Technical Note QD515
Color QuickDraw Q&As

CONTENTS

This Technical Note contains a collection of archived Q&As relating to a specific topic - questions sent the Developer Support Center (DSC) along with answers from the DSC engineers. Current Q&As can be found on the Macintosh Technical Q&As web site.

[Oct 01 1990]

Color QuickDraw not available to 68000 Macintosh models

Date Written: 2/4/93

Last reviewed: 3/17/93

Inside Macintosh Volume VI says Color QuickDraw is built into System 7, but when I use `Gestalt` to check for Color QuickDraw on a Macintosh PowerBook 100 running System 7, the call returns 0 in `QDversion`. What's wrong?

The Color QuickDraw routines were written in 68020 assembly, meaning that only Macintosh models with 68020 and later CPUs have access to Color QuickDraw. Models such as the Macintosh SE, Macintosh Classic, and PowerBook 100 can use only Classic QuickDraw. However, this isn't such a bad thing because the processing power required to run Color QuickDraw would be an enormous burden on the 68000 machines.

The bulk of Color QuickDraw routines reside in the larger ROMs of models using the 68020 and greater CPUs, thus taking the burden away from system software. However, if changes have to be made to Color QuickDraw (usually in the form of patches) they reside in system software.

Back to top

Check 'cicn' ID if menu icons are tiny

Date Written: 2/16/93

Last reviewed: 7/2/93

The icons that appear in our application's menu lists are very, very small. It looks like they're 8 x 8 (scaled) instead of the standard 16 x 16. Can you tell from our test code why this is happening?

This was a wild one! What's causing the weird behavior is that you have a `'cicn'` resource with ID = 256 that's smaller than 32 x 32. When the MDEF finds such an icon, it uses it to size the area to use for the menu icons. So, for your application the solution is either to change the 'cicn' to be larger (32 x 32) or to give it a different ID.

Back to top

Color QuickDraw and large pixmaps

Date Written: 3/11/93

Last reviewed: 6/24/93

Somewhere in QuickDraw, a routine is using the 14th bit of the rowBytes field (the 15th is used to differentiate a pixmap from a bitmap) and it really shouldn't. This is a problem now that it's so easy to create large pixmaps of 4096 32-bit pixels or more. What's causing the problem and what's the recommended solution for handling large pixmaps?

This is a long story. It's described in *Inside Macintosh* Volumes V and VI but we'll summarize it here.

Color QuickDraw allows you to pass one of three addresses to CopyBits: the address of a bitmap, the address of a pixmap, or the address of the pixmap handle of a port. The first two conditions (bit and pixmap) are determined by bit 15 of rowBytes: if the bit isn't set, the address points to a bitmap; otherwise, it points to a pixmap. The third case is the problem here. A color port's portVersion field is expected to be set to 0xC000, so after deciding it's dealing with a possible pixmap CopyBits checks bit 14 and if it's set it assumes it got the address of the pixmap handle in a CGrafPort and proceeds to dereference it in order to get the real address of the pixmap.

Today's applications have trouble with this because they must either refuse to create pixmaps as big as users want or cause crashes by confusing CopyBits into dereferencing the base address of the pixmap if rowBytes exceeds the established limit of less than 0x4000.

Engineering is studying future solutions. It's possible that a future QuickDraw release will support pixmap with a rowBytes constant value indicating that the real rowBytes is contained in the planeBytes field instead. You might think of cases where this is also going to cause problems but the problems probably are less important than the limitation being overcome.

For the present, the solution depends on the conditions under which the problem affects you. If you're writing an application, the solution could be to patch CopyBits and call DrawPicture. When CopyBits is called while a picture is being drawn the source is a pixmap; if rowBytes is too big, your application could split the job banding the image vertically until the resulting rowBytes values fall within range. With a little debugging it would be possible to find where CopyBits calls the bit/pixmap/port checking routine and bypass that, given that the actual routine doing the blitting doesn't have problems dealing with pixmaps of less than 0x8000 rowBytes.

Another possibility is to call StdBits directly since it doesn't mind dealing with larger than legal rowBytes. The problem here is that the destination is implied and the application has to make sure that everything is all right. Also if the destination spans multiple devices the application must divide the task, targeting each device at the time. See the DeviceLoop procedure in Inside Macintosh Volume VI for ideas on this.

To recap: The limitation of rowBytes is becoming increasingly painful, now that applications can easily create pixmaps (and PICTs) that exceed the limit of 0x4000. It's possible for an application to patch CopyBits in order to work around this limitation but you have to decide what is appropriate for each set of conditions.

Back to top

Animation speed on the Macintosh

Date written: 1/18/93

Last reviewed: 4/1/93

How can I get reasonably fast animation on Macintosh models? So far, I've created off-screen pixmaps with the image and mask, and an extra off-screen pixmap for use in restoring the original picture. However, CopyPix is still too slow. I had to write my own CopyPix routine in assembly, which works great. But I can't help wondering how Apple expected fast animation to be accomplished.

You're certainly right that the way to increase performance is by off-screen drawing and QuickDraw's CopyBits procedure. The key information that's useful to you is in the Macintosh Technical Note "Of Time and Space and -CopyBits." This tech note covers some of the factors affecting CopyBits and what can be done to improve the speed of calling CopyBits.

Just to mention some of the factors that might improve the speed of calling `CopyBits`: Avoid color mapping, or even try faking out color mapping by setting your `ctSeed` to be the same. Alignment of pixels in the source map relative to their alignment to the destination pixel map can be important. If the source and destination rectangles are different sizes, `CopyBits` has to scale the copied image, which slows it down a lot. Also dithering and bit depth has effect on the speed of `CopyBits`.

Whereas QuickDraw often trades performance for flexibility, there are times we'd just as soon trade flexibility for performance. In these cases, we can achieve tremendous gains in speed by writing custom routines to draw to off-screen worlds. I recommend the article "Drawing in GWorlds for speed and versatility" (*develop,* issue 10) which shows you exactly how to do that.

Many developers want to go beyond the speed of QuickDraw. Writing directly to the screen can allow you to create faster animation and graphics than possible with QuickDraw. However, Apple has always maintained that writing to video memory is unsupported, since it may cause your application to break on future systems. If you write directly to the screen, your application will forfeit the use of many Toolbox managers and will put future compatibility at risk. Since most applications require the Window Manager and other basic Macintosh managers, writing to the screen is for only a few specialized applications, such as video games and some animation packages that compete on the quality and speed of graphics. The most important thing to remember is *don't write directly to the screen if you don't have to.* But if you do need to, an article that provides some guidelines is "Graphical Truffles: Writing Directly to the Screen," *develop,* Issue 11.

Following are some additional articles that animation and game developers find useful:

1. "Palette Manager Animation," *develop,* Issue 5.
2. "Using the Palette Manager Off-Screen," *develop,* Issue 10.
3. "QuickDraw's `CopyBits` Procedure: Better Than Ever in System 7.0," *develop,* Issue 6.
4. "Graphical Truffles: Animation at a Glance," *develop,* Issue 12.

