New Technical Notes
Macintosh

®

Developer Support

Color QuickDraw Q&As
Imaging

Revised by: Developer Support Center June 1993
Written by: Developer Support Center October 1990

This Technical Note contains a collection of Q&As relating to a specific topic—questions you've sent the Developer Support Center (DSC) along with answers from the DSC engineers. While DSC engineers have checked the Q&A content for accuracy, the Q&A Technical Notes don't have the editing and organization of other Technical Notes. The Q&A function is to get new technical information and updates to you quickly, saving the polish for when the information migrates into reference manuals.

Q&As are now included with Technical Notes to make access to technical updates easier for you. If you have comments or suggestions about Q&A content or distribution, please let us know by sending an AppleLink to DEVFEEDBACK. Apple Partners may send technical questions about Q&A content to DEVSUPPORT for resolution.

New Q&As in this Technical Note:
Animation speed on the Macintosh

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:

• "Palette Manager Animation," *develop,* Issue 5.
• "Using the Palette Manager Off-Screen," *develop,* Issue 10.
• "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0," *develop,* Issue 6.
• "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.

***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.

Disabling Macintosh Color QuickDraw for testing
Date written: 9/14/92
Last reviewed: 11/1/92

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.

Using a Macintosh PICT file that’s larger than available memory
Date Written: 6/18/90
Last reviewed: 9/24/91

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).

Getting a single scan line from a PICT file
Date Written: 6/18/90
Last reviewed: 9/24/91

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.

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.

Direct RGB PICT file compression

Date Written: 10/24/90
Last reviewed: 2/14/91

How are bits packed in direct RGB PICT files created by 32-Bit QuickDraw? I looked at the Macintosh Technical Note “Things You Wanted to Know About _PackBits…”, but this run-length encoded compression is clearly inefficient for cases where pixelSize is greater than 8 bits. I write software for machines other than Macintosh that decodes PICT files; therefore, I cannot issue any QuickDraw calls such as _unpackBits.

___

You’re quite right; compressing direct pixels using straight run-length encoding doesn’t work very well. Fortunately, direct pixel maps aren’t compressed this way. Compression schemes are discussed in *Inside Macintosh* Volume VI in the section titled “The New OpCodes: Expanded Format.”

In short, if the packType field holds the value 1, then no compression is done at all. The complete pixel image is saved in the PICT. If the packType field holds the value 2 and the pixel map is 32-bits per pixel, then all that’s done is that the alpha-channel byte is removed. So this:

```
00 FF FF FF  00 FF FF FF
```

is compressed to:

```
FF FF FF  FF FF FF
```

If the packType field holds the value 3 and the pixel map is 16 bits per pixel, then run-length encoding is done, but not through PackBits. Instead, a run-length encoding algorithm private to QuickDraw is used. This algorithm is very similar to PackBits, but where PackBits compresses runs of bytes, this routine compresses runs of words. The format of the resulting data is exactly the same as described in the Technical Note “Things You Wanted to Know About _PackBits…”, but you’ll get words instead. To build on the example in this Tech Note, lets say the 16-bit pixel image begins with these pixel values:

```
   AAAA AAAA AAAA 8080 0000 2A2A AAAA AAAA AAAA AAAA 8080 0000
   2A2A 2222 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
```

```
After being packed by QuickDraw’s internal compression routine, this becomes:
```

```
   FE AAAA
   02 8080 0000 2A2A
   FD AAAA
   03 8080 0000 2A2A 2222
   F7 AAAA
```

or

```
   FEAA AA02 8080 0000 2A2A FDAA AA03 8080 0000 2A2A 2222 F7AA AA
   *      *                 *      *                      *
```

where the asterisks mark the flag-counter bytes. Notice that you can’t assume that the pixel values are word-aligned. PackBits packs data 127 bytes at a time, though it will do this for up to 32,767 total bytes. Similarly, the internal compression routine packs data 127 words at a time.

If the packType field holds the value 4 and the pixel map is 32-bits per pixel, then run-length encoding via PackBits is done, but only after some preprocessing is done. QuickDraw first rearranges the color components of the pixels so that each color component of every pixel is consecutive. So the following four pixels (the row below the pixel values indicates a = alpha channel, r = red, g = green, b = blue, and the pixel offset):

```
            00 FF FF FF  00 FF C0 00  00 FF 80 00  00 C0 80 00
            a0 r0 g0 b0  a1 r1 g1 b1  a2 r2 g2 b2  a3 r3 g3 b3
```

is rearranged to become:

```
            FF FF FF C0  FF C0 80 80  FF 00 00 00
            r0 r1 r2 r3  g0 g1 g2 g3  b0 b1 b2 b3
```

The first four bytes are the red components of the four pixels, the next four bytes indicate the green components of the four pixels, and so on. The alpha channel isn’t included unless the cmpCount field contains 4 rather than the normal 3. If cmpCount contains 4, then all the alpha channel bytes are placed before the red bytes. Once this is done, then PackBits is called to compress the rearranged data.

