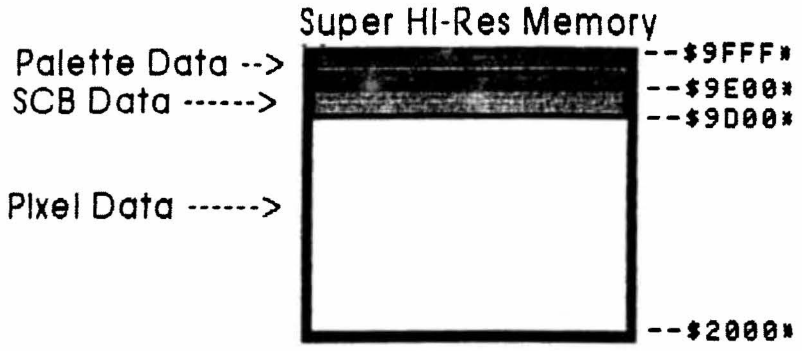
1992 Apple IIGS Graphics and Sound College

Presented by Bill Heineman and Nate Trost

Super Hi-Res Screen Layout

The Apple IIGS Super High-Res Graphics screen operates in two modes of resolution: 320x200 and 640x200. In the 320x200 resolution each pixel is represented by 4 bits (thus there are two pixels per byte), in 640x200 each pixel is represented by 2 bits (4 pixels per byte). The size of the graphic screen is 32,000 bytes and is followed by 200 scan line control bytes (one for each SHR line), 56 bytes of unused space, and 512 bytes of palette information (for a total of 32,768 bytes or 32K). The SHR graphics memory is located in memory bank $E1 from memory locations $2000 to $9FFF.

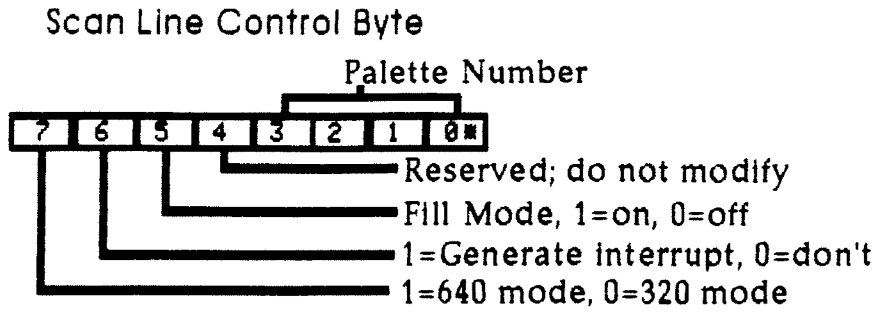


The Super Hi-Res Pixels

The pixel data for the Super Hi-Res screen is arranged in a linear fashion from $2000 to $9CFF in memory bank $E1. Each line on the Super Hi-Res screen is represented by 160 bytes in memory. In 320 mode each of the 320 pixels are 4 bits (totalling 160 bytes), in 640 mode each of 640 pixels are 2 bits (again, totalling 160 bytes.) Line 0 on the SHR screen starts at $2000, Line 1 starts 160 bytes after Line 0, and so on, until Line 199, which ends at $9CFF. Each pixel value designates which color value will be used from the palettes located at $9E00 $9FFF.

