- A8 A8 A8 A8 A8 A8 A8 A8
9500- 50 50 50 50 50 50 50 50
9508- D0 D0 D0 D0 D0 D0 D0 D0
9510- 50 50 50 50 50 50 50 50
9518- D0 D0 D0 D0 D0 D0 D0 D0
9520- 50 50 50 50 50 50 50 50
9528- D0 D0 D0 D0 D0 D0 D0 D0
9530- 50 50 50 50 50 50 50 50
9538- D0 D0 D0 D0 D0 D0 D0 D0
9540- 20 24 28 2C 30 34 38 3C
9548- 20 24 28 2C 30 34 38 3C
9550- 21 25 29 2D 31 35 39 3D
9558- 21 25 29 2D 31 35 39 3D
9560- 22 26 2A 2E 32 36 3A 3E
9568- 22 26 2A 2E 32 36 3A 3E
9570- 23 27 2B 2F 33 37 3B 3F
9578- 23 27 2B 2F 33 37 3B 3F
9580- 20 24 28 2C 30 34 38 3C
9588- 20 24 28 2C 30 34 38 3C
9590- 21 25 29 2D 31 35 39 3D
9598- 21 25 29 2D 31 35 39 3D
95A0- 22 26 2A 2E 32 36 3A 3E
95A8- 22 26 2A 2E 32 36 3A 3E
95B0- 23 27 2B 2F 33 37 3B 3F
95B8- 23 27 2B 2F 33 37 3B 3F
95C0- 20 24 28 2C 30 34 38 3C
95C8- 20 24 28 2C 30 34 38 3C
95D0- 21 25 29 2D 31 35 39 3D
95D8- 21 25 29 2D 31 35 39 3D
95E0- 22 26 2A 2E 32 36 3A 3E
95E8- 22 26 2A 2E 32 36 3A 3E
95F0- 23 27 2B 2F 33 37 3B 3F
95F8- 23 27 2B 2F 33 37 3B 3F
```

worse, you could land up loading shape bytes on top of GOD-ONLY-KNOWS-WHAT and causing total destruction.

Here's one good way to test your table . . . Enter the following Applesoft program lines.

```
10 HGR:X=PEEK(49234):CALL 37799
20 FOR Y=0 TO 191:POKE 6,Y:CALL 37777
30 X=PEEK(38)+PEEK(39)*256:POKE X,127:
   NEXT
```

Be sure that **YTABLE** is in memory and **RUN** the program. You should end up with a vertical white line, 7 dots wide, drawn from the top to the bottom on the left side of the screen. If there are any irregularities, or other lines on the screen, you'll know there is an error, and roughly where in the table the error occurred. If you're curious about the **X=PEEK(49234)** in line 10, it simply sets **page 1** to **full screen graphics**.

To run the test on page 2, simply change line 10 to read, **10 HGR2:CALL 37799**.

LET'S CREATE A BLOCK SHAPE

If you thought we'd never get to actually creating a BLOCK SHAPE, don't despair. We're finally ready to begin work.

So how does one create a Block shape? The answer is simple: the same way a 700 lb. Gorilla eats bananas . . . ANY WAY HE WANTS TO!! Seriously now, **the first step in creating a block shape is to physically create or load a shape into Hi-Res memory**. How you go about doing that is your business. Frankly I usually create an Applesoft program that uses HPLOT statements. Then I can change it around until I like the shape. As we said earlier, **it's not important HOW the shape originally gets ON the screen, we really aren't dealing with a BLOCK SHAPE until we start to get it OFF the screen, and into a BLOCK TABLE.**

THE SCAN ROUTINE

Our SCAN routine is 48 bytes long and resides just under YTABLE. It is this routine that does all the work of converting whatever you've drawn on the screen into a BLOCK TABLE. Once you have it in memory, you can create all the block shapes you want, with very little effort.

Let's start out by making a short little program that draws our sample shape on the screen.