Also, some sample code that can speed your development is given as SC.015.Offscreen and SC.016.OffSample (Dev CD Jan 93:Technical Documentation:Sample Code: Macintosh Sample Code).

Animation on the Macintosh does take some work. Nevertheless, we've seen some pretty amazing animations developed on the Macintosh.

Back to top

Inside Macintosh Volume V PnPixPat & BkPixPat doc fix

Date written: 12/4/92

Last reviewed: 3/1/93

Inside Macintosh Volume V (page 103) says that when a PICT pattern opcode (for instance, 0x0012) comes along, and the pattern isn't a dither pattern, the full `pixMap` data follows the old-style 8-byte pattern. The full pixMap data structure described on page 104 indicates that a `pixMap` starts with an unused long (`baseAddr` placeholder), followed by the `rowBytes`, bounds, and so on. However, looking at the Pict.r file on the October 1992 Developer CD, at the same opcode (`BkPixPat` == 0x0012), the first data field after the old-style pattern (hex `string[8]`) is the `rowBytes` field (broken down into three bitstrings). The `baseAddr` placeholder field isn't there. Which is correct?

The *Inside Macintosh* documentation on pages V-103 and V-104 is wrong. The Pict.r file correctly describes the format of the `PnPixPat` and `BkPixPat` opcodes. So there shouldn't be a `baseAddr` field in the `pixMap` record of a pattern as stored in the `PnPixPat` of a PICT. However, the `baseAddr` does occur in a `'ppat'` resource as described on page V-79. Thanks for pointing out this discrepancy.

Back to top

Disabling Macintosh Color QuickDraw for testing

Date written: 9/14/92

Last reviewed: 6/14/93

Is there an easy way to disable Color QuickDraw on a Macintosh? I want to do this for testing our application, to make sure it works correctly on a machine without Color QuickDraw.

There's no easy, or perhaps even hard way to disable features built into the system software your particular machine requires. It's designed to work well, not to be toggle-able.

Even the hard way isn't a sure thing - trying to patch out all the Color QuickDraw traps could confuse the rest of the system software, which internally may use undocumented routines to accomplish its tasks.

The easiest way to test on non-Color QuickDraw machines is to get one. Fortunately, the machines without Color QD are the lowest end of the Macintosh price spectrum - such as the Macintosh Classic, PowerBook 100, and Macintosh SE. You can probably rent or borrow one of these if the prices don't fit your current budget.

Back to top

Using a Macintosh PICT file that's larger than available memory

Date Written: 6/18/90

Last reviewed: 6/14/93

How can I read a 2 MB PICT file into only 1 MB of memory?

You can't read it in since you don't have enough memory, but drawing the picture contained in the file using a technique called "spooling" increases your chances of using a 2 MB PICT file with 1 MB memory. Spooling is documented in the Color QuickDraw chapter of *Inside Macintosh* Volume V (pages 88-89).

Back to top

Getting a single scan line from a PICT file

Date Written: 6/18/90

Last reviewed: 6/14/93

Is there any way to obtain a particular scan line from a PICT file?

A PICT file may contain more than just pixmaps, so getting one scan line out of it is not possible. The file may also contain other elements that overlap, such as rects and arcs. The only way to obtain a single line is to draw it off-screen and then, once the whole image is in memory, you can go and study individual pixels.

Back to top

Determining pixel depth from PICT files

Date Written: 6/20/90

Last reviewed: 9/17/91

How do you find out the pixel size of a PICT file on the disk?

A picture is by nature independent of depth. For example, you can have a picture containing `DrawRects` and `LineTos` and therefore lacking any info regarding depth.

On the other hand, if the picture you are looking at has pixmap opcodes in it, then each pixmap contains its own pixel size and in this case a picture can have a number of depths associated with it.

If you want to see the pixel size for each pixmap opcode in a picture, replace all the bottleneck routines and every time the `bitsProc` is called you can see the pixmap and get the info out. Since the picture is in a file, you can use the spooling technique described in the QuickDraw chapter in *Inside Macintosh* Volume V. Be ready to deal with multiple, possibly different, pixmaps as well as direct pixmaps if the picture was created under 32-bit QuickDraw.

"KnowsPICT," on the *Developer CD Series* disc, extracts this kind of information. The System 7.0 Picture Utilities package gets this information too.

Back to top

BitMapToRgn for non-Color QuickDraw Macintosh models

Date Written: 11/9/90

Last reviewed: 6/14/93

Is `_BitmapToRegion` available on any pre-System 7 non-Color QuickDraw configurations such as the Macintosh Classic, Plus, or SE? If not, is source or a library module available so that I don't have to take the time and compatibility risk of rolling my own?

`BitMapToRegion` works on pre-color Macintosh systems. You can license `BitMapToRegion` from

Software Licensing

Apple Computer, Inc.

20525 Mariani Ave. MS:38-I

Cupertino, CA 95014

Phone:(408) 974-4667

Back to top

Macintosh QuickDraw pixel map stack requirements

Date Written: 12/3/90

Last reviewed: 5/21/91

What are the guidelines for determining how much of an image `CopyBits` can copy to a Macintosh pixel map at one time, given a particular set of characteristics for the source map and the destination map and given how much stack space is available? For example, say that we have an 8-bit-deep pixmap to be copied to a 32-bit-deep pixMap using the `ditherCopy` mode and expanded by a factor of 4, and we have 45K of stack space.

`CopyBits`' stack requirement depends on the width of each scan line (`rowBytes`). The rule of thumb is that you need at least as much stack as the `rowBytes` value in your image (which can be huge with 32-Bit QuickDraw), with the following additional modifiers: Add an additional `rowBytes` for dithering; add an additional `rowBytes` for any stretching (source rect != dest rect); add an additional `rowBytes` for any color map changing; add an additional `rowBytes` for any color aliasing. The stack space you need is roughly five times the `rowBytes` of your image. In general, you're better off processing narrower scan lines. Reducing the vertical size will not affect stack requirements. Narrow, tall bands (if you can use them) will reduce the stack requirements.

Back to top

Color and non-Color QuickDraw trap dispatch differences

Date Written: 1/28/91

Last reviewed: 6/14/93

Why does a call to `RGBForeColor` cause a corruption of the stack without resulting in an unimplemented trap error on non-Color QuickDraw Macintosh systems?

The trap dispatcher on Color QuickDraw and non-Color QuickDraw machines are different. If you look at page 89 of *Inside*

Macintosh Volume I, you'll see the toolbox trap word format as it was in the days before Color QuickDraw. Bit 9 was "reserved for future use" and was ignored by the trap dispatcher, and so it was normally set to 0. That means that valid toolbox traps could either look like $A8XX or $A9XX as long as the auto-pop bit was turned off. Color QuickDraw machines have a trap dispatcher that uses that reserved bit to allow for more trap words, and therefore it has a much larger trap dispatch table. Color QuickDraw traps have that reserved bit set, so those traps look like $AAXX or $ABXX.