Those are the only four compression schemes (including no compression) that are supported for direct pixel maps in PICTs. As always, reading PICTs yourself puts you in danger of not being able to read PICTs generated by future versions of QuickDraw. For compatibility reasons, these compression algorithms as I’ve described them probably can’t change in the future. It’s possible that new values for packType could be implemented though.

X-Ref:
Macintosh Technical Note “Things You Wanted to Know About _PackBits…”

Saving 32-bit Macintosh PICTs

Date Written: 10/30/90
Last reviewed: 2/20/91

I am using a packType=4 and have several questions about saving Macintosh 32-bit images in a PICT format:

1. What are the ramifications of using cmpCount=3 as opposed to cmpCount=4?

2. How is the pixData actually stored? According to several references, each line is run-length encoded; that is, [byteCount][data]. If rowBytes>250 then byteCount is a word. After looking at several examples, I came to the conclusion that you are actually using PackBits and that the term “run-length” encoded is a misnomer.

___

1. The difference between using cmpCount=3 or cmpCount=4 is that in the first case only the R, G, and B values are stored in the picture; in the second case QuickDraw stores in the picture the alpha channel plus R, G, and B. cmpCount=4 can be used when it is important to also save the alpha channel (as when you have some flags stored there).

2. Unless rowBytes is less than 8, the pixmap is compressed using PackBits, and the length is a word or a byte depending on rowBytes, but it always refers to the number of bytes comprising one scan line. My guess is that “run-length encoding” refers to the fact that pictures have data organized one row at a time.

To show how direct RGB pixmaps are stored inside a picture, I am including the decoding of a picture that I created to show how the different packing schemes change the resulting pixmap opcode data. First I created a direct RGB pixmap and drew three lines into it. The first line is 8 pixels long with a color of {0x1111,0xAAAA, 0x7777}, the third line is 8 pixels of {0xFFFF, 0x3333, 0xBBBB} and the line in between is two pixels of the first color then two pixels of the second color and so on until you make 8 pixels. Then I created a picture made of CopyBits calls copying the same lines but using different packing schemes. Finally I dumped the contents and the result is what you see here. I have put comments I think help make it clear how the packing scheme of choice changes the results.