```
100 HGR : HCOLOR=3
110 FOR Y = 10 TO 14: READ X: HPLOT
    X,Y: READ X: HPLOT TO X,Y: NEXT
120 FOR Y = 1 TO 5: READ X: HPLOT X,15:
    READ X: HPLOT TO X,15: NEXT
130 FOR Y = 16 TO 21: READ X: HPLOT
    X,Y: READ X: HPLOT TO X,Y: NEXT
200 DATA 15,18,13,20,11,22,9,24,7,26,7,9,
    12,13,16,17,20,21,24,26,8,25,9,24,10,23,
    11,22,13,20,16,17
```

When you RUN this, the space ship from Figure 1 will appear on the screen. Now we're ready to convert it into a BLOCK SHAPE. **We already know that the highest vertical coordinate is 10, and the lowest is 21. Therefore we know the proper values for VT and VB. Getting the proper values for HR and HL is a little different, however.**

As we've already found out, our screen is only **40 bytes wide**, and **each byte contains 7 X-coordinates**. Byte 0 holds coordinates 0–6, byte 1 holds coordinates 7–13, byte 2 holds coordinates 14–20, and so forth until we get to byte 39 which holds coordinates 273–279. Therefore, since our shape starts at 7, the **value of HL**=**1**, and since the shape ends at 26, the **value of HR**=**3**.

The first thing you'll need to provide, before using SCAN, is the appropriate values for **VT**, **VB**, **HR**, and **HL**, which we just found were **10**, **21**, **3**, and **1** respectively. **These values need to be POKEd into memory locations 252**, **253**, **254**, and **255** to tell the SCANner where on the Hi-Res screen the shape is located. **Next we need to tell the routine where in memory it is to assemble and store the completed block shape table**.

Your Apple contains "pages" of memory. These pages have nothing to do with the graphics pages that we've been talking about so far. Instead they refer to blocks of memory, each of which is 256 bytes long. **Each of our shapes will begin at the very first byte of a given memory page**, and our shape table may, if needed, overflow onto the next page of memory. In other words, your shape may be longer than 256 bytes.

You'll need to decide on which memory page you'll store your shape table. We will number our shape with the page # where our shape is stored. For our exercise we'll store our shape on page #144, which starts at $9000. (Note: 144 is the decimal equivalent of $90). To tell the SCAN routine where to store our shape, POKE the shape number into memory location 251 ($FB): **POKE 251, SHAPE#**. Now simply CALL 37729 and the SCAN routine will create your shape table. That was easy, wasn't it? The SCAN routine is contained in **LISTING 3.**

Since SCAN and ALL of our other routines will use YTABLE, you'll **always** need to be sure it's also in memory, and that you've CALLed the SETUP routine first.

SAVING THE SHAPE

To save your shape table to disk, enter **BSAVE SHAPE #144,A$9000,L36**. The format for SAVEing a shape table is BSAVE (NAME), A$(START ADDRESS),L(NUMBER OF BYTES). Bear in mind that your Apple doesn't need to be in the graphics mode when you run SCAN, as long as your shape is in graphics memory. If you're in TEXT mode when you run SCAN, be sure to POKE a 32 or 64 into memory location 230, to tell SCAN which Hi-Res page (1 or 2) to find the shape on.

HOW SCAN WORKS

The assembly listing explains each step, so we won't get into it in much detail. Basically the routine gets the address for VB, then moves over to HR. Then it gets the byte off the screen and puts it into the table. Then it moves left 1 byte and checks to see if we want this byte. If so, it repeats the process until we get to HL. Once we get to HL it moves up 1 line, gets the address, and moves over to HR again. This process repeats until we've gotten all of the bytes through VT/HL. At this point the routine is finished.

THE DRAW ROUTINE

Now that we have the ability to create BLOCK TABLES, we'll need a routine that will DRAW them on the screen. **Listing 4** is a listing of DRAW, which starts at $932F, and fits right under the SCAN routine.

To use the DRAW routine, you'll need to provide the same information as we did for the SCAN routine, namely **VT**, **VB**, **HR**, **HL**, **and SHAPE#**. The only real difference between the routines is that instead of getting our shape from the Hi-Res screen and putting it into a table, we'll get it from the table and put it on the Hi-Res screen. Otherwise the SCAN and DRAW routines work the same.

