The Graphics Magician Programming Tutorial

by Mark Pelczarski

This manual contains information about a few new features added to the original Graphics Magician package along with short program examples that show how various options available through the Graphics Magician can be controlled from your own software. We weren't kidding when we said this was a powerful package. Several arcade and adventure games are already commercially available that use the routines from Graphics Magician for animation and picture retrieval. The regular manual tells how to use the editors in the Magician package and outlines all the options available for programmers. This manual will show some specific examples of how programmers have used the routines in their own software, and should be of help to you when you are writing your programs. You should use this in conjunction with your regular manual.

New options

How do you tell which version you have?

With the newer versions, we started putting a number in the lower left corner of the menu screen (the one you see when you boot the Magician disk). The current version at the printing of this tutorial is V5.82. If no number appears there, you have an earlier version. The number corresponds to the month and year of the revision, you'll be able to identify future revisions by date. If we have your registration card, you will receive notices of further updates if they occur. We will update to any new revision if you send your original disk, registration number (see the inside front cover of your manual), and $5. If the update cannot be shipped UPS surface, you must additionaly include enough to cover shipping expense.

Faster PICDRAW

An extra version of the PICDRAW subroutine, PICDRAWF, has been added. PICDRAWF redraws pictures about 20% faster than the original PICDRAW. The tradeoff is RAM space, as PICDRAWF takes 512 more bytes. To put PICDRAWF on your disk, use the binary transfer utility. The new starting address in RAM is $8C00, or 35840 decimal. All POKEs, CALLs, and JSRs are exactly the same, however.

Disk Access

A new option 'C' has been added to the master menu that allows you to change the disk drive specifications for the master and data disks. You will be asked for the location of each. If you have a one drive system, use 'D1' for both. If you have a two drive system with one controller card (the most common set-up for two drives), use 'D1' for the master and 'D2' for the data disk. You may also use the S and V specifications for slot and volume. If you are using two controller cards, you may use 'S6' and 'S5', for example. Certain hard disk systems use specifications like 'D15' or 'V23'. Check with your dealer if you are unsure of what to use with your particular system.

Collision Detect Subroutine

A machine language subroutine, COLLISION DETECT, has been added to the disk. Its use is described in the tutorial.

Shape Starter

For those who want to convert existing shapes from Applesoft shape tables to pre-shifted shapes for animation, a SHAPE START routine has been added. To use it, RUN SHAPE START, then insert the disk with your shape table and type in your shape table's name. You will then be asked for the number of the shape you wish to convert, and the size of the border you wish to leave around it. A border line will be put on the screen, and you will have to position the upper left shape flush against the border using the IJKM keys for direction. When the shape is positioned properly, insert the Graphics Magician disk and press 'S'. The Shape Editor program will then be run, with your shape positioned properly on the screen. You may modify it, or compile it immedately for saving. Pressing 'N' in the SHAPE START program allows you to load in a new shape before running the Shape Editor.

Saving in .PIC format

An extra option has been added to the Picture Editor that lets you save any screen in standard screen format (.PIC) instead of the sequential format. This is handy if you want to use pictures from the Picture Editor for backgrounds in the animation system.

The Animation Routines

Most questions we receive concern the various options in the animation part of the system. A lot of options were built into the animator that even we didn't have definite uses for when created. The attitude, when written, was that "Gee, it would be neat to let the programmer access this...", or that, or whatever. The manual reflects that, basically saying "Here are the options that you can access." Well, here are some specific examples of using those options.

Path Tables and Controlled Paths

Page 19 of the regular manual discusses how a path can technically be modified for each movement. Here we'll go through some specific examples. To thoroughly control the path, you'll have to learn some binary. Each byte in a path gives the information for one move. The diagram on page 19 shows that two bits are used for each direction, Y- (up), Y+ (down), X- (left), and X+ (right). For now, we'll stick with 2 unit moves, since they preserve color. You can ac-

tually move 0-3 dots in any direction (or combination), but we'll leave that to binary specialists.