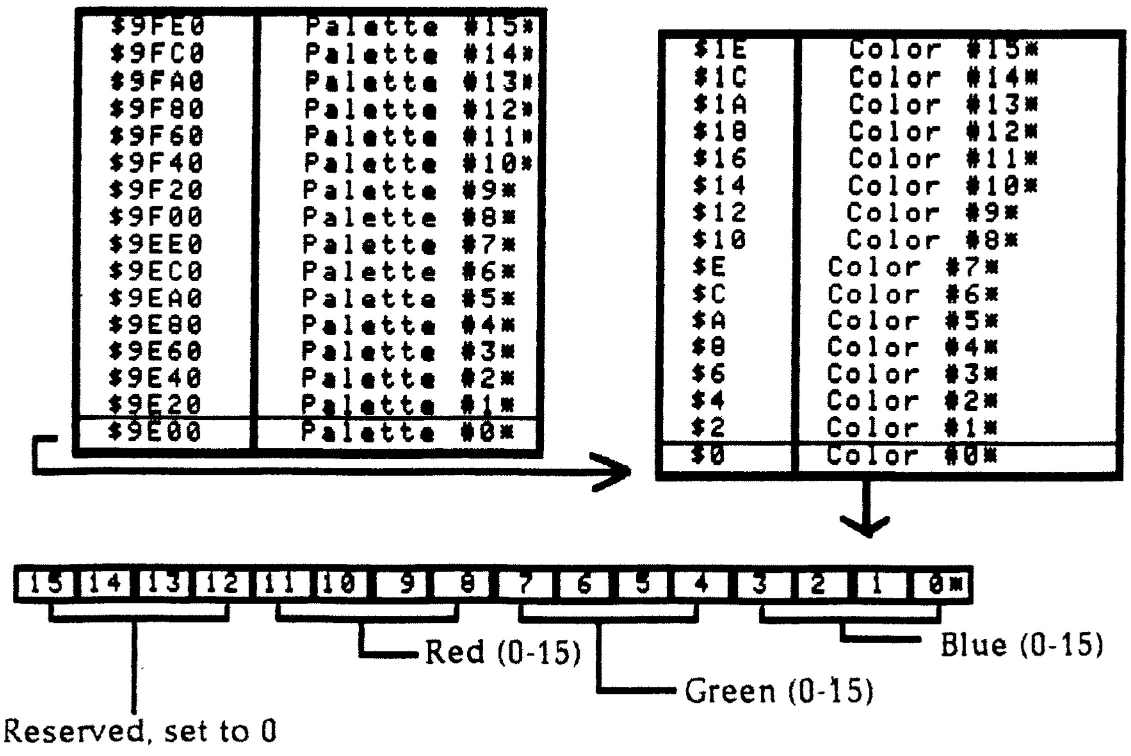
Scan Line Control Bytes (SCBs)

Each of the 200 vertical lines on the Super Hi-Res screen has a single byte allocated to it. These bytes are known as Scan Line Control Bytes, or SCBs. SCBs regulate whether a line is in 320 or 640 mode, whether fill mode is operating for the line (320 mode only), if an interrupt is generated when the line is refreshed, and which of the 16 palettes will be used to represent the colors available to the pixels on the line.



Palettes

There are 16 palettes (also known as color tables) that occupy the 512 byte space from $9E00 to $9FFF. Each palette has 16 entries (of two bytes each) that specify a color. The highest four bits are reserved, and the Red, Green, and Blue values are represented by four bits each. This allows 16 possible values for each of the three primary colors for a total of 4096 possible colors (16x16x16=4096). The value of each pixel is an offset into the palette being used for its line. For example, if a pixel in 320 mode had a value of 4, the color of the pixel would depend on what the fifth color entry in the line's palette was set to (for example, if it was $F00 the pixel would be a bright red, $0F0 bright green, $00F, blue...).



$9FE0	Palette #15*
$9FC0	Palette #14*
$9FA0	Palette #13*
$9F80	Palette #12*
$9F60	Palette #11*
$9F40	Palette #10*
$9F20	Palette #9*
$9F00	Palette #8*
$9EE0	Palette #7*
$9EC0	Palette #6*
$9EA0	Palette #5*
$9E80	Palette #4*
$9E60	Palette #3*
$9E40	Palette #2*
$9E20	Palette #1*
$9E00	Palette #0*

$1E	Color #15*
$1C	Color #14*
$1A	Color #13*
$18	Color #12*
$16	Color #11*
$14	Color #10*
$12	Color #9*
$10	Color #8*
$E	Color #7*
$C	Color #6*
$A	Color #5*
$8	Color #4*
$6	Color #3*
$4	Color #2*
$2	Color #1*
$0	Color #0*