LISTING 2

```
                1000 *
                1002 *
                1010 *    YTABLE SETUP & ACCESS
                1020 *
                1030 * GRAPHICS WORKSHOP II
                1040 * BY ROBERT R. DEVINE
                1050 *
                1060 * COPYRIGHT (C) 1983
                1070 * BY MICROSPARC INC
                1080 * ALL RIGHTS RESERVED
                1090 *
                1100 * SETUP Y TABLE WITH CALL 37799
                1110 * RETRIEVE ADDRESSES WITH CALL 37777
                1120  .OR $9391
                1130  .TA $800
0006-           1140 VCOORD .EQ $06
00CE-           1150 LTBLO  .EQ $CE      ** POKE 206,128 ($80)
00CF-           1160 LTBHI  .EQ $CF      ** POKE 207,148 ($94)
001E-           1170 HTBLP1 .EQ $EE      ** POKE 238,64 ($40)
001F-           1180 HTBHP1 .EQ $EF      ** POKE 239,149 ($95)
00DE-           1190 HTBLP2 .EQ $DE      ** POKE 222,192 ($C0)
00DF-           1200 HTBHP2 .EQ $DF      ** POKE 223,147 ($93)
00E6-           1210 SCREEN .EQ $E6
9391- A4 06     1220 YADDR LDY $06       ** CALL 37777 TO ENTER
9393- B1 CE     1230  LDA (LTBLO),Y      ** GET SCREEN ADDR. LO BYTE
9395- 85 26     1240  STA $26            ** STORE IT
9397- A5 E6     1250  LDA SCREEN         ** GET SCREEN POINTER
9399- C9 40     1260  CMP #$40           ** ARE WE ON SCREEN 2 ?
939B- D0 05     1270  BNE P1             ** NO-GET HI BYTE IN PAGE 1 TABLE
939D- B1 DE     1280  LDA (HTBLP2),Y     ** GET HI BYTE FOR PAGE 2
939F- 85 27     1290  STA $27            ** STORE IT
93A1- 60        1300  RTS                ** DONE-EXIT
93A2- B1 EE     1310 P1 LDA (HTBLP1),Y   ** GET HI BYTE FOR PAGE 1
93A4- 85 27     1320  STA $27            ** STORE IT
93A6- 60        1330  RTS                ** DONE-EXIT
93A7- A9 80     1340 SETUP LDA #$80      ** CALL 37799 TO ENTER
93A9- 85 CE     1350  STA $CE            ** POKE 206,128
93AB- A9 94     1360  LDA #$94
93AD- 85 CF     1370  STA $CF            ** POKE 207,148
93AF- A9 40     1380  LDA #$40
93B1- 85 EE     1390  STA $EE            ** POKE 30,64
93B3- A9 95     1400  LDA #$95
93B5- 85 EF     1410  STA $ EF           ** POKE 31,149
93B7- A9 C0     1420  LDA #$C0
93B9- 85 DE     1430  STA $DE            ** POKE 222,192
93BB- A9 93     1440  LDA #$93
93BD- 85 DF     1450  STA $DF            ** POKE 223,147
93BF- 60        1460  RTS
                1470 * END OF ROUTINE
```

HOW WE ERASE BLOCK SHAPES

There is one new statement in this routine (line 1450) that we should take a moment to look at. It is this statement that allows us to **use the same routine to ERASE as well as to DRAW**. It is also this statement that allows us to restore the background after erasing.

1450 EOR (HBASL),Y EOR means Exclusive-OR with Accumulator

Here's what it does: First we load the shape byte from our table into the Accumulator (line 1440). For you non-assembly programmers, the Accumulator is simply a special Register (byte) inside the 6502 microprocessor.

Next we compare (EOR) the bits in the Hi-Res screen address that we're presently working on, with the shape byte in the Accumulator, and modify the shape byte according to the following rules. (This has no effect on the shape byte in our table.)