The number 2 in binary is 10. To move in units of 2, you need to put the binary number 10 in one or more of the pairs or bits that specify direction. For example, if you want to move up two units, you'd put 10 in the leftmost pair, giving 10 00 00 00, which translated from binary gives the number 128. To move down two units, you'd use 00 10 00 00, or 32 in decimal. Left two units would be 00 00 10 00, or 8, and right two units would be 00 00 00 10, or 2.

Now the trick is to locate where the path should be in memory. Create a path on your data disk called ONE MOVE. You should create that path with the Path Editor by making one move in any direction, then saving the path. Whichever objects you want to control in your program, you should assign a path of ONE MOVE from the Animation Editor, followed by a repeat (255). Remember the object numbers.

The Path Lists tell where in memory an object's paths are located (p.18). A path list can contain up to 3 paths, but most of the time they'll only contain one and a repeat command. The path list for each object is 7 bytes long, and the path list table starts at decimal location 38144. Look at listing #1, Joystick Animation. Line 40 has the formula for finding the location, P, of an object's path, where N is the object number. During animation, if you POKE that location with a new direction value, that object will start moving in the new direction.

Listing #1 is an example of controlling an object's moves with a joystick. The animation file JOYSTICK.ANM contains one object with a path of ONE MOVE (we used block animation). Line 10 loads the animation file, and line 20 sets full screen page one graphics. Line 30 sets N, the object number, to 0 (since we had only one object). It can actually be any object you want. Line 40 finds the location of object N's path. The animation loop starts at line 50 with both paddles (or joystick readings) being read.

The path computation is in line 60. Notice that the four multiplying factors are 128, 32, 8, and 2, which correspond to the movement numbers for up, down, left, and right. Looking at the first part of the equation, (Y<80) gives a value of one if true, zero if false. That means that if the joystick Y setting is less than 80 (of 0-255), an upward movement will register (1*128). Likewise, if the joystick Y setting is greater than 160, a downward move will register. Between 80 and 160, no up-down movement registers. Following through the equation, similar computations are made for left-right movements. Finally, if the result is 0 (no movement), it is changed to 255 (also no movement). Why? Zero designates the end of a path.

Line 70 POKEs the new path value into its location and CALLs the animator. Line 80 loops back.

```
5  REM  LISTING #1
10  PRINT  CHR$ (4);"BLOAD JOYSTICK.ANM"
20  HGR : POKE  - 16302,0
30 N = 0
40 P =  PEEK (38144 + 7 * N) * 256 +  PEEK (38145 + 7 * N)
50 X =  PDL (0):Y =  PDL (1)
60 M = (Y < 80) * 128 + (Y > 160) * 32 + (X < 80) * 8 + (X > 160) * 2: IF
     M = 0 THEN M = 255
70  POKE P,M: CALL 36928
80  GOTO 50
```

Alternately, listing #2 gives a similar example with keyboard instead of joystick control. Using the same animation file, line 50 checks for a keypress. If there was none, the program jumps to line 100 and loops. The keys affecting movement (lines 60-80) are space (stop movement; path = no move), 'A' (up), 'Z' (down), left arrow (left), and right arrow (right). For each the variable M is assigned the proper path value and then M is POKEd into the path. One note is the HIMEM statement in the beginning that sets the upper limit for string variable use. Since the program uses a string variable (A$) we want to make sure the values don't interfere with the animation routine. If you set HIMEM less than the bottom address listed in the animation editor, you won't have any problem.

```
1  REM  LISTING #2
5  HIMEM: 30000
10  PRINT  CHR$ (4);"BLOAD JOYSTICK.ANM"
20  HGR : POKE  - 16302,0
30 N = 0
40 P =  PEEK (38144 + 7 * N) * 256 +  PEEK (38145 + 7 * N)
45  POKE P,255
50  IF  PEEK ( - 16384) < 128 THEN 100
60  GET A$: IF A$ = " " THEN M = 255: GOTO 95
65  IF A$ = "A" THEN M = 128: GOTO 95
70  IF A$ = "Z" THEN M = 32: GOTO 95
75  IF A$ =  CHR$ (8) THEN M = 8: GOTO 95
80  IF A$ =  CHR$ (21) THEN M = 2
95  POKE P,M
100  CALL 36928: GOTO 50
```

Collisions

Listing #3 shows an example of using the collision table to tell when two objects collide. 'COLLIDE.ANM' for our example has two objects. Object 0 has a path of ONE MOVE and will be controlled with the joystick. Object 1 has a path, any path, that will move it around the screen by itself. We used XDRAW animation, although you can also use 'Draw with Background Save'.