When a non-Color QuickDraw machine tests to see if a trap is implemented or not, it just checks the trap dispatch table to see if a routine is implemented for that trap or not. Because the reserved bit is ignored, trap words that look like $AAXX are treated as equivalent to $A8XX and trap words that look like $ABXX are treated as equivalent to $A9XX. The trap word for RGBForeColor is $AA14. If you call RGBForeColor on a non-Color QuickDraw machine, $AA14 is treated as $A814, which is the trap word for SetFractEnable. SetFractEnable is implemented on 128K ROM machines or greater, so no unimplemented trap error occurs.

If you look at recent DTS sample programs, such as the Utilities sample (SC.025.Utilities, which you can find on AppleLink in Developer Support and on the current developer CD), you'll see a routine in Utilities.c called TrapExists. It takes into account the size of the trap dispatch table so that you can tell in one call whether a routine is implemented or not regardless of whether it's a Color QuickDraw trap or not and regardless of what kind of Macintosh you're running on.

Under system software version 7.0, the trap dispatcher is modified on non-Color QuickDraw machines so that many Color QuickDraw traps are implemented and work as well as they can in black and white.

Back to top

Macintosh OpenCPicture 72-dpi calculation bug

Date Written: 2/12/91

Last reviewed: 6/14/93

The 32-Bit QuickDraw _OpenCPicture call incorrectly calculates the 72-dpi frame width if the height of the native resolution srcRect exceeds 910 dots. To work around this problem, I calculate the 72-dpi frame independently, and store it in the PicHandle returned by _OpenCPicture.

It's a known bug that under Macintosh system software versions 6.0.5 and 6.0.7 with 32-Bit QuickDraw 1.2, OpenCPicture doesn't properly calculate the right coordinate of the 72-dpi picFrame if the height of the srcRect (native resolution rectangle) multiplied by 72 exceeds $0000FFFF. That works out to a maximum height of 910 pixels, just as you found. This bug is fixed in System 7.0, but gestaltQuickdrawVersion returns $0220 both under system software versions 6.0.5 and 7.0, so you can't tell whether the bug is fixed that way. Instead, you should use Gestalt with the gestaltSystemVersion selector. If the returned value is $0700 or greater, then let OpenCPicture handle the picFrame calculation; otherwise you should do the calculation yourself.

Back to top

GetGWorldPixMap bug and workaround

Date Written: 3/12/91

Last reviewed: 6/14/93

Why does GetGWorldPixMap (when called on a Macintosh II, IIcx, or IIx running system software version 6.0.5 or 6.0.7 with 32-Bit QuickDraw 1.2) return a combination of the device field (two bytes) and the first two bytes of the portPixMap field? Is this a bug?

Your analysis of GetGWorldPixMap is exactly right: It doesn't work correctly in system software version 6.0.5 and 6.0.7 with 32-Bit QuickDraw 1.2. It returns a value that's two bytes before the value it's supposed to return.

The solution is to use GWorldPtr->portPixMap instead of GetGWorldPixMap. It's safe to do this, but you should use GetGWorldPixMap under System 7. Not only is the bug fixed there, but dereferencing the port is dangerous under System 7 because it may not be CGrafPort. Use Gestalt with the gestaltQuickdrawVersion selector to determine whether you can use GetGWorldPixMap. If Gestalt returns a value from gestalt8BitQD ($0100) through gestalt32BitQD12 ($0220), then GetGWorldPixMap either doesn't exist or is the buggy version. If it returns

gestalt32BitQD13 ($0230) or higher, then GetGWorldPixMap does exist and works correctly. Interestingly, GetGWorldPixMap can be called on a black-and-white QuickDraw machine under System 7. It returns a handle to a structure which should be treated as a BitMap structure, though there are some undocumented fields after the normal BitMap fields. To tell whether GetGWorldPixMap is available on a black-and-white QuickDraw machine, you must check the system software version by calling Gestalt with the gestaltSystemVersion selector. If it returns $0700 or higher, GetGWorldPixMap is available.

Back to top

System 7 TextMode problem and workaround

Date Written: 6/12/91

Last reviewed: 8/13/91

Our application uses the TextMode (blend + mask) as documented in *Inside Macintosh* Volume V (blend is equal to the current ditherCopy constant) to make translucent text. Under System 7, this transfer mode causes garbage to appear when the text is drawn. Is there a way to work around the problem? Will there be a fix?

The problem you are seeing is due to the use of CopyDeepMask instead of the old-fashioned CopyBits to do the job. It is being studied now, and the hope is that it will work as advertised in a future release. One workaround is to render the text to an off-screen pixmap and then call CopyBits (using blendMode) to actually put it in the picture.

Back to top

Using dithering and animation on the same Macintosh image

Date Written: 6/19/91

Last reviewed: 6/14/93

When setting up a dithered grayscale image for subsequent animation (to adjust brightness, for example), a conflict arises between the use of Palette animation and the ditherCopy CopyBits mode. This problem is demonstrated in the *develop* #5 GiMeDaPalette code sample: If you change srcCopy to ditherCopy in the CopyBits call, then run the program and select Animate, the resulting image is pure black and white, with what appears to be an attempt to dither with just the black-and-white color table entries (that are not reserved for animation).

This happens because ditherCopy tries to use the inverse table to do color matching, but when the image is animated, the inverse table colors are limited to just black and white.

To work around the problem, you can jump into the bottlenecks and when you see the PICT hitting the opcode for CopyBits, change the mode adding the ditherCopy constant. This way the dithering happens when you do the call to DrawPicture and not later on. This makes it possible to use dithering and animation on the same image.

Back to top

Rendering color PICTs in a black-and-white environment

Date Written: 7/22/91

Last reviewed: 9/17/91

I want to be able to render a color PICT as a black-and-white image substituting patterns for colors. My images are pretty small and have fewer than 16 colors. What do you suggest as the easiest way?

One easy way is to take advantage of 32-Bit QuickDraw and System 7.0's ditherCopy transfer mode modifier or flag (documented in *Inside Macintosh* Volume VI, page 17-17). Call DrawPicture into an off-screen pixmap with the pixel depth of the original color PICT. Then call CopyBits to copy the pixmap to the screen, with srcCopy + ditherCopy as the transfer mode. This will result in a nicely dithered image on the black-and-white end.

Under System 6 without 32-Bit QuickDraw, the solution is not nearly so cut and dried. One way might be to take advantage

of the fact that DrawPicture goes through the QuickDraw bottlenecks for drawing. For each grafproc in your PICT, you'd intercept StdBits during DrawPicture and call your own dithering routine to examine the foreground color and set the pen pattern or fill pattern so that it has about the same lightness as the original color.

Well, this came out as a great sales pitch for writing a System 7-savvy app!

Back to top

Highlighting ignored if foreground same as background color

Date Written: 8/7/91

Last reviewed: 6/14/93

Under System 7, but not System 6, HiliteMode doesn't work when the foreground and background colors are similar. Is this a bug?

Yes, it's a bug. The problem you encounter exists whenever the background and foreground color map to the same color table index. If the foreground color is the same as the background color, highlighting is ignored. Therefore, you should always make sure the foreground and background colors are different when using HiliteMode.