I recommend that you have the *Inside Macintosh* Volume V QuickDraw chapter at hand to check the opcodes and data associated with them. If you had *Inside Macintosh* Volume VI then you wouldn’t need to also check the 32-Bit QuickDraw docs for the direct RGB pixmap opcodes. The Technical Note “Things You Wanted To Know about _PackBits…” gives details on how the packed data is arranged.

```
0256 0000 0000 0008 0008                    /* size and rect        */

0011 02FF 0C00 FFFF FFFF                    /* PICT2 Header         */
0000 0000 0000 0000 0008
0000 0008 0000 0000 0000
```

```
001E                                    /* Default hilite       */

001A FFFF 3333 BBBB                     /* RGB Fore Color       */

0001 000A 8001 8001 7FFF 7FFF           /* ClipRgn              */
```

```
009A              /* direct pixels opcode; see IM VI QD chapter for details */
0000 00FF 8020 0000 0000 0003 0008 0000    /* See IM V or VI QuickDraw chapter
*/
0001 0000 0000 0048 0000 0048 0000 0010    /* packType 0001 = no packing */
0020 0003 0008 0000 0000 006D A7DC 0000
0000 0000 0000 0003 0008 0000 0000 0003
0008 0000

0011 AA77 0011 AA77 0011 AA77 0011 AA77    /* line one xRGB after xRGB */
0011 AA77 0011 AA77 0011 AA77 0011 AA77

0011 AA77 0011 AA77 00FF 33BB 00FF 33BB    /* second line same */
0011 AA77 0011 AA77 00FF 33BB 00FF 33BB

00FF 33BB 00FF 33BB 00FF 33BB 00FF 33BB    /* third line same  */
00FF 33BB 00FF 33BB 00FF 33BB 00FF 33BB

009A                                       /* same direct pixels opcode   */
0000 00FF 8020 0000 0000 0003 0008 0000
0002 0000 0000 0048 0000 0048 0000 0010    /* packType 2 = fourth byte off*/
0020 0003 0008 0000 0000 006D A7DC 0000
0000 0000 0000 0003 0008 0003 0000 0006
0008 0000

11AA 7711 AA77 11AA 7711 AA77              /* one line of RGB after RGB   */
11AA 7711 AA77 11AA 7711 AA77              /* compare with previous       */

11AA 7711 AA77 FF33 BBFF 33BB              /* same here*/
11AA 7711 AA77 FF33 BBFF 33BB

FF33 BBFF 33BB FF33 BBFF 33BB              /* and here */
FF33 BBFF 33BB FF33 BBFF 33BB

009A                                       /* bits opcode */
0000 00FF 8020 0000 0000 0003 0008 0000
0004 0000 0000 0048 0000 0048 0000 0010    /* packType 4 = pack     */
0020 0003 0008 0000 0000 006D A7DC 0000    /* componentCount = 3    */
0000 0000 0000 0003 0008 0006 0000 0009    /* R,G and B separated   */
0008 0000

                   /* for details on packed data see Tech Note referenced above*/
06                  /* first line made of 6 bytes including count    */
F9 11F9 AAF9 77     /* -(-7)+1 of 11, -(-7)+1 of AA, and
                                  -(-7)+1 of 77 -> 8 RGB triplets   */
19                                  /* second row made of $19 bytes */
1711 11FF FF11 11FF FFAA AA33 33AA AA33 3377 77BB BB77 77BB BB
                                        /* $17+1 unpacked values */
06                                         /* third line same as first      */
F9FF F933 F9BB                             /* repeated R and B and G values */

009A                                          /* bits opcode again */
0000 00FF 8020 0000 0000 0003 0008 0000
0004 0000 0000 0048 0000 0048 0000 0010       /* same packing = 4 but       */
0020 0004 0008 0000 0000 006D A7DC 0000       /* component count = 4        */
0000 0000 0000 0003 0008 0009 0000 000C       /* alpha channel, R, G, and B */
0008 0000

08                                            /* first line */
F9 00F9 11F9 AAF9 77
             /* same as before but packing the high byte (00 value) also */
1B    /* second line first has -(-7)+1 '00' and then the same line as above */
F900 1711 11FF FF11 11FF FFAA AA33 33AA AA33 3377 77BB BB77 77BB BB
```

```
08                                              /* third line same as first */
F900 F9FF F933 F9BB
00FF                                            /* end of PICT */
```

A section of the code that produces this picture follows:

```
   RGBColor    oneColor = {0x1111, 0xaaaa, 0x7777},
               twoColor = {0xffff,0x3333,0xbbbb};

   SetGWorld(GgwPtr, nil);                     /* set the off-screen GWorld */

   if ( LockPixels(GgwPtr -> portPixMap) )
   {

       EraseRect(&(GgwPtr->portRect));

       RGBForeColor(&oneColor);
       MoveTo(0,0);
       LineTo(7,0);
       MoveTo(0,1);
       LineTo(1,1);
       RGBForeColor(&twoColor);
       MoveTo(2,1);
       LineTo(3,1);
       RGBForeColor(&oneColor);
       MoveTo(4,1);
       LineTo(5,1);
       RGBForeColor(&twoColor);
       MoveTo(6,1);
       LineTo(7,1);
       MoveTo(0,2);
       LineTo(7,2);
       SetRect(&localR,0,0,8,3);
       rr = localR;
       p = OpenPicture(&(GgwPtr->portRect));
       /* first no packing */
       (*(GgwPtr -> portPixMap)) -> packType = 1;
           CopyBits((*(GgwPtr -> portPixMap)),(*(GgwPtr -> portPixMap)),
               &rr,&localR,srcCopy,nil);
       /* second pack 2: remove alpha chanel */
           OffsetRect(&localR,0,3);
       (*(GgwPtr -> portPixMap)) -> packType = 2;
           CopyBits((*(GgwPtr -> portPixMap)),(*(GgwPtr -> portPixMap)),
               &rr,&localR,srcCopy,nil);
       /* third pack 4: packing component by component */
           OffsetRect(&localR,0,3);
       (*(GgwPtr -> portPixMap)) -> packType = 4;
           CopyBits((*(GgwPtr -> portPixMap)),(*(GgwPtr -> portPixMap)),
               &rr,&localR,srcCopy,nil);
       /* The last case is pack 4 but storing the alpha channel also */
           OffsetRect(&localR,0,3);
       (*(GgwPtr -> portPixMap)) -> packType = 4;
       (*(GgwPtr -> portPixMap)) -> cmpCount = 4;
           CopyBits((*(GgwPtr -> portPixMap)),(*(GgwPtr -> portPixMap)),
               &rr,&localR,srcCopy,nil);
       ClosePicture();
   UnlockPixels(GgwPtr -> portPixMap);
   }
```