The program is very similar to the joystick animation example, except that line 45 does the initial draw for the animation (both XDRAW and DRAW WITH BACKGROUND require this initial CALL), and lines 80 and 90 check for collisions. Note that the collision table starts at location 38368, so the collision value for object 0 is in location 38368, for object 1 is at 38369, and so on. Line 80 prints the actual collision values on the screen. Line 90 stops the program if either value is non-zero, until a key is pressed. Note when running this example that sequence determines whether one or both of the collision counters are set. One object may move with no collision, and then the other may move on top of the first, causing a collision flag on the second.

```
1  REM  LISTING #3
5  HIMEM: 30000
10  PRINT  CHR$ (4);"BLOAD COLLIDE.ANM"
```

```
20  HGR
30 N = 0
40 P =  PEEK (38144 + 7 * N) * 256 +  PEEK (38145 + 7 * N)
45  CALL 37284
50 X =  PDL (0):Y =  PDL (1)
60 M = (Y < 80) * 128 + (Y > 160) * 32 + (X < 80) * 8 + (X > 160) * 2: IF
     M = 0 THEN M = 255
70  POKE P,M: CALL 36928
80  PRINT  PEEK (38368), PEEK (38369)
90  IF  PEEK (38368) OR  PEEK (38369) THEN  GET A$
100  GOTO 50
```

The fourth example shows how you can detect collisions with the background. This can be used with BLOCK WITH BACKGROUND, XDRAW, or DRAW WITH BACKGROUND SAVE. We use BLOCK WITH BACKGROUND for the example, with one object with path ONE MOVE, controlled by joystick. New are the two lines that load the background picture on page 1, then on page 2 (XDRAW and DRAW WITH BACKGROUND only need load the picture on page 1). For a picture, we created a maze with the Picture Editor and saved it in .PIC format (standard screen image). You can use whatever background you want. Lines 80 and 90 print the collision value for your object and pause if a collision is detected.

```
1  REM  LISTING #4
5  HIMEM: 30000
10  PRINT  CHR$ (4);"BLOAD BLOCKBACK.ANM"
20  HGR
22  PRINT  CHR$ (4);"BLOAD MAZE.PIC,A8192"
24  PRINT  CHR$ (4);"BLOAD MAZE.PIC,A16384"
30 N = 0
40 P =  PEEK (38144 + 7 * N) * 256 +  PEEK (38145 + 7 * N)
50 X =  PDL (0):Y =  PDL (1)
60 M = (Y < 80) * 128 + (Y > 160) * 32 + (X < 80) * 8 + (X > 160) * 2: IF
     M = 0 THEN M = 255
70  POKE P,M: CALL 36928
80  PRINT  PEEK (38368)
90  IF  PEEK (38368) THEN  GET A$
100  GOTO 50
```

Using the Object List

This example uses an animation file that is already on your disk, LETTERS.ANM. This animation file already has each of the animated alphabet letters, stored as shapes 0-25. It also has a blank character as shape number 26, and each object is assigned a location along a single rectangular path. The x,y location and path location were figured before running the animation editor by placing each object at a specific location on the screen and counting which step along the path would put the object there. That way, the letters in the end result will move in single file, since each is slightly ahead of the next in the path.

Worth noting here is that with all the letters, and all your shapes, the first definition of the shape (upper left shape) will appear when the object is in columns that are multiples of sevens. Therefore, to get a non-distorted letter, it

should be put in column 0, 7, 14, 21, and so on. The objects in the rectangular path are arranged so that the vertical movements will be on such columns, so the letters will appear normal. Of course, if your shape does not animate within itself, the column doesn't matter.