Each BIT in the screen address is compared to the matching BIT in the Accumulator. If EITHER of the two bits is a 1, then the bit in the Accumulator is set to 1; however if BOTH bits are 1, or if BOTH bits are 0, then the Accumulator bit is set to 0.

Let's see how this affects us, and use **11001110** as our sample shape byte. If we were DRAWing over a blank background, we would **EOR** with **00000000**, and the result would be **11001110**, leaving our shape byte unaffected.

If we were ERASING our existing shape, we would **EOR** with **11001110**, and the result would be **00000000**, effectively erasing what was there.

Now let's assume that we were going to draw on a byte which contained **background graphics** in bits 4 and 5: **00001100**.

When we first DRAW, we would EOR with 00001100, and the result would be 11000010, which would draw our shape byte, **less** the bits that it shared with the background. Now when we go to ERASE this byte we would EOR with 11000010, and the result would be 00001100, effectively erasing those portions of the shape byte, and restoring the screen byte to its original state. Neat huh?

You should be aware that when we get involved with color shapes and/or backgrounds, things don't always go quite that smoothly.

OTHER USES

While the major emphasis of this article deals with shape animation, don't lose sight of the fact that these routines can be used for many other purposes. You could use the **SCAN** routine to save a screen display, and later use **DRAW** to display portions of Hi-Res pictures, titles, scoreboards etc. They could also be used to move or rearrange parts of a graphics display without needing to redraw the entire screen.

LISTING 3

```
                 1000 *
                 1010 *
                 1020 *    SCAN ROUTINE
                 1030 *
                 1040 *   GRAPHICS WORKSHOP II
                 1050 *   BY ROBERT R. DEVINE
                 1060 *
                 1070 *   COPYRIGHT (C) 1983
                 1080 *   BY MICROSPARC INC.
                 1090 *   ALL RIGHTS RESERVED
                 1100 *
                 1110 *  POKE 252,VT: POKE 253,VB
                 1120 *  POKE 254,HR: POKE 255,HL
                 1130 *  TO TELL SCAN WHERE TO ASSEMBLE
                 1140 *  AND STORE THE SHAPE.
                 1150 *
                 1160 *  THEN CALL 37729 TO SET TABLE
                 1170 *
                 1180  .OR $9361
                 1190  .TA $800
00FC-            1200 VT .EQ $FC          ** DECIMAL 252
00FD-            1210 VB .EQ $FD          ** DECIMAL 253
00FE-            1220 HR .EQ $FE          ** DECIMAL 254
00FF-            1230 HL .EQ $FF          ** DECIMAL 255
0026-            1240 HBASL .EQ $26       ** DECIMAL 38  (SCREEN BASE
0027-            1250 HBASH .EQ $27       ** DECIMAL 39  ADDRESS)
0006-            1260 YO .EQ $6           ** DECIMAL 6
00FA-            1270 BASL .EQ $FA        ** DECIMAL 250 (TABLE BASE
00FB-            1280 BASH .EQ $FB        ** DECIMAL 251 ADDRESS)
9391-            1290 YADDR .EQ $9391     ** DECIMAL 37777 (READ YTABLE)
9361- A9 00      1300 SCAN LDA #0         ** SCANNER CALL 37729 TO ENTER
9363- 85 FA      1310  STA BASL           ** POINT TO START OF TABLE
9365- A5 FD      1320  LDA VB             ** GET BOTTOM Y COORDINATE
9367- 85 06      1330  STA YO             ** STORE IN $6 FOR USE BY YADDR
9369- 20 91 93   1340 L1 JSR YADDR        ** RETURNS-LO=HBASL/HI=HBASH
936C- A4 FE      1350  LDY HR             ** SET Y-REG TO RIGHTMOST BYTE
936E- A2 00      1360  LDX #0             ** SET TABLE OFFSET=0
9370- B1 26      1370 L2 LDA (HBASL),Y    ** GET SHAPE BYTE FROM SCREEN
9372- 81 FA      1380  STA (BASL,X)       ** PUT IN SHAPE TABLE
9374- 88         1390  DEY                ** POINT TO NEXT BYTE <---
9375- 18         1400  CLC
9376- E6 FA      1410  INC BASL           ** POINT TO NEXT TABLE ELEMENT
9378- D0 02      1420  BNE NC1            ** IF <256 BYTES-JUMP
937A- E6 FB      1430  INC BASH           ** PAGE OVERFLOW-GOTO NEXT PAGE
937C- C0 FF      1440 NC1 CPY #$FF        ** HAS Y-REGISTER REACHED 0 ?