X-Ref:
“Color QuickDraw,” *Inside Macintosh* Volumes V and VI
Macintosh Technical Note “Things You Wanted To Know about _PackBits…”

BitMapToRgn for non-Color QuickDraw Macintosh models
Date Written: 11/9/90
Last reviewed: 2/20/91

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
AppleLink: SW.LICENSE
Phone:(408) 974-4667

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.

Color and non-Color QuickDraw trap dispatch differences
Date Written: 1/28/91

Last reviewed: 2/13/91

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.

Macintosh OpenCPicture 72-dpi calculation bug

Date Written: 2/12/91
Last reviewed: 2/20/91

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.

GetGWorldPixMap bug and workaround

Date Written: 3/12/91

Last reviewed: 10/9/91

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.

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.

Using dithering and animation on the same Macintosh image

Date Written: 6/19/91
Last reviewed: 10/15/91

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 PICT from a srcCopy to a ditherCopy. 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.

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!

Highlighting ignored if foreground same as background color

Date Written: 8/7/91
Last reviewed: 9/24/91

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.

Gestalt 'qdrw' selector bug and workaround
Date Written: 8/1/91
Last reviewed: 10/9/91

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.

Version 2 PICTs on pre-Color QuickDraw models
Date Written: 8/13/91
Last reviewed: 10/22/91

Inside Macintosh Volume V says a System 4.1 (and later) patch ensures that version 2 PICTs are displayed correctly on earlier machines that don't have Color QuickDraw, such as the Macintosh Plus and SE. However, my version 2 PICT, consisting primarily of a pixmap (opcode = $90 since rowBytes <8) displays correctly on a Macintosh IIfx but displays garbage on a Macintosh SE.

___

The PICT problem you reported is caused by a bug in 6.0.7 QuickDraw. The workaround for the time being is to:

• Use System 7.0, where it's been fixed, or
• Don't use opcode $90. Instead, use padding so that opcode $98 can be used. (Opcode $98 is the packed CopyBits version that works for rowBytes >= 8.). You can do this by creating the picture with a pixmap that's wider than you actually need, and then use the clip region to clip out the part you don't need.

GetPixelsState is slow sometimes
Date Written: 8/27/91
Last reviewed: 9/24/91

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.

OpenCPicture and PICTs other than 72 dpi

Date Written: 10/2/91
Last reviewed: 10/8/91

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.)

No System 7 QuickDraw alpha channel support
Date Written: 10/23/91
Last reviewed: 11/27/91

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.

BitsToRgn and MPW BitMapToRegionGlue

Date Written: 10/29/91
Last reviewed: 12/11/91

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.

Ensuring even rowBytes for 'cicn' resources

Date Written: 12/4/91
Last reviewed: 1/27/92

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.

CopyBits blend mode: OpColor's affect & eliminating banding

Date Written: 12/11/91
Last reviewed: 2/17/92

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.

Icon dimming under System 7 and System 6

Date Written: 1/6/92
Last reviewed: 2/6/92

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.

QuickDraw out of memory if debugger invoked by “Jackson”
Date Written: 3/11/92
Last reviewed: 4/7/92

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...

```
       _Debugger ; Hey! sysError came back! Better drop into the debugger
```

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.

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.

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:

```
    BitMapPtr  =  ^BitMap
```

then use it to coerce the pixmap as follows:

```
    SeedCFill(BitMapPtr(@myPixMap)^, ...);
```

If you have a PixMapHandle, do the following:

```
    SeedCFill(BitMapPtr(myPixMapHandle^)^, ...);
```

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).

Macintosh Color QuickDraw and packed PICTs

Date Written: 1/1/90
Last reviewed: 12/7/90

Does Macintosh 32-Bit QuickDraw support packed PICTs? What's the technique for saving packed PICT formats? What compression schemes are supported?