Colors in 640 Mode

The way colors and palettes operate in 640 mode is different from 320 mode. In 640 mode each pixel is represented by only two bits, allowing 4 possible values. In order to access all 16 possible entries in the palette, each of the 4 pixels in a 640 mode pixel byte access a different set of 4 colors in the 16 color palette. The first pixel of the byte uses colors 8-11, the second pixel, colors 12-15, third pixel, colors 0-3 and fourth pixel colors 4-7. For example, pixel 47 on line 20 would be the fourth pixel in its byte. If this pixel has a value of 1, it would use the 5th color entry from the palette assigned to its line.

Bit	7	6	5	4	3	2	1	0*
Pixel (640)	1		2		3		4*	

Pixel	Value	Palette Color	Pixel	Value	Palette Color
3*	0*	$0*	1*	0*	$8*
	1*	$1*		1*	$9*
	2*	$2*		2*	$A*
	3*	$3*		3*	$B*
4*	0*	$4*	2*	0*	$C*
	1*	$5*		1*	$D*
	2*	$6*		2*	$E*
	3*	$7*		3*	$F*

Dithering

When in 640 mode, it is possible to achieve the effect of having 16 true colors. Because of the small width of the pixels, it is possible to alternate pixels of different colors in close proximity in order to fool the eye into perceiving a solid color that is a mix of the two alternating colors. For example, alternating blue and white gives a lighter shade of blue while blue and black give a darker shade.

Fill Mode

Color-Fill mode is available in 320 mode only. By setting bit 5 of a line's SCB, fill mode is enabled for that line. When fill mode is active, a pixel value of 0 automatically takes on the color of the last pixel value ($1-$F) used before it on the right, so that:

```
100400000050060403060000002000000*
```

with fill mode off would become:

```
111444444455566443366666662222222*
```

with fill mode on. Note that the first pixel value of a line must not be 0 (when fill mode is on), or a random color will result.

Turning SHR on and off

Turning on:
call QuickDraw II routine *GrafOn*

```
ldal  $E1C029*
ora   #%11000000*
stal  $E1C029*
```

Turning off:
call QuickDraw II routine *GrafOff*

```
ldal  $E1C029*
and   #%00111111*
stal  $E1C029*
```

Quagmire™ Animation Editor File Format

Preliminary Format Description, revision 2

```
+000  5 bytes   ID
        'BWANI' -- ASCII ID. string (hi-bit clear)
+005  2 bytes   VERSION
        file format version number, hi byte major, low byte minor, e.g. $010D=1.13
+007  1 byte    TYPE
        type of animation data:
                255-2: reserved   1: 640 mode SHR   0: 320 mode SHR
+008  1 byte    COMPRESSION
        reserved
+009  1 byte    PIXELSIZE
        size of each pixel in bits
+010  WORD      FLAGS
        bits 15-2:reserved
        bit 1: if set, all frames same height/width
        bit 0: if set, all frames have mask data following image data
+012  WORD      FRAMES
        number of frames in document
+014  LONG      FIRSTOFF
        offset from beginning of file to first frame record
+018  LONG      TABLEOFF
        offset from beginning of file to table of frame record offsets
+022  LONG      COLORTABLESIZE
        size of color table in bytes
+026  See Above  COLORTABLE  color table for document (for SHR, 32 byte in palette format)
FRAME RECORDS
+000  LONG      NEXTOFF
        offset from beginning of frame record to next frame record, nil if last
        frame
+004  WORD      XSIZE
        XSize in words (number of pixels across/4)
+006  WORD      YSIZE
        YSize in pixels (height)
+008  WORD      FLAG
        Bits 15-1: reserved   Bit 0: mask data included after image data
+00A  LONG      FRAMESIZEC
        If frame compressed, size of compressed frame data
+00E  LONG      FRAMESIZE
        Size of uncompressed frame data
+012  8bytes  USER
        8 byte reserved for whatever
+01A  FRAMEDATA...
```

There is a table of frame offsets located in the document...an offset to this table is located in the frame header, the table is made up of LONG values for each frame starting with the first...these values contain the offset from the beginning of the file to the beginning of the frame's record.