937E- F0 04      1450  BEQ NXTLN          ** YES-GOTO NEXT LINE
9380- C4 FF      1460  CPY HL             ** IS Y-REGISTER >=HL ?
9382- B0 EC      1470  BCS L2             ** YES-GET THE NEXT BYTE
9384- C6 06      1480 NXTLN DEC YO        ** MOVE UP TO NEXT LINE
9386- A5 06      1490  LDA YO             ** GET NEW Y COORDINATE
9388- C9 FF      1500  CMP #$FF           ** HAS Y-COORDINATE REACHED 0 ?
938A- F0 04      1510  BEQ RTN            ** YES-WE'RE FINISHED
938C- C5 FC      1520  CMP VT             ** HAVE WE REACHED VT YET ?
938E- B0 D9      1530  BCS L1             ** NO-START THE NEXT LINE
9390- 60         1540 RTN RTS             ** DONE-EXIT ROUTINE
                 1550 * END OF ROUTINE
```

Another idea comes to mind where SCAN and DRAW might be handy. Suppose that you wanted to use your Apple to design a home, or the layout of the furniture. Using SCAN you could easily move an entire room, or individual pieces of furniture, from place to place, until everything looks just right.

In the next part of this series we'll create a reversing routine so that you could even see how the entire house, or just individual rooms, would look if they were completely turned around. The possibilities are endless.

SIMPLE ONE PAGE ANIMATIONS

We won't get into any heavy animation procedures until the next installment, but let's take a moment to see how we might get started animating the shapes. To run the first test, you'll need to have all of the routines in memory, along with the sample spaceship shape.

To enter the shape, you could write a few HPLOT statements (all the proper coordinates are shown in our example), **then CALL 37799 (YTABLE setup), POKE VT, VB, HR & HL (also shown in our example), then POKE 251,144 (set shape #), and finally CALL 37729 (SCAN).** Or, you could simply enter the monitor and enter the Hex bytes shown in our example, beginning at $9000. Finally, save it to disk **BSAVE SHAPE #144,A$9000,L36.** Doing it the long way might be good practice.

Since in our program we're not going to use any character strings, you won't need to worry about HIMEM. However in your normal programs you should set HIMEM just below your lowest shape. Otherwise you could damage your YTABLE address pointers.

HOW THE ANIMATION WORKS

The first things that we need to do are: Be sure that YTABLE is properly set up (line 50); our SHAPE# is entered (line 60); and the starting values for HR and HL are POKEd into memory (lines 90 and 95).

For the rest of our program we're simply using a FOR...NEXT loop to move our shape up and down, POKEing VT and VB based on the value of our loop. After we've gone down and back up again, we move right by changing the values of HR and HL.

In line 180 we check to see if we'll run off the right edge of the screen. The program is rather slow, because other than our ERASE/DRAW routine, we're still using standard Applesoft commands, and the interpreter has to do all the work. Once we get into using other parts of the driver that we'll look at in the next installment, things will get much faster. In fact we may need to change our increment from 2 to 1, just to slow it down a bit. In our test you'll probably note that the shape is a bit slower on the way down; however it's also brighter, with less flicker. That's because of the delay loop (in line 120) between DRAW and ERASE.

To move your shape horizontally across the screen you need to use the same approach, except that you need to change HR and HL every time, to get right-left movement. Horizontal movement will not (at this point) be as smooth as vertical movement due to the fact that we can't make a horizontal move of less than 7 dots per move. In future installments we'll get into SHIFT animation which will let us move less than 1 byte per move.