Listing #5 shows how the object list is used to specify and change which shapes are animating. Lines 5-20 initialize the routines, lines 30 and 40 allow you to enter a group of letters, and line 50 sets the hi-res graphics mode. The lines from 60 to 90 then POKE in the proper shapes to the path list. This is done by finding the ASCII value of each character (line 65), and subtracting 65. Since the letter 'A' is ASCII 65, the result for 'A' would be 0, corresponding to its shape number. 'B' would result in a 1, 'C' in a 2, etc. That number is then POKEd into next position in the object list. The object list starts at 37888, and contains one byte for each object, giving the shape number of that object. After the string is decoded, line 90 fills the rest of the objects with blank shapes (shape #26). Line 100 does all the animation by CALLing the routine and looping back until a key is pressed. Note that BLOCK WITH BACKGROUND was used, so if you want to use a background you can load it into page 1 and page 2 and it will be preserved.

```
1  REM  LISTING #5
5  HIMEM: 26139
10  PRINT  CHR$ (4);"BLOAD LETTERS.ANM"
15  HGR2
20  TEXT : HOME
30  PRINT "TYPE SOMETHING 28 CHARACTERS OR LESS.": INPUT A$
40  IF  LEN (A$) > 28 THEN 30
50  HGR : POKE  - 16302,0
60  FOR I = 1 TO  LEN (A$)
65 A =  ASC ( MID$ (A$,I,1)) - 65
70  IF A < 0 OR A > 25 THEN  POKE 37888 + I - 1,26: GOTO 80
75  POKE 37888 + I - 1,A
80  NEXT I
90  FOR J = I TO 28: POKE 37888 + J - 1,26: NEXT : POKE 37916,255
100  CALL 36928: IF  PEEK ( - 16384) < 127 THEN 100
110  GET A$: GOTO 20
```

Deactivating Objects

You can also use the Object List for activating and deactivating objects. Listing #6 is a crude game of sorts, in which you control one object with a joystick while five others move around randomly until you collide with them, at which time they freeze. First, create an animation file with the animation editor that has six objects with shapes of your own choosing. Use XDRAW animation, since we'll want to tell when objects collide with each other by looking at the collision table. The first object (#0) will be the one you control. Load the path ONE MOVE six times, as you'll need it in six separate locations. Assign path 0 to object 0, path 1 to object 1, all the way to path 5 to object 5. Save the animation file as 'COLLGAME'.

The program sets HIMEM (check to see that it's low enough for the file you created), then loads the animation routine and a COLLISION DETECT subroutine that we added to the disk. The collision detect routine is very short (15 bytes), and sits in the low part of memory at address 768 (hex $300). When you CALL the routine at 769, it will quickly search the collision table until it finds a non-zero value, put the object number that collided in location 768, and return. If there

was no collision, the value 255 is put in location 768. As is, the routine searches through all 32 objects for a collision, in reverse order (from 31 to 0). If you have fewer objects, you can POKE the number of the last object in location 770. In our case, with 6 objects, 0-5, we'll POKE 770,5. See line 50.

To make the other objects move at random, first we'll assign the movement values to a direction array, D, in line 25. Then we set up an array, P, that holds the path locations of each of the six objects' paths (lines 30-45). Line 50 does the initial draw for the XDRAW routine and tells the collision detector how many objects to check. Then lines 60-75 assign random directions to each of objects 1-5. The stuff in the parentheses after D in line 70 selects a random number from 1 to 4.

The set of lines from 80 to 150 repeat 15 times before each of the other objects are assigned new random directions. You should recognize lines 90-110 from the joystick control example.