Back to top

Gestalt 'qdrw' selector bug and workaround

Date Written: 8/1/91

Last reviewed: 6/14/93

Why does Gestalt tell me I have Color QuickDraw features on a non-Color QuickDraw machine?

The gestaltQuickdrawFeatures ('qdrw') selector, used for determining your system's Color QuickDraw features, has a bug that causes it to tell you incorrectly that noncolor machines have color. The fix is quite simple: Gestalt has another selector, gestaltQuickdrawVersion ('qd '), which simply returns the QuickDraw version number. This version number is < gestalt8BitQD for classic QuickDraw and >= gestalt8BitQD for Color QuickDraw (see *Inside Macintosh* Volume VI, page 3-39, for more information). The trick is to ask Gestalt for the QuickDraw version first; once you've determined that you have Color QuickDraw, the 'qdrw' selector is OK to use to find out specifics.

Back to top

GetPixelsState is slow sometimes

Date Written: 8/27/91

Last reviewed: 6/14/93

Why do I sometimes see incredible slowdowns under System 7.0 when calling either GetPixelsState or LockPixels (I'm not sure which) for the PixMapHandle of a GWorld allocated in temporary memory?

GetPixelsState takes an arbitrary amount of time since it makes a call to RecoverHandle to get the handle pointing to the baseaddr. Therefore, the slowdown you see as actually due to the call to RecoverHandle, which is slow because it must traverse the heap to find the pointer to the baseaddr. LockPixels is not responsible for the slowdown because it does not make call to any traps that could take an extended amount of time.

Back to top

OpenCPicture and PICTs other than 72 dpi

Date Written: 10/2/91

Last reviewed: 6/14/93

Can I use OpenCPicture to create PICTs with a higher resolution than 72 dots per inch (dpi)?

——

There's good news and bad news: The good news is that you're on top of the situation, which means the bad news is that there aren't better ways to do what you want to do, mostly. Here's the scoop:

You can use vRes and hRes in pictures opened with OpenCPicture to tell QuickDraw it's not a 72-dpi picture, and as long as the application that receives the picture uses DrawPicture to image it, QuickDraw will Do The Right Thing--scaling it on the screen to 72 dpi instead of making it humongously large. Unfortunately, this way you lose hairlines; if you print such a picture to a 72-dpi grafPort (like the LaserWriter driver normally returns), you'll get 1/72-inch lines instead of 1/300-inch lines as you probably want.

(This *can* work correctly, but the receiving application has to notice that your picture is bigger than 72 dpi and ask PrGeneral to increase the resolution of the printing grafPort accordingly, and this doesn't always or often happen.)

Back to top

No System 7 QuickDraw alpha channel support

Date Written: 10/23/91

Last reviewed: 6/14/93

How can I directly access the alpha channel (the unused 8 bits in a 32-bit direct pixel using QuickDraw) under System 7? Under System 6 it was easy, but under System 7's CopyBits the alpha channel works with srcXor but not with srcCopy.

With the System 7 QuickDraw rewrite, all "accidental" support for the unused byte was removed, because QuickDraw isn't supposed to operate on the unused byte of each pixel. QuickDraw has never officially supported use of the extra byte for such purposes as an alpha channel. As stated in *Inside Macintosh* Volume VI, page 17-5, "8 bits in the pixel are not part of any component. These bits are unused: Color QuickDraw sets them to 0 in any image it creates. If presented with a 32-bit image--for example, in the CopyBits procedure--it passes whatever bits are there."

Therefore, you cannot rely on any QuickDraw procedure to preserve the contents of the unused byte, which in your case is the alpha channel. In fact, even CopyBits may alter the byte, if stretching or dithering is involved in the CopyBits, by setting it to 0. Your alternatives are not to use the unused byte for alpha channel storage since the integrity of the data cannot be guaranteed, or not to use QuickDraw drawing routines that can alter the unused byte.

Back to top

BitsToRgn and MPW BitMapToRegionGlue

Date Written: 10/29/91

Last reviewed: 6/14/93

Which version of the system software first contained the call BitsToRgn? Is there a workaround for this call if my users have an earlier version of system software?

The call BitmapToRegion was introduced with 32-Bit QuickDraw and became fully documented in Volume VI of *Inside Macintosh,* which is primarily System 7 information. However, since the differences between System 7's QuickDraw and 32-Bit QuickDraw are minor, most of System 7's QuickDraw routines are available in system software prior to System 7.0 using the 32-bit QuickDraw INIT.

To check to see if a system contains 32-bit QuickDraw, you can use the following snippet of code:

```
/* Find out if GWorlds and CQD are implemented on this machine */
(void) Gestalt (gestaltQuickdrawVersion, /*<*/&qdVersion);
gHasGWorlds = (qdVersion > gestaltOriginalQD &&
               qdVersion < gestalt8BitQD)
              || qdVersion >= gestalt32BitQD;
```

If you are using MPW as a development platform, MPW has a library call you can use that will allow you to use the routine regardless of whether or not 32-bit QuickDraw exists. The glue routine is called `BitMapToRegionGlue()` and is available to MPW users. Substitute this call for `BitMapToRegion` calls and the glue code will take care of patching in the proper code if 32-bit QuickDraw does not exist. If you're using Think C, you can use the oConv utility to convert the MPW object file into a Think C usable format.

Back to top

Ensuring even `rowBytes` for 'cicn' resources

Date Written: 12/4/91

Last reviewed: 6/14/93

Is there any way to force bitmaps and masks within a `'cicn'` resource to have an even `rowBytes` (using ResEdit)? I want to avoid duplicating icon bitmaps--one for color systems set to B&W and one for B&W systems--to reduce program size as well as development and maintenance costs. The bitmaps in the `'cicn'` can also be sized specifically to the task, whereas the old B&W icons are of a fixed size and contain no sizing information. It's simple enough to read in a `'cicn'` and extract the bitmap. The problem is that on a 68000 (no Color QuickDraw), if `rowBytes` is odd, an odd address trap results.

There isn't any way to get ResEdit itself to create bitmaps with even `rowBytes` for 8 x 8 'cicn' resources, but here are few suggestions:

You could process your `'cicn'` resources first, so that they have bitmaps as you require them. To alter the resource with a quick little program would be trivial, especially given that the bitmap data sits last in the `'cicn'`. All you'd need to do is expand the bitmap image data by padding each line to an even length and then changing the `rowBytes` value. Or you could de-rez the `'cicn'`s and patch them with a text editor, either by hand or with a search-and-replace script of some kind.

Back to top

`CopyBits` blend mode: OpColor's affect & eliminating banding

Date Written: 12/11/91

Last reviewed: 6/14/93

I have two gray-colored pixmaps that I wish to blend together; one is on the screen, the other in an off-screen pixmap. I use `CopyBits` to copy the off-screen to the screen, but it does not seem to blend them. Instead, it seems to match the colors of the screen bitmap to the closest colors in some table, thus having the effect of reducing the number of colors displayed on the screen bitmap. Any suggestions?