HOW TO ENTER THESE ROUTINES

You'll note that the first routine we entered was YTABLE, which fit just under DOS. Then came SCAN which fit under YTABLE, and finally DRAW which fit under SCAN. This is the process that we'll follow as we develop the other routines that will make up our BLOCK SHAPE DRIVER. You entered YTABLE as hex bytes, and that will be the easiest way to enter the other routines. Your best bet is to load into memory whatever prior routines that we've worked with, then enter the monitor, and enter the hex bytes for the newest routine.

TABLE 1
SUMMARY OF DRIVER ENTRY POINTS (SO FAR)

Routine Name	CALL Address	Hex Address	Routine Function
SETUP	37799	$93A7	Setup table pointers for YADDR
YADDR	37777	$9391	Returns screen address (Byte 0) for specified Y-coordinate.
SCAN	37729	$9361	Get shape off the screen and into BLOCK TABLE.
DRAW	37679	$932F	DRAW block shape on the screen
ERASE	37679	$932F	ERASE existing shape from the screen.

Special POKEs to use with the driver

POKE	Function
POKE 251,SHAPE#	Sets shape# is equal to the Hi-Byte portion of the memory page where the shape is stored.
POKE 252,VT	Specify location of topmost Y-coordinate
POKE 253,VB	Specify location of lowest Y-coordinate
POKE 254,HR	Specify rightmost BYTE of the shape.
POKE 255,HL	Specify leftmost BYTE of the shape.
POKE 6,Y	Specify Y-coordinate that you want byte 0 address for.

LISTING 4

```
:
:ASM
                  1000 *          .
                  1010 *
                  1020 *    DRAW ROUTINE
                  1030 *
                  1040 *   GRAPHICS WORKSHOP II
                  1050 *   BY ROBERT R DEVINE
                  1060 *
                  1070 *   COPYRIGHT (C) 1983
                  1080 *   BY MICROSPARC INC.
                  1090 *   ALL RIGHTS RESERVED
                  1100 *
                  1110 *  POKE 251,SHAPE TABLE PAGE #
                  1120 *  POKE 252,VT: POKE 253,VB
                  1130 *  POKE 254,HR: POKE 255,HL
                  1140 *  CALL 37679 TO DRAW
                  1150 *
                  1160  .OR $932F
                  1170  .TA $800
00FC-             1180 VT .EQ $FC        ** DECIMAL 252
00FD-             1190 VB .EQ $FD        ** DECIMAL 253
00FE-             1200 HR .EQ $FE        ** DECIMAL 254
00FF-             1210 HL .EQ $FF        ** DECIMAL 255
0026-             1220 HBASL .EQ $26     ** DECIMAL 38  (SCREEN BASE
0027-             1230 HBASH .EQ $27     ** DECIMAL 39  ADDRESS)
0006-             1240 YO .EQ $6         ** DECIMAL 6
00FA-             1250 BASL .EQ $FA      ** DECIMAL 250 (TABLE BASE
00FB-             1260 BASH .EQ $FB      ** DECIMAL 252 ADDRESS)
9391-             1270 YADDR .EQ $9391   ** DECIMAL 37777 (READ YTABLE)
932F- A9 00       1280 DRAW LDA #0       ** CALL 37679 TO ENTER
9331- 85 FA       1290  STA BASL         ** POINT TO START OF TABLE
9333- A5 FD       1300  LDA VB           ** GET BOTTOM Y-COORDINATE
9335- 85 06       1310  STA YO           ** STORE IN $6 FOR USE BY YADDR
9337- 20 91 93    1320 L1A JSR YADDR     ** RETURNS-LO=HBASL/HI=HBASH
933A- A4 FE       1330  LDY HR           ** SET Y-REG TO RIGHTMOST BYTE
933C- A2 00       1340  LDX #0           ** SET TABLE OFFSET=0
933E- A1 FA       1350 L2A LDA (BASL,X)  ** GET SHAPE BYTE FROM TABLE
9340- 51 26       1360  EOR (HBASL),Y    ** MODIFY BYTE TO BACKGROUND
9342- 91 26       1370  STA (HBASL),Y    ** LOAD SHAPE BYTE ON SCREEN
9344- 88          1380  DEY              ** POINT TO NEXT SCREEN ADDRESS
9345- 18          1390  CLC
9346- E6 FA       1400  INC BASL         ** POINT TO NEXT TABLE ELEMENT
9348- D0 02       1410  BNE NC2          ** IF <256 BYTES JUMP
934A- E6 FB       1420  INC BASH         ** PAGE OVERFLOW-GOTO NEXT PAGE
934C- C0 FF       1430 NC2 CPY #$FF      ** HAS Y-REGISTER REACHED 0 ?