The last set of lines that need explaining are 120-140. Line 120 CALLs the collision detector, puts the object number of the collision in C, and checks if there is no collision (C = 255) or if 1-5 didn't collide, but you (C = 0, your object) did. (It is possible for only one of two colliding objects to register a collision. It depends if one moved last, on top of one that had already moved and occupied a point.) If there was no collision, the loop continues at line 150. If there was a collision, the object is deactivated by POKEing 254 in the object list (line 130), and the collision flag for that object is set back to zero (line 135). In case there was more than one collision, the program then jumps back up to line 120 to check the collision table again.

```
1  REM  LISTING #6
5  HIMEM: 30000
10  PRINT  CHR$ (4);"BLOAD COLLGAME.ANM"
15  PRINT  CHR$ (4);"BLOAD COLLISION DETECT"
20  HGR : POKE  - 16302,0
25 D(0) = 128:D(1) = 32:D(2) = 8:D(3) = 2
30  FOR N = 0 TO 5
40 P(N) =  PEEK (38144 + 7 * N) * 256 +  PEEK (38145 + 7 * N)
45  NEXT N
50  CALL 37284: POKE 770,5
60  FOR I = 1 TO 5
70  POKE P(I),D( INT ( RND (1) * 4))
75  NEXT I
80  FOR I = 1 TO 15
90 X =  PDL (0):Y =  PDL (1)
100 M = (Y < 80) * 128 + (Y > 160) * 32 + (X < 80) * 8 + (X > 160) * 2: IF
     M = 0 THEN M = 255
110  POKE P(0),M: CALL 36928
120  CALL 769:C =  PEEK (768): IF C = 0 OR C = 255 THEN 150
130  POKE 37888 + C,254
135  POKE 38368 + C,0
140  GOTO 120
150  NEXT I
160  GOTO 60
```

Switching Shapes Midstream

You can also use the object list to change the shape of an object in middle of animation.The only caution is to have the shapes that you are switching the same size. If they are different sizes (specifically, if the new one is smaller), you may leave a trail. Listing #7 has a short example using the LETTERS.ANM file. After each fifty moves, the letter is changed by POKEing the next shape number into the object list. Note that all objects after #0 are deactivated by POKEing object #1 with a 255.

```
1  REM   LISTING #7
5  HIMEM: 26139
10  PRINT  CHR$ (4);"BLOAD LETTERS.ANM"
15  HGR2
20  POKE 37889,255
50  HGR : POKE  - 16302,0
60  FOR L = 0 TO 25
70  POKE 37888,L
80  FOR I = 1 TO 50
90  CALL 36928
100  NEXT I: NEXT L
110  GOTO 60
```

A Word about HI,LO and LO,HI Address Formats

It takes two bytes to save an address in your computer. There are various path and shape pointers throughout the animation tables, and they have their addresses in one of two formats, HI,LO or LO,HI. LO,HI is the usual method used in machine language, but in a few cases we were able to gain some speed in reversing the format. To convert a decimal address to either format, you need to divide by 256 and save the integer result in the HI byte, and the remainder in the LO byte. A quick formula for computing these is:

HI = INT(A/256)
LO = A - HI*256
where A is the address to be split.

If an address is to be saved in HI,LO format, use POKE L,HI:POKE L + 1,LO, where L is the first of the two address locations. Reverse HI and LO for LO,HI format.

Plotting and not Animating

If you want to plot a shape at a location and leave it there instead of animating it, you can directly CALL the plot subroutine explained on pages 19 and 20 of the regular manual. You need to POKE in the x location, y location, and shape number before CALLing the routine. With DRAW WITH BACKGROUND SAVE, you need to also specify a buffer number for the background. We use the object number, but you can use any value from 0 to 31, as long as you don't

interfere with one of your currently animating objects. If X and Y are the screen locations for plotting, and S is the shape number, use the following:

```
POKE   0,INT(X/7)
POKE   1,X-INT(X/7)*7
POKE   2,Y
POKE   3,S
CALL   36608
```

For DRAW WITH BACKGROUND, if B is the buffer number you want used, insert:

```
POKE   4,B
```

To erase the object, except in XDRAW mode, use CALL 36770 after the same POKEs. In XDRAW, use CALL 36608 again.

Finding Object Locations

To find the screen location of any object, use the Object Location table described on page 17 of the regular manual. As an example, if the object number is N, its location is:

```
X = PEEK(38048 + N*3)*7 + PEEK(38049 + N*3)
Y = PEEK(38050 + N*3)
```

If you are using the object locations for collision checking, try to limit the options. If you have a lot of objects and try to check every object against every other one, you'll use a lot of time. Check specifically those objects that can possibly collide, or those that you suspect may collide and your program will run faster.