There are two distinct questions here: 1) Why ain't it blending? and 2) What's this banding for? The first problem is almost certainly because `OpColor` isn't set properly. This is a third, implicit, operand on several arithmetic graphics operations, including blend. For blend, it describes the proportions to mix the source and destination colors in the blend. For an equal mix, you should set this color to a halfway gray. (Call `OpColor()` with a color where red, green, and blue all equal $8000.) This effect is described in the description of the blend mode on page 60 of *Inside Macintosh* Volume V. Unfortunately, the initial value for `OpColor` is black (0,0,0), so you were seeing no mixing of your off-screen data.

The second half of your question is why you're getting a banding effect. (When you fix the above problem, you'll still get banding.) Unfortunately, the arithmetic modes are constrained by the size of the inverse table. As your screen no doubt uses the default 4-bit inverse table, you'll find that you'll get only 2^4 = 16 levels of gray. If you enlarge your screen's

inverse table to 5 bits, the maximum allowable, you'll still only get 32 gray levels. (To do this, set the gdResPref field in the GDevice to 5, then call MakeITable().) The only way to get a fully-gradual, great-looking effect is to do all the work off-screen in 24-bit deep pixmaps, and then copy it to the screen. Because they can operate directly on colors, rather than having to work through the intermediary of color indices, direct pixmaps are not limited by inverse tables (in fact, they don't even have real inverse tables). You could use 16-bit pixmaps, but they only provide 32 grays (having only 5 bits for each component), so this wouldn't be any better than increasing the size of the inverse table.

Back to top

Icon dimming under System 7 and System 6

Date Written: 1/6/92

Last reviewed: 6/14/93

When you bring up the Finder windows under System 7 on a color system and click a control panel item icon, it paints itself that fancy gray. How can I get that effect?

To get the fancy System 7 icon dimming to work in your program, read the Macintosh Technical Note "Drawing Icons the System 7 Way," and use the icon-drawing routines contained in it. The routines show how to use the Icon Toolkit, which is what the Finder uses. If you want the same effect under System 6, you'll have to emulate the dimming of the icons via QuickDraw; the IconDimming sample code in the Snippets folder on the *Developer CD Series* disc shows how to do this.

Back to top

QuickDraw out of memory if debugger invoked by "Jackson"

Date Written: 3/11/92

Last reviewed: 6/14/93

I am getting a strange bug in which the Macintosh debugger is being invoked by an A-trap marked "Jackson" when I call SetCCursor in certain situations and a second monitor is hooked up. The cursor structure being passed appears to be valid. I've also been crashing unexpectedly in this same spot for the past few weeks. I assume Jackson is some kind of error assertion that was left in System 7's Color QuickDraw code. What gives?

Jackson was a code name for 32-Bit QuickDraw. The trap you refer to is in fact never called; it's not supposed to be encountered by you ever. The trap is reserved for Apple to use in future versions of Color QuickDraw. If you examine the code directly preceding the _Debugger, you will notice that it is doing

```
        MOVEQ       #$19,D0
        JSR         ([$1524])
```

which for you and me is

```
         MoveQ #25,D0                ; say that memory is full...
        _SysError                    ; and call syserror
```

the line following would be...

What's all this tell you? You have a debugger installed that is rts'ing from the SysError vector (you aren't supposed to return from SysError normally), or you have installed your own SysError vector which is rts'ing. At any rate, if you examine the code directly following the debugger statement and see what it does, you might imagine the source code looks something like this:

```
MemFull       MoveQ #25,D0              ; say that memory is full...
              _SysError                 ; and call syserror
; If it returns better go into the debugger since its not supposed to return
              _Debugger                 ; Hey! sysError came back!
;
CallNewHand   _NewHandle
              bne.S  MemFull            ; could not get the memory, just die
              rts
```

What's happening is that you're running out of memory somehow (several places call `MemFull`, not just the above place), so you'd need to use a stack crawl to figure out how you got there. But, the bottom line, QuickDraw has run out of memory and cannot continue; it tried to put up a system error dialog to tell the user and for some reason the machine did not get restarted and the SysError vector returned. You are now in your debugger, since QuickDraw put up the system error dialog because it could not continue.

Back to top

ditherCopy not supported on LaserWriter or ImageWriter

Date Written: 5/31/91

Last reviewed: 11/6/91

`ditherCopy` is not supported on LaserWriters or ImageWriters. On a LaserWriter, `ditherCopy` gets misinterpreted and inverts the image. On an ImageWriter it's treated as a `srcCopy`. The ImageWriter driver doesn't support color `grafPorts`, which is the only way to do the pixel image required for `ditherCopy`. Use `srcCopy` instead for both printers.

Back to top

Macintosh Color QuickDraw CalcCMask and SeedCFill clarified

Date Written: 1/1/90

Last reviewed: 11/21/90

I'm having trouble using `CalcCMask` and `SeedCFill`. What am I doing wrong?

There is some confusion regarding the use of the Macintosh Color QuickDraw routines `CalcCMask` and `SeedCFill`, which are analogous to the older `CalcMask` and `SeedFill`. Much of the confusion was caused by early documentation errors. Be sure you have the release version of Volume 5 of *Inside Macintosh* and version 2.0 or later of the MPW interface files.

The correct interface for CalcCMask is:

```
PROCEDURE CalcCMask(srcBits, dstBits: BitMap;
          srcRect, dstRect: Rect;
          seedRGB:     RGBColor;
          matchProc:    ProcPtr;
          matchData:    LongInt);
```

The correct interface for SeedCFill is:

```
PROCEDURE SeedCFill(srcBits, dstBits: BitMap;
          srcRect, dstRect: Rect;
          seedH, seedV:   INTEGER;
          matchProc:    Ptr;
          matchData:    LongInt);
```

Each routine calculates a one-bit deep bitmap representing either the mask or the fill area depending upon the routine. In both cases, the source bitmap may be either a bitmap or a pixmap, but the destination must be a bitmap, because it must have a depth of one-bit.

It is difficult to pass a pixmap for the source parameter because of Pascal's type checking. To get around this difficulty, you can declare a new type:

then use it to coerce the pixmap as follows:

If you have a PixMapHandle, do the following:

If you are using a grafPort (or a window), you can pass myWindow^.portBits and not have to worry about whether the port uses a bitmap or a pixmap.

Most of the other parameters are explained in detail in *Inside Macintosh.* To use the matchProc and the matchData parameters, though, you need more information.

As stated in *Inside Macintosh,* the matchProc parameter is a pointer to a routine that you would like to use as a custom SearchProc. To better understand how this is used, it is helpful to know how SeedCFill and CalcCMask actually work.

Both routines start by creating a temporary bitmap which, by definition, is one bit deep. The source pixmap (or bitmap) is then copied to the temporary bitmap using CopyBits. This copy causes the image to be converted to a depth of one-bit. Now with a normal black-and-white image, the standard CalcMask or SeedFill routine is used to generate the destination bitmap.

Most of the real work is done in the original call to CopyBits, which maps the pixmap image to a monochrome bitmap equivalent. For each color in the source pixmap, CopyBits will map it to either black or white. Which colors map to black and which ones to white is determined by the SearchProc.