Drawing

The act of drawing an object to the screen is actually a very simple matter. All that drawing involves is the copying of pixel data from its buffer to the Super Hi-Res screen memory. Before we can draw the shape, we have to know where to draw it.

Calculating SHR Addresses

We need a way to calculate a starting address from a pair of X/Y screen coordinates so we know where to start drawing. Here is the formula:

320 Mode/640 Mode w/ dithering: (Y * #$A0) + (X / $2) + #$2000 = starting address in SHR memory
640 Mode : (Y * #$A0) + (X / $4) + #$2000 = starting address in SHR memory

* NOTES, each scan line is $A0 (160 bytes) across and there are 2 pixels per byte when in 320 mode or 640 w/ dithering and 4 pixels per byte when in 640 mode proper

Now can we can figure the address, let's look at the work involved in drawing this simple shape:

```
0000FF0000
000FFFF000
00FFFFFF00
00F0770F00
00F7007F00
000FFFF000
0000FF0000
```

This shape is 7 pixels high and 5 bytes across (we'll assume 320 mode which makes the width 10 pixels across) for a total size of 35 bytes. Unfortunately, we just can't loop and copy 35 bytes to the SHR screen at our starting address. Why not? Because each line of our image is only 5 bytes across, each SHR line is 160 bytes across, we would end up with a long horizontal line that looks nothing like our image. What is the solution? We have to draw each line of the image to successive lines in SHR memory. This means we copy the 5 bytes to SHR memory on the starting line, then we add 160 bytes to our SHR address to point it to the next line, and we add 5 bytes to our pointer to our shape and then copy the next 5 bytes...and so on and so forth until we have completed the job. That's all there is to drawing!

Masking*

While the method we described above works, it has one noticable limitation...there is no way to screen out unwanted pixels in the source image, everything is copied. Also, there is no way to make parts of the image 'see-through', any existing 'background' pixel data is overwritten. In order to screen out unwanted pixels and allow the background to be preserved where these pixels are located, we need to use a technique called masking.
In order to know which pixels to draw and which to ignore, we need a seperate pixel image called a mask. In our mask, $F pixels represent areas where we won't copy a pixel and will preserve the background pixel and $0 pixels represent areas where we will copy the pixel. Our mask for our image would look like this:

```
IMAGE          MASK

0000FF0000     FFFF00FFFF
000FFFF000     FFF0000FFF
00FFFFFF00     FF000000FF
00F0770F00     FF0F00F0FF
00F7007F00     FF00FF00FF
000FFFF000     FFF0000FFF
0000FF0000     FFFF00FFFF
```

In order to draw this image using the mask, we:
1 **LOAD** existing pixel data in SHR memory

2 **AND** this data with the corresponding mask data

3 **ORA** the result with the corresponding image data

4 **STORE** the result back to SHR memory

Here is what the process looks like going through drawing a line:

```
LOAD existing data:   7788777444
AND with mask:        FFFF00FFFF
RESULT================7788007444
ORA with image:       0000FF0000
RESULT================7788FF7444
STORE back to memory!
```

'Erasing

The other critical step in animating in image is ERASING. The shape that was drawn, in most cases, must be erased before the next frame is drawn. Erasing can mean either simply zeroing (or another value) over the drawn image on the SHR screen (usually done only if there is no background that needs to be preserved), or restoring the background data that was overwritten when the shape was drawn. Writing a routine to zero a block of memory is easy, restoring the background is slightly more complex.