934E- F0 04       1440  BEQ NXTLN2       ** YES-GOTO NEXT LINE
9350- C4 FF       1450  CPY HL           ** IS Y-REGISTER >=HL ?
9352- B0 EA       1460  BCS L2A          ** YES-JUMP TO LOOP2A
9354- C6 06       1470 NXTLN2 DEC YO     ** MOVE UP TO NEXT LINE
9356- A5 06       1480  LDA YO           ** GET NEW Y-COORDINATE
9358- C9 FF       1490  CMP #$FF         ** HAS Y-COORDINATE REACHED 0 ?
935A- F0 04       1500  BEQ RTN2         ** YES-WE'RE FINISHED
935C- C5 FC       1510  CMP VT           ** HAVE WE REACHED VT YET ?
935E- B0 D7       1520  BCS L1A          ** NO-START THE NEXT LINE
9360- 60          1530 RTN2 RTS          ** DONE-EXIT ROUTINE
                  1540 * END OF ROUTINE
```

```
        KEY PERFECT 4.0
            RUN ON
    BLOCK ROUTINES $92E6
=====================================
    CODE       ADDR# - ADDR#
    ------     -------------
    25BB       09C7 - 0A16
    24E7       0A17 - 0A66
    2A84       0A67 - 0AB6
    2BA9       0AB7 - 0B06
    3226       0B07 - 0B56
    21ED       0B57 - 0BA6
    275E       0BA7 - 0BF6
    2DC3       0BF7 - 0C46
    290A       0C47 - 0C96
    2627       0C97 - 0CE0
TOTAL PROGRAM CHECK IS : 031A
```

```
   APPLE CHECKER

ON: BLOCK ROUTINES $92E6
TYPE: B

LENGTH: 031A
CHECKSUM: EE
```

At this point you'll have everything published (so far) in memory. To save all the routines in a single disk file, enter **BSAVE (New name), A$ (Hex address of newest routine), L(38400-CALL address of newest routine).**

That's all for this part of our discussion. By now you should have the necessary tools to create block shapes of almost any graphics you'd like to draw, or which might be hanging around on your disks. In the next installment, we'll expand on how to animate your shapes, but for now why don't you simply experiment with some of your own animation. Just bear in mind that once your BLOCK SHAPE is created and in a table, **all you have to do to move the block shape around the screen is to change the values of VT and VB to move up and down, or HL and HR to move left and right, and then CALL the DRAW routine. Have fun!!!!**

LISTING 5

```
10  REM
20  REM    SAMPLE ONE PACE ANIMATION
30  REM
40  REM     GRAPHICS WORKSHOP II
50  REM     BY ROBERT R DEVINE
60  REM
70  REM     COPYRIGHT (C) 1983
80  REM     BY MICROSPARC INC.
90  REM     ALL RIGHTS RESERVED
100  REM
110  HGR2 : CALL 37799
120  POKE 251,144
130 HR = 2:HL = 0
140  POKE 254,HR: POKE 255,HL
150  FOR VT = 0 TO 170 STEP 2
160  POKE 252,VT: POKE 253,VT + 11
170  CALL 37679: FOR X = 1 TO 25: NEXT
     : CALL 37679
180  NEXT
190  FOR VT - 170 TO 0 STEP  = 2
200  POKE 252,VT: POKE 253,VT + 11
210  CALL 37679: CALL 37679
220  NEXT
230 HR = HR + 2:HL = HL + 2: IF HR > 39
      THEN 130
240  GOTO 140
```