SeedCFill installs a default SearchProc that maps all colors to black except for the color of the pixel at (seedH,seedV). SeedFill then calculates as usual the fill mask for the white bits.

The default SearchProc for CalcCMask maps all colors to white except the color passed in the seedRGB parameter. The seedRGB parameter, then, would be the color of the item that you wanted to "lasso."

But suppose you want to fill over all colors that were shades of green, not just the particular shade of green at (seedH,seedV). Or maybe you want to fill over all colors that are lighter than 50 percent brightness. Or maybe you want to use dark colors as edge colors for CalcCMask. To do such things, you need to pass a pointer to your own SearchProc in the matchProc parameter.

Because your matchProc is just a custom search procedure for the Color Manager, it should be declared as one, but Volumes I-V of *Inside Macintosh* have documented this routine incorrectly. The correct declaration for a custom SearchProc is as follows:

```
FUNCTION SearchProc(VAR RGB: RGBColor;
          VAR result: LongInt) : Boolean;
```

Normally, as each SearchProc is installed, it is added to the head of the SearchProc chain, so that it is called before all of the other ones that were already installed. When a SearchProc is installed, it can do one of three things:

1. Completely ignore the call by returning FALSE and not modifying any of the input parameters;

2. Completely handle the call by setting the result parameter to be the index into the color table that matches (according to your rules) the RGB parameter. In that case, the SearchProc returns TRUE;
3. Partially handle the call by modifying the RGB parameter, then returning FALSE.

In cases 1 and 3, the Color Manager continues down the SearchProc chain until it finds one that returns TRUE. If none of the custom routines handle the call, then the built-in default routine is used. In case 3, you can change the RGB color that is being matched. For example, if you want all shades of green to map to pure green, modify the RGB color, then return FALSE, letting the Color Manager find the index of that green in the color table.

In case 2, you return TRUE to indicate that you handled the call, and you return the color table index in the result parameter. The Color Manager then uses that index. For example, if you want to substitute white for all colors that can't be matched exactly in the color table, then each time you get called you either return the index into the color table of the exact color, or 0 (which is the index for white) for all other colors.

A custom SearchProc for SeedCFill and CalcCMask should always return TRUE because the default Color Manager SearchProc usually doesn't make sense. Because SeedCFill and CalcCMask are using CopyBits to copy to a 1-bit bitmap, you need to set the result to be either 0 or 1 (the only possible values in a 1-bit bitmap). A result of 0 is white, and a result of 1 is black.

All colors for SeedCFill that should be "filled over" would generate a result of 0 (white), and all colors that stop the fill generate a 1 (black). SeedFill is then called to fill the white area. All colors for CalcCMask that you want to form boundaries should generate results of 1 (black).

When your SearchProc gets called, the gdRefCon field of the current GDevice (theGDevice^^.gdRefCon) contains a pointer to the following record:

```
matchRec  =  RECORD
  red:       Integer;
  green:     Integer;
  blue:      Integer;
  matchData: LongInt;
END;
```

The red, green, and blue parameters for SeedCFill are the values of the color of the pixel at (seedH,seedV). For CalcCMask, they are the fields from the seedRGB parameter. Your SearchProc can use this information to decide which colors are "fill-over" colors and which colors are "boundary" colors. For example, if you always set (seedH,seedV) to be the mouse point, your SearchProc then bases its decisions using the color of the pixel under the cursor. For example, the user clicks a shade of green, so all shades of green get filled over.

The matchData field contains the value that you passed into the SeedCFill or CalcCMask routines in the matchData parameter. The use of this field is completely user-defined. For example, since your SearchProc routine may be a separate module, you might want to use this field to pass a handle to your variables. This field can contain a handle, a pointer, a long integer, or whatever; or you can just ignore this field altogether.

Warning:
There are some features of CalcCMask and SeedCFill you should be aware of. To understand them, you should be familiar with the use of CalcMask and SeedFill, which are described in the QuickDraw chapter of *Inside Macintosh* Volume IV.

CalcCMask and SeedCFill both use a parameter set that is very similar to the one used by CopyBits. CalcMask and SeedFill, however, are a different story. Instead of passing bitmaps and rectangles to SeedFill and CalcMask, these routines use an unusual set of parameters that describe the memory to be operated upon in terms of pointers, height, width, and offsets to the next row (rowBytes). Although these parameters are fairly easy to calculate, there are some limitations.

The most restrictive limitation is that the width of the rectangle used must be an even multiple of 16 bits. This limitation exists because the width of the rectangle is passed to SeedFill and CalcMask as a number of words (2 bytes). When calculating this parameter, SeedCFill and CalcCMask round down to an even word boundary. This rounding means that the rectangles you pass to CalcCMask and SeedCFill should be an even multiple of 16 pixels in width. If they are not,

then the rightmost portion of the mask will be garbage.

To figure out the color of the pixel at (seedH,seedV), SeedCFill calls GetCPixel. GetCPixel finds the color of the pixel at (h,v) in the current port. Therefore, if you pass a pixmap that is not the pixmap of the current port you will get bizarre results. In other words, seedH and seedV are expressed in the local coordinates of the current port, not the coordinate of the source pixmap.

You have two methods to make it work. First, always pass the pixmap of the current port as the source parameter. If you are using an off-screen pixmap, it is a good idea to have an associated port for it, and then call SetPort, passing it a pointer your off-screen port, before you call SeedCFill.

The second method involves letting SeedCFill get some wrong value for the color at (seedH,seedV) then using your own custom SearchProc to do the real work. The default SearchProc for SeedCFill relies on getting the correct color, but your SearchProc doesn't have to.

SeedCFill also makes the assumption that the seedH and seedV parameters are in the local coordinate system of the destination bitmap. This assumption comes into play when SeedCFill calculates the seedH and seedV parameters for SeedFill.

All this means that SeedCFill only works correctly if the source pixmap, destination pixmap, and current port all use the same coordinate system. Because of the above problem, this is almost automatic since the current port's portRect and the bounds of the source pixmap have to be the same anyway.

The easiest way to make all this work is to have your main port be an even multiple of 16 pixels wide. Then, make sure that your source and destination structures (pixmap or bitmap) are all the same size and all have origins of (0,0).

Back to top

Macintosh PICT color picture file format

Date Written: 1/1/90

Last reviewed: 6/14/93

Is there a general file format for color pictures that is common to all of the color paint programs? If so, where is it documented?

Apple supports (and encourages developers to support) one file type for pictures: the PICT file type. Most paint-type programs handle PICT files.

A PICT file is composed of two parts in its data fork; the first 512 bytes are for the file header, which contains application-dependent information. You have to contact the individual publishers to find out their particular data structures. For example, you can contact Claris Technical Support at AppleLink CLARIS.TECH or (415) 962-0371 for the file header MacDraw writes to its files.

The rest of the data in the file is picture data as created by Macintosh QuickDraw with OpenPicture. You can find the information about this data in Volume V of *Inside Macintosh* (pages 84-105); this section also shows how to read/write PICT files.