___

Color QuickDraw has always supported packed PICTs. See *Inside Macintosh* Volume V for details on how CLUT pixmaps are packed. Under 32-Bit QuickDraw, to pack direct RGB pixmaps in PICTs, call CopyBits with the packType field in the source pixmap set to one of the following constants that apply to direct RGB pixmaps:

0—default packing (pixelSize 16 defaults to packType 3 and pixelSize 32 defaults to packType 4)
1—no packing
2—remove pad byte (32-bit pixels only)
3—run-length encoding by pixel size chunks (16-bit pixels only)

4—run-length encoding, all of one component at the time, one scan line at a time (24-bit pixels only)

Scheme 4 will store the alpha channel also if cmpCount is set to to 4. PackSize is not used and should be set to 0 for compatibility reasons. These are the only compression techniques supported at this time.

Macintosh PICT color picture file format
Date Written: 1/1/90
Last reviewed: 11/21/90

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 Tech Note “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”

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”

Using Macintosh System 7 OpenCPicture for higher resolution
Date Written: 1/1/90
Last reviewed: 12/7/90

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.

How to identify 32-Bit QuickDraw version
Date Written: 1/1/90
Last reviewed: 11/21/90

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");
}
```

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.

Macintosh CopyBits transfer modes changed for System 7

Date Written: 1/1/90
Last reviewed: 12/7/90

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.

Which QuickDraw versions support SetEntries
Date Written: 3/3/92
Last reviewed: 6/30/92

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.

Macintosh pixel map maximum rowBytes change

Date Written: 4/22/91
Last reviewed: 6/10/91

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 $3FFF or less applies only to 32-bit QuickDraw. Using pixmaps with rowBytes greater than $1FFF 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.

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.

Construct a 'clut' instead of changing b/w palette entries
Date Written: 6/10/91
Last reviewed: 8/1/91

How can I change the first and last (white and black) entries in a Macintosh palette?

___

The answer to your question about changing the black and white entries in a palette is a little strange. You can't simply change the palette associated with an on-screen window, because too many portions of the Toolbox/OS assume that the first entry is white and the last entry is black.

However, what you *can* do is create an off-screen GWorld and construct a 'clut' for it that does what you want. Creating the 'clut' is fairly straightforward:

```
/* Making a reversed gray-scale color table */

CTabHandle offColors;

offColors = (CTabHandle) NewHandleClear (sizeof (ColorTable) + 255 *
             sizeof (ColorSpec));
(**offColors).ctSize = 255;
for (index = 0; index <= (**offColors).ctSize; index++)
 {
   (**offColors).ctTable [index].value = index;
   (**offColors).ctTable [index].rgb.red = (index << 8) | index;
   (**offColors).ctTable [index].rgb.green = (index << 8) | index;
   (**offColors).ctTable [index].rgb.blue = (index << 8) | index;
 }
(**offColors).ctFlags = 0;
(**offColors).ctSeed = GetCTSeed ();
```

Note that using this 'clut' with an off-screen GWorld will work fine *except* if you attempt to draw text into the GWorld. Apparently drawing text off screen carries the same assumptions that all drawing does on screen.

Once you have done your off-screen drawing with the reversed 'clut' as described above, all that remains is to CopyBits from your off-screen GWorld to your on-screen window. Your on-screen window will need the appropriate palette. Fortunately, constructing that palette from the 'clut' is trivial:

```
/* Make a palette out of it */
aPalette = NewPalette (offColors, (**offColors).ctSize + 1, pmTolerant, 0);
```

Attaching this palette to the window will cause the correct remapping to occur when you CopyBits from the GWorld to the window, and everything should look just fine.

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) forPlotCIcon 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.

CopyBits maps source pixmap colors to GDevice inverse table
Date Written: 4/3/91
Last reviewed: 6/17/91

I’m trying to draw off screen so I made my own CGrafPort, pixmap, and color table, but when I draw into it, the colors come out all wrong. What’s going on here?

___

It’s a very common misconception that CopyBits maps the colors available in the source pixmap’s color table to the colors available in the destination pixmap’s color table.

What actually happens is that CopyBits maps the colors in the source pixmap to the colors available in the current GDevice’s inverse table. See *Inside Macintosh* Volume V, pages 137 through 139 for a description of inverse tables. Inverse tables is a backwards color table. With a color table, you use a pixel value as an index into the table to return a color. With an inverse table, you use a color-like value as an index into the table to return a pixel value. When CopyBits maps colors from one pixel map to another, it takes a source pixel value, uses the source pixmap’s color table to get the corresponding color, and uses that color as an index into the current GDevice’s inverse table to get the pixel value of the closest color in the destination pixmap’s color table.

Generally speaking, every conceivable color table has exactly one conceivable inverse table. If you alter the contents of a color table, then the inverse table must likewise be altered. It's just like numbers: for any number you give me, I can give you its negative. For any color table you give me, MakeITable can give you its inverse table. MakeITable is documented in *Inside Macintosh* Volume V, page 142.

Think of the current GDevice as an implied parameter to CopyBits. If you don't deal with GDevices at all, then the current GDevice is always the main screen's GDevice as far as you're concerned. Color QuickDraw often switches between different screens' GDevices so that you can draw to multiple screens, but that's all handled behind your back. If you create a pixmap, give it a color table, and CopyBits to it, then the main screen's GDevice's inverse table is used to map colors from the source pixmap. That's OK as long as your destination pixmap's color table is compatible with the main screen's inverse table. If you change the depth and/or color table of the main screen, and then still CopyBits to this same pixmap with the same old depth and color table, then things won't work correctly because the main screen's GDevice's inverse table changes, making it incompatible with your destination pixmap's color table. This problem usually manifests itself as incorrect colors, but it can result in crashes.

To fix this, you'll have to remove your reliance on the main screen's GDevice. To do that you'll have to make your own GDevice. There's a routine called NewGDevice, but it always makes the GDevice in the system heap. Instead, you should just call NewHandle to allocate a GDevice record yourself. Here's what the fields should hold:

gdRefNum—The GDevice has no driver, so set this to 0.

gdID—It doesn't matter what this is set to—might as well set it to 0.

gdType—Set this to 2 if the off-screen uses direct colors (16 or 32 bits per pixel) or 0 if the off-screen uses a color table (1 through 8 bits per pixel).

gdITable—Allocate a small (maybe just a 2-byte) handle for this field. After you're done setting up this GDevice and your off-screen pixmap, color table (if any), and CGrafPort, then set this GDevice as the current GDevice by calling SetGDevice. Then call MakeITable, passing it NIL for both the color table and inverse table parameters, and 0 for the preferred inverse table resolution.

gdResPref—I'd guess that more than 99.9 percent of all inverse tables out there have a resolution of 4. Unless you have some reason not to, I'd recommend the same here.

gdSearchProc—Set to NIL. Use AddSearch if you want to use a SearchProc.

gdCompProc—Set to NIL. Use AddComp if you want to use a CompProc.

gdFlags—Set to 0 initially, and then use SetDeviceAttribute after you've set up the rest of this GDevice.

gdPMap—Set this to be a handle to your off-screen pixmap.

gdRefCon—Set this to whatever you want.

gdNextGD—Set this to nil.

gdRect—Set this to be equal to your off-screen pixmap's bounds.

gdMode—Set this to -1. This is intended for GDevices with drivers anyway.

gdCCBytes—Set to 0.

gdCCDepth - Set to 0.

gdCCXData - Set to 0.

gdCCXMask - Set to 0.

gdReserved - Set to 0.

For gdFlags, you should use SetDeviceAttribute to set the noDriver bit and the gDevType bit. You should set the gDevType bit to 1 even if you have a monochrome color table. The 0 setting was only used when monochrome mode was handled by the video driver, and 32-Bit QuickDraw eliminated that convention. Your GDevice doesn't have a driver anyway.

Once this is done, the GDevice and the off-screen pixmap should be treated as inseparable. When you CopyBits or draws into the pixmap, first call SetGDevice to set its GDevice as the current GDevice. When that's done, call SetGDevice to restore the previous GDevice. Doing this insulates you from changes in a screen's GDevice.

If you alter your pixmap's color table, you should make sure you update the ctSeed of that color table, either by assigning to it the result of GetCTSeed (documented on page 143 of *Inside Macintosh* Volume V) or by passing a handle to the color table to CTabChanged. The next time that pixmap is drawn into, Color QuickDraw will update your GDevice's inverse table automatically when it realizes that the ctSeed is different from the current GDevice (which had better be yours) inverse table's iTabSeed.

GWorlds work this way on Color QuickDraw machines. Every GWorld comes with a pixmap and a GDevice. When you call SetGWorld, that sets both the GWorld and its GDevice as current.

How Macintosh system draws small color icons

Date Written: 3/31/92
Last reviewed: 5/21/92

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.

Spooling and preserving Macintosh QuickDraw pixmap depth

Date Written: 2/11/92
Last reviewed: 9/15/92

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.

Determining the resolution of a PICT

Date Written: 6/10/92
Last reviewed: 9/15/92

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;
}
```