Using Pictures from the Picture Editor in your Programs

The commands necessary for redrawing a sequential picture created with the picture editor are few. The trick within your program will be memory management. In other words, you'll have to (a) set aside the memory space from $8E00 to $95FF (decimal 36352 to 38399), and (b) set aside a buffer area (a place to load the binary files that contain the picture/object commands) for loaded pictures and objects. Depending on your application and the size of your program, you may want to set aside a single buffer for every picture and object that you load, or you may want to be able to load several pictures and objects at a time. The key to determining the size of your buffer is to use the size, in bytes, of your longest picture or object. That number is displayed as you work with your picture in the picture editor, or, if you forgot the size, you can find the length with the binary transfer utility.

Say, for example, your longest picture/object file is less than 3000 bytes. You could safely set the memory area above $8000 (decimal 32768) as your buffer. PICDRAWF starts at $8C00 (35840), giving you a 3,072 byte buffer. If you need more room, you could either move the starting address of the buffer lower in memory, or use PICDRAW, which starts at $8E00 (decimal 36352) and gives a 3,584 byte buffer.

If 3,072 bytes is long enough, you can use the program in listing #8 to load and redraw a picture. Substitute the actual name of your picture for 'PNAME'.

```
1  REM  LISTING #8
5  HIMEM: 32768
10  PRINT  CHR$ (4);"BLOAD PICDRAWF"
20  PRINT  CHR$ (4);"BLOAD PNAME.SPC,A32768"
30  HGR
35  REM  LINE 40 CONTAINS 32768 PRE-COMPUTED IN LO,HI FORMAT
40  POKE 36352,0: POKE 36353,128
50  CALL 36400
```

Note that the address of the starting location of the buffer is used in the BLOAD statement of the picture, and is POKEd in LO,HI format into locations 36352 and 36353 before the redraw routine is CALLed.

Putting an Object over a Picture

Listing #9 can be added to listing #8, and shows the extra commands needed for putting an object on the picture. Substitute the name of your object for 'ONAME'. Note that the x and y location where the object should be drawn is POKEd into 36354-36356. If part of the object goes off the screen, you may get a bit of a mess, so position it so that it will be within the screen boundaries. Some trial and error is necessary, occasionally. In the POKEs for the x location, (X>255) is 1 if true, 0 if false. Since the division by 256 will never result in a number greater than one (since the screen goes from 0 to 279), this is a shortcut over using division and the INT function.

```
59  REM  LISTING #9
60  PRINT  CHR$ (4);"BLOAD ONAME.SPC,A32768"
70  POKE 36352,0: POKE 36353,128
80 X = 100:Y = 80
90  POKE 36354,X - (X > 255) * 256: POKE 36355,X > 255: POKE 36356,Y
100  CALL 36361
```

PICDRAWF vs. PICDRAW

The only differences between using PICDRAW and PICDRAWF are the speed and amount of memory used. All POKEs and CALLS are identical. If there is no conflict of memory locations between $8C00 and $8DFF with a program, anything designed to use PICDRAW may also use PICDRAWF. Any program designed to use PICDRAWF can also later be changed to use PICDRAW, freeing 512 extra bytes.

BLOADing Groups of Pictures in One File

It's also easy to load many pictures in a single binary file. This was done for the DEMO program, and many more could have fit in that particular example. The first step is to find the exact length of each picture file you want in your combined file. Use the binary transfer utility for that. Suppose you had the results:

NAME	LENGTH
House.SPC	1254
Tree.SPC	879
Rhinoceros.SPC	2318

You first need to choose a starting location. For the most pictures, use the end of the first hi-res page, $4000 (or 16384). This leaves room for 19,456 bytes of pictures for PICDRAWF; 19,968 for PICDRAW.

The next step is to BLOAD each picture sequentially in memory. For the first, you'd use the following from Applesoft (square bracket prompt):

BLOAD HOUSE.SPC,A16384

which loads the first picture at 16384. Then add its length to 16384 for the next address for loading. In our example, 16384 + 1254 = 17638, so we'd follow with:

BLOAD TREE.SPC,A17638

Remember the location that each is loaded at, since that's the number you'll have to POKE into the address locations before CALLing the PICDRAW routine for any individual picture. If you forget the number, you'll have to recompute it, or CALL the PICDRAW routine for each preceding picture first.