You can also check the Macintosh Technote "Displaying Large PICT Files" for more details on the subject.

X-Refs:

DTS Macintosh Technical Note "QuickDraw's Internal Picture Definition"

DTS Macintosh Technical Note "Displaying Large PICT Files"

Back to top

Mac pixmap is clipped to visRgn defined by screenBits.bounds

Date Written: 1/1/90

Last reviewed: 11/21/90

I'm drawing into a large off-screen bitmap (pixmap), but anything drawn outside the 640- by 480-pixel Macintosh screen area doesn't get written to the pixmap. Why not?

When you create a new port with `OpenPort` or `OpenCPort` the `visRgn` is initialized to the rectangular region defined by `screenBits.bounds` (IM I:163). If your port has a large `portRect`, any drawing will be clipped to the `visRgn` and you will lose any drawing outside of the `screenBits.bounds` rectangle.

To correct this set the `visRgn` of the port to coincide with your port's portRect after creating the port.

Also note that OpenPort initializes the `clipRgn` to a wide-open rectangular region (-32768, -32768, 32767, 32767). Some operations, like `OpenPicture`, can fail with this setup, so try setting clipRgn to a smaller rectangle.

X-Refs:

DTS Macintosh Technical Note "Pictures and Clip Regions"

DTS Macintosh Technical Note "Drawing into an Off-Screen Pixel Map"

Back to top

Using Macintosh System 7 OpenCPicture for higher resolution

Date Written: 1/1/90

Last reviewed: 6/14/93

We want to use `OpenCPicture` for higher resolution, not for color per se. Can `OpenCPicture` in System 7 be used with non-Color as well as Color QuickDraw Macintosh computers?

Yes, with System 7, `OpenCPicture` can be used to create extended PICT2 files from all Macintosh computers. Under System 6.0.7 or later, you must test for 32-Bit QuickDraw before using `OpenCPicture`. You can do this by calling Gestalt with the `gestaltQuickdrawVersion` selector. If it returns `gestalt32BitQD` or greater, then 32-Bit QuickDraw is installed.

Back to top

How to identify 32-Bit QuickDraw version

Date Written: 1/1/90

Last reviewed: 6/14/93

How can my program find out which version of Macintosh 32-Bit QuickDraw is running?

The following code snippet demonstrates how to use the Gestalt Manager to determine which version of 32-Bit QuickDraw is installed. There is no way to determine the version of 32-Bit QuickDraw before Gestalt. For 32-Bit QuickDraw version 1.2, `Gestalt` returns 2.2. *Inside Macintosh* Volume VI describes the Gestalt Manager in detail.

```
#defineTRUE0xFF
#defineFALSE0
#define Gestalttest0xA1AD
#define NoTrap0xA89F

main()
{
OSErrerr;
longfeature;

if ((GetTrapAddress(Gestalttest) != GetTrapAddress(NoTrap))) {
err = Gestalt(gestaltQuickdrawVersion, &feature);
if (!err) {
if ((feature & 0x0f00) == 0x0000)
printf ("We have Original QuickDraw version 0.%x\n", (feature & 0x00ff));
else if ((feature & 0x0f00) == 0x0100)
printf ("We have 8 Bit QuickDraw version 1.%x\n", (feature & 0x00ff));
else if ((feature & 0x0f00) == 0x0200)
printf ("We have 32 Bit QuickDraw version 2.%x\n", (feature & 0x00ff));
else
printf ("We don't have QD\n");
}
else
printf ("Gestalt err = %i\n",err);
}
else
printf ("No Gestalt\n");
}
```

Back to top

Macintosh QDError function under System 6 and System 7

Date Written: 1/1/90

Last reviewed: 12/7/90

Under what System 7 and System 6 conditions is it legal to call the Macintosh QDError function?

Under System 7, `QDError` can be called from all Macintosh computers. (System 7 supports `RGBForeColor`, `RGBBackColor`, `GetForeColor`, and `GetBackColor` for all Macintosh computers as well.) On a non-Color QuickDraw Macintosh, `QDError` always returns a "no error." Under System 6, QDError cannot be used for non-Color QuickDraw Macintosh systems.

Back to top

Macintosh `CopyBits` transfer modes changed for System 7

Date Written: 1/1/90

Last reviewed: 6/14/93

Why do some Macintosh `CopyBits` transfer modes produce different results for System 7 than for System 6?

Under System 6, the `srcOr`, `srcXor`, `srcBic`, `notSrcCopy`, `notSrcOr`, `notSrcXor`, and `notSrcBic` transfer modes do not produce the same effect for a 16- or 32-bit (direct) pixel map as for an 8-bit or shallower (indexed) pixel map. With Color QuickDraw these classic transfer modes on direct pixel maps aren't color-based; they're pixel-value-based. Color QuickDraw performs logical operations corresponding to the transfer mode on the source and destination pixel values to get the resulting pixel value.

For example, say that a multicolored source is being copied onto a black-and-white destination using the srcOr transfer mode, and both the source and destination are 8 bits per pixel. Except in unusual cases, the pixel value for black on an indexed pixel map has all its bits set, so an 8-bit black pixel has a pixel value of $FF. Similarly, the pixel value for white has all its bits clear, so an 8-bit white pixel has a pixel value of $00. CopyBits takes each pixel value of the source and performs a logical OR with the corresponding pixel value of the destination. Using OR to combine any value with 0 results in the original value, so using OR to combine any pixel value with the pixel value for white results in the original pixel value. Using OR to combine any value with 1 results in 1, so using OR to combine any pixel value with the pixel value for black results in the pixel value for black. The resulting image shows the original image in all areas where the destination image was white and shows black in all areas where the destination image was black.

Take the same example, but this time make the source and destination 32 bits per pixel. The direct-color pixel value for black is $00000000 and the direct-color pixel value for white is $00FFFFFF. CopyBits still performs a logical OR on the source and destination pixel values, but notice what happens in this case. Using OR to combine any source pixel value with the pixel value for white results in white, and using OR to combine any source pixel value with the pixel value for black results in the original color. The resulting image shows the original image in all areas where the destination image was black and shows white in all areas where the destination image was white--roughly the opposite of what you see on an indexed pixel map.

The newer transfer modes addOver, addPin, subOver, subPin, adMax, and adMin work consistently at all pixel depths, and often, though not always, correspond to the theoretical effect of the old transfer modes. For example, the adMin mode works similarly to the srcOr mode on both direct and indexed pixel maps. Also, 1-bit deep source pixel maps work consistently and predictably regardless of the pixel depth of the destination even with the old transfer modes.

Under system software version 7.0, the old transfer modes now perform by calculating with colors rather than pixel values. You'll find that transfer modes like srcOr and srcBic work much more consistently even on direct pixel maps.

Back to top

Which QuickDraw versions support SetEntries

Date Written: 3/3/92

Last reviewed: 6/14/93

I'm calling SetEntries to update the on-screen CLUT. Who implements this call? Does 32-Bit QuickDraw? In other words, does the 32-Bit QuickDraw INIT need to be around for this to work? What about monochrome machines?