There are two basic ways to restore the background. The first is to copy the background data before it is overwritten WHILE IN THE DRAWING loop. The data is copied to a buffer of identical size of the shape being drawn. To restore the background, the data is simply copied back to the screen from the buffer at the exact location the drawing took place. The second method involves having another 32K buffer that acts as a background save buffer, where a clean copy of the background is always kept. Data is copied from the background save buffer to the SHR screen to restore the data overwritten by the image.

Both techniques have advantages and disadvantages. The Background Save Buffer (BSB) method is faster in most cases, but needs another 32K, and can be awkward with anything other than a static, unchanging, background. The drawing buffer approach is slower because of the time needed to copy the data into the buffer (this is not needed with the BSB), but it a bit more flexible about non-static backgrounds.

Order

Depending on the drawing/erasing method being used, the order in which shapes are drawn and erased can be important. If you are using the Background Save Buffer to restore the background, it doesn't make any difference what order you draw/erase the shapes.

However, if you are using the local buffer method, you MUST erase the shapes in the opposite order you drew them, e.g.: Draw Circle, Draw Square, Draw Triangle....Erase Triangle, Erase Square, Erase Circle.

The reason for this reverse erasure order is that each of the shapes has their own buffer with their own copy of the background. If the square and the circle overlap, the square will copy the background to its buffer...which will contain part of the circle image. The circle contains a perfectly clean copy since it was drawn first. If the circle were erased first and then the square, the square's buffer would bring back some the the circle image that should have been erased.

Shadowing

In order to avoid the annoying effect of flicker when animating (especially with large and compex objects), a technique called shadowing can be brought to use. By clearing bit 3 of the SHADOW register at $C035, shadowing for the SHR memory is enabled. What this means is that the memory at $E1/2000-9FFF now has a counterpart at $01/2000-9FFF. When shadowing is on, any data written to $01/2000-9FFF is automatically copied to $E1/2000-9FFF.

The procedure of drawing using the shadowed screen is as follows:

ERASE (if needed) previous shape on $01 screen (with shadowing off)

DRAW new shape on $01 screen (shadowing off)

TURN on shadowing and copy affected data on $01 screen back onto self

By using this process (drawing and erasing with shadowing off), it is impossible to see the shape when half-drawn or half-erased. When drawn/erased to the bank $01 screen with shadowing off, the data is not copied to the $e1 screen and therefore does not appear on the display. By turning shadowing on and recopying (loading and storing back onto itself) the data, you can quickly display the changes all at once.

Stack Updates

The fastest way to update the SHR screen is to map the stack and direct page onto the bank $01 screen and use an instruction called PEI. By setting bits 5 and 4 in the State register at $C068, the stack and direct page are mapped to bank $01 instead of bank $00. PEI takes the two byte value found at a pair of direct page locations and pushes them on the stack, e.g. PEI $10 would take the value of direct page locations $10 and $11 and push them on the stack. By setting up the stack and having a long list of PEI's with the direct page set accordingly, it is possible to push data onto itself in only 7 cycles per word (6 if the direct page is aligned on a page boundary). **NOTE**, interrupts **MUST** be disabled when the stack and direct page are mapped to bank $01!

PaintWorks Animations

PaintWorks animation files (filetype $C2) provide a simple method for saving full-screen animations. The file format (never officially specified) is as follows:

```
+$0000-$7FFF - 32K screen image of
the first frame

+$8000-$8003 - Length of animation
data block

+$8004-$8007 - Delay time per frame
in 60th/second ticks

+$8008-EOF     - Animation data block
```

The animation data block is made up of records that tell us how to modify each screen image for the next frame. There is a 4 byte value at the beginning that is an offset to the starting records, however, some animations do not have a valid value in this block so you should just skip over it to get to the first record. Each record four bytes, the first two bytes is an offset into a display screen, addihg $2000 gives the address in SHR memory. The second word is the pixel data to store at the address specified. If the offset into SHR memory is zero, it means that this is the end of the frame.