The third file would be loaded at 17638 + 879 = 18517, in our example, so we'd use:

BLOAD RHINOCEROS.SPC,A18517

Finally, once all the pictures are BLOADed, compute the total length by adding each individual length together. For us, it would be 1254 + 879 + 2318 = 4451. Then BSAVE the entire file with:

BSAVE PICTURE GROUP,A16384,L4451

substituting the desired name for PICTURE GROUP and the appropriate starting address and length for 16384 and 4451.

Using a Group of Sequential Pictures

Listings #10 and #11 give examples of using the picture group we just saved in a program. Listing #10 cycles through the three pictures in order, waiting for a keypress between changes. Listing #11 uses the addresses we noted to allow you to select which picture is drawn, and when. Pressing 1, 2, or 3 draws the appropriate picture.

Interesting to note in listing #10 is that the starting addresses do not have to be POKEd in when the pictures are shown sequentially. That is because the address locations you POKE the values in is also used as a counter by the machine language PICDRAW subroutine. As it executes each command in your picture, the counter is incremented for the next command. When the end of the picture is reached, the counter is automatically pointing at the next byte! If you used the method described above for saving pictures one right after another, that next byte is the first location of the next picture!

```
1  REM LISTING #10
5  HIMEM: 8192
10  PRINT  CHR$ (4);"BLOAD PICDRAWF"
15  HGR
20  PRINT  CHR$ (4);"BLOAD PICTURE GROUP,A16384"
25  REM LINE 25 CONTAINS 16384 PRE-COMPUTED IN LO,HI FORMAT
30  POKE 36352,0: POKE 36353,64
40  FOR I = 1 TO 3
50  CALL 36400: GET A$
60  NEXT
```

```
1  REM LISTING #11
5  HIMEM: 8192
10  PRINT  CHR$ (4);"BLOAD PICDRAWF"
15  HGR
20  PRINT  CHR$ (4);"BLOAD PICTURE GROUP,A16384"
25  DIM L(3): REM  L CONTAINS THE LOCATIONS OF EACH PICTURE
30  FOR I = 1 TO 3: READ L(I): NEXT : DATA  16384,17638,18517
40  GET A$: IF A$ < "1" OR A$ > "3" THEN 40
45 V = L( VAL (A$))
50  POKE 36352,V -  INT (V / 256) * 256: POKE 36353, INT (V / 256)
55  CALL 36400
60  GOTO 40
```

Using Super Shapes

Listings #12 and #13 show examples of using the super shape subroutine in short and long form. Since the routine is relocatable (you can load it at any free area of memory), the two CALLs are relative to the location at which the routine is loaded.

The only requirement for where you load your table is that the address at which it's loaded must be an exact multiple of 256. When the regular manual refers to the starting page of a table, it's asking for the multiple of 256 at which it was loaded, i.e., the address divided by 256 (it must not have a remainder).

In listing #13, note that the x location is POKEd in exactly the same way that we used for objects with the picture editor. Also note that all parameters only need to be POKEd when they are to be changed. If they are to stay the same, most POKEs can be left out.

```
1  REM  LISTING #12
10  PRINT  CHR$ (4);"BLOAD SST/ML,A16384"
15  HGR
20  PRINT  CHR$ (4);"BLOAD NAME.SST,A24576"
25 SH = 1:PG = 24576 / 256
30  POKE 253,SH: POKE 254,PG
35  CALL 16384
```

```
1  REM  LISTING #13
10  PRINT  CHR$ (4);"BLOAD SST/ML,A16384"
15  HGR
20  PRINT  CHR$ (4);"BLOAD NAME.SST,A24576"
25 SH = 1:PG = 24576 / 256
30  POKE 253,SH: POKE 254,PG
35 RO = 0:CS = 0:D = 1:SO = 0:X = 100:Y = 80
40  POKE 252,RO: POKE 255,CS: POKE 4,D: POKE 5,SO
45  POKE 249,X - (X > 255) * 256: POKE 250,X > 255: POKE 251,Y
50  CALL 16400
```