I'm creating off-screen buffers by hand instead of using GWorlds. Is this the proper way of doing off-screen buffering when we don't want to require the user to have 32-Bit QuickDraw?

SetEntries is part of the Color Manager, which exists with all Color QuickDraw versions. A good rule of thumb to follow is that if it is documented in *Inside Macintosh* Volume V, you don't need 32-Bit QuickDraw to use it. *Inside Macintosh* Volume V documents standard Color QuickDraw. SetEntries does not work on monochrome Macintosh models, including the Classic II, SE, and PowerBooks.

Off-screen buffering: You should always use GWorlds if they exist; use Gestalt to test for them. This will assure that you can take advantage of the latest speed improvements. It is important to remember that under System 7 NewGWorld and accompanying calls are present in all Macintosh computers including black-and-white systems such as Classic and PowerBook 100 systems.

Back to top

Macintosh pixel map maximum rowBytes change

Date Written: 4/22/91

Last reviewed: 6/14/93

The Color QuickDraw section of *Inside Macintosh* Volume VI states that the restriction on the rowBytes field in a pixmap has been relaxed from $2000 to $4000. When did this happen? Is it true for all 32-Bit QuickDraw versions? This affects our user configuration recommendations.

The maximum `rowBytes` extension to $3FFE or less applies only to 32-bit QuickDraw. Using pixmaps with `rowBytes` greater than $1FFE when 32-bit QuickDraw is not present is likely to cause problems such as garbage images or system crashes. Remember that 32-bit QuickDraw is always present under System 7.0 or higher.

Back to top

Use assembly to flip a 24-bit off-port color pixmap

Date Written: 5/7/91

Last reviewed: 7/25/91

What's the best approach to horizontally flip a 24-bit off-port color pixmap?

Unfortunately, you won't be able to use `CopyBits` for this kind of procedure; you'll have to write your own routine to move each pixel. I'd suggest doing this in assembly language to squeeze the best possible performance out of your code.

Back to top

Why PlotCIcon requires GetCIcon instead of Get1Resource

Date Written: 4/26/91

Last reviewed: 6/17/91

Why do I have to use `GetCIcon(resID)` instead of `Get1Resource('cicn',resID)` for `PlotCIcon` to work correctly?

You apparently thought something that, at first, I thought also: that `GetCIcon(resID)` is just a utility routine that translates to `Get1Resource('cicn',resID)`. However, this is not the case; `GetCIcon` not only gets the `'cicn'` resource, but it also performs some minor surgery on the results, fills in some placeholder fields in the resource data, and the like. Basically, `PlotCIcon` can't work without the things that `GetCIcon` does.

Back to top

How Macintosh system draws small color icons

Date Written: 3/31/92

Last reviewed: 6/14/93

The code I added to my application's MDEF to plot a small icon in color works except when I hold the cursor over an item with color. The color of the small icon is wrong because it's just doing an `InvertRect`. When I drag over the Apple menu, the menu inverts behind the icon but the icon is untouched. Is this done by brute force, redrawing the small icon after every InvertRect?

The Macintosh system draws color icons, such as the Apple icon in the menu bar, every time the title has to be inverted. First `InvertRect` is called to invert the menu title, and then `PlotIconID` is called to draw the icon in its place. The advantage of using `PlotIconID` is that you don't have to worry about the depth and size of the icon being used. The system picks the best match from the family whose ID is being passed, taking into consideration the target rectangle and the depth of the device(s) that will contain the icon's image.

The Icon Utilities call PlotIconID is documented in the Macintosh Technical Note "Drawing Icons the System 7 Way"; see this Note for details on using the Icon Utilities calls.

Back to top

Spooling and preserving Macintosh QuickDraw pixmap depth

Date Written: 2/11/92

Last reviewed: 6/14/93

When a picture that contains a pixmap is spooled into a window, how and when is the depth of the pixmap in the picture converted to the depth of the screens the window is on?

When a picture is spooled in, if QuickDraw encounters any bitmap opcode, it allocates a pixmap of the same depth as the data associated with the bitmap opcode, expands the data into the temporary pixmap, and then calls `StdBits`. `StdBits` is what triggers the depth and color conversions as demanded by the color environment (depth, color table, B&W settings) of the devices the target port may span (as when a window crosses two or more screens).

If there's not enough memory in the application heap or in the temporary memory pool, QuickDraw bands the image down to one scan line and calls `StdBits` for each of these bands. Note that if you're providing your own `bitsProc`, QuickDraw will call it instead of `StdBits`.

This process is the same when the picture is in memory, with the obvious exception that all the picture data is present; the color mapping occurs when StdBits does its stuff.

Back to top

Determining the resolution of a PICT

Date Written: 6/10/92

Last reviewed: 6/14/93

In a version 2 picture, the `picFrame` is the rectangular bounding box of the picture, at 72 dpi. I would like to determine the bounding rectangle at the stored resolution or the resolution itself. Is there a way to do this without reading the raw data of the PICT resource itself?

With regular version 2 PICTs (or any pictures), figuring out the real resolution of the PICT is pretty tough. Applications use different techniques to save the information. But if you make a picture with `OpenCPicture`, the resolution information is stored in the `headerOp` data, and you can get at this by searching for the `headerOp` opcode in the picture data (it's always the second opcode in the picture data, but you still have to search for it in case there are any zero opcodes before it). Or you can use the Picture Utilities Package to extract this information.

With older picture formats, the resolution and original bounds information is sometimes not as obvious or easily derived. In fact, in some applications, the PICT's resolution and original bounds aren't stored in the header, but rather in the pixel map structure(s) contained within the PICT.

To examine these pixmaps, you'll first need to install your own `bitsProc`, and then manually check the bounds, `hRes`, and `vRes` fields of any pixmap being passed. In most cases the `hRes` and `vRes` fields will be set to the `Fixed` value 0x00480000 (72 dpi); however, some applications will set these fields to the PICT's actual resolution, as shown in the code below.

```
Rect      gPictBounds;
Fixed      gPictHRes, gPictVRes;

pascal void ColorBitsProc (srcBits, srcRect, dstRect, mode,
  maskRgn)
BitMap    *srcBits;
Rect      *srcRect, *dstRect;
short      mode;
RgnHandle  maskRgn;
{
  PixMapPtr  pm;
  pm = (PixMapPtr)srcBits;
  gPictBounds = (*pm).bounds;
  gPictHRes = (*pm).hRes;    /* Fixed value */
  gPictVRes = (*pm).vRes;    /* Fixed value */
}
void FindPictInfo(picture)
PicHandle  picture;
{
  CQDProcs    bottlenecks;
  SetStdCProcs (&bottlenecks);
  bottlenecks.bitsProc = (Ptr)ColorBitsProc;
  (*(qd.thePort)).grafProcs = (QDProcs *)&bottlenecks;
  DrawPicture (picture, &((**picture).picFrame));
  (*(qd.thePort)).grafProcs = 0L;
}
```

Back to top

Downloadables

	Acrobat version of this Note (K).	Download

---