Technical Note QD505

Basic 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]

Determining whether `CopyBits` to a PICT was successful

Date Written: 1/7/93

Last reviewed: 3/1/93

If I'm recording a PICT and doing a `CopyBits` of a really big image into the PICT, how can I determine whether I'm out of memory?

The only reliable way to see whether a `CopyBits` to a PICT succeeded is after the fact. You need to test the PICT's `picFrame rect` (*Inside Macintosh* Volume V, page V-87) to see whether it's empty after the `CopyBits`. The test would look like:

```
If (EmptyRect(&(**myPicture).picFrame))
```

Back to top

Random function requires prior `InitGraf rect` call

Date Written: 12/16/92

Last reviewed: 3/1/93

The `Random` function call listed in QuickDraw.h can't be called from MPW tools without crashing my system. It appears to work when the function is called from applications or cdevs. What could be causing this problem?

Use SANE's `RandomX` function instead of QuickDraw's `Random` function if possible because it gives you better randomness. If you do use QuickDraw's `Random` function, be sure to call `InitGraf rect` before calling `Random` from any application or tool. `InitGraf rect` initializes a set of QuickDraw global variables for use with the QuickDraw tools; these globals must be initialized because the `Random` function uses one of them as a seed to generate the random number.

Normally, it's not good practice to call initialization routines from within an MPW tool, but calling InitGraf rect is OK. For more information on which initialization routines are OK to call and which ones aren't, see page 7 in the MPW Tools chapter of Building and Managing Programs in MPW.

Sometimes it isn't obvious when you need to call InitGraf rect before using the Random function. For example, if you're using the Macintosh serial tool in a faceless background application, you'll need to initialize QuickDraw because the tool calls Random.

Back to top

GetIconCacheData rect & SetIconCacheData rect bug and workaround

Date Written: 12/8/92

Last reviewed: 3/1/93

The Icon Utilities routine GetIconCacheData rect leaves two bytes of garbage on the stack. This surfaced as a problem for me because it prevented a saved register from getting restored properly. SetIconCacheData rect probably has the same problem, since it calls the same trap internally. I solved the problem by encapsulating GetIconCacheData rect within my own static function, like so:

```
static OSErr _GetIconCacheData rect( Handle theCache, void **theData )
{
return  GetIconCacheData rect( theCache, theData );
}
#define  GetIconCacheData rect  _GetIconCacheData rect
```

I then call GetIconCacheData rect normally. The #define redirects my call to my static wrapper function. The extra two bytes on the stack are recovered when the wrapper function UNLKs and returns. This method has the advantage that the calling code will still work even after the trap is fixed. Am I correct?

You're quite correct; this is a bug in GetIconCacheData rect and SetIconCacheData rect. Here's the offending code from the source:

```
EXIT    MOVEA.L    (SP)+, A0   ; Pop return address into A0
        ADDQ.L     #6, SP      ; Point stack at return value
        MOVE.W     D0, (SP)    ; Put return value on the stack
        JMP       (A0)         ; Return
```

As you can see, the exit routine is adding 6 to the stack to clear up the input parameters instead of 8 (handle and handle), so an extra word of garbage is being left on the stack. Thanks for letting us know about the problem.

Back to top

DrawText and DrawString patched to be script aware

Date Written: 11/16/92

Last reviewed: 6/14/93

While localizing our software, we were told not to assume that a character is only one byte, and thus not to use DrawChar. Does this mean that we can't use DrawText or DrawString?

DrawChar takes a one-byte character as a parameter, so it isn't suitable for drawing a character whose internal representation requires two bytes. However, DrawText and DrawString (both end up in the same bottleneck procedure StdText) are patched in script-aware systems, and do recognize whether a given byte in a given font-script still corresponds to a one-byte character, or is the first byte of a two-byte character. In the latter case, it transparently fetches the next byte, and looks up the right glyph encoded by a double byte, before actually drawing the glyph.

Back to top

QuickDraw globals at INIT time

Date Written: 6/1/92

Last reviewed: 9/15/92

If I call InitGraf rect before I reference CurrentA5, will CurrentA5 be valid and can the QuickDraw globals be referenced off it? The screenBits bounds values seem screwy on some machines. Does the problem lie with CurrentA5? Should I be referencing A5?

Here's the process used by ShowINIT, which is remarkably compatible with system software and other INITs (and it had better be, because it's used by more than half the system extensions available):

1. It saves the value in the CurrentA5 global to restore it later.
2. It points the A5 register at 4 bytes of storage for use by the system.
3. It copies the value now in A5 into the CurrentA5 global.
4. It calls InitGraf rect, passing a pointer to the thePort field of a QuickDraw globals structure.
5. It opens a port and draws as necessary. [This is where all the functionality goes.]
6. After it's done, it closes its port.
7. It copies the value saved in step (1) into the A5 register.
8. It copies the restored A5 value into the CurrentA5 global.

To summarize, ShowINIT saves the A5, creates and initializes its own A5 world, does its drawing, then restores the previous A5 world. For more information on this subject, see the Macintosh Technical Note "Stand-Alone Code."

Back to top

Macintosh QuickDraw LineTo bug and workaround

Date Written: 4/23/92

Last reviewed: 7/13/92

Our zooming function crashes into flames when we pass valid coordinate values to LineTo, as in the following example:

```
    SetPort(myPort);
    MoveTo(154,31619);
    LineTo(74, -31742); (* You are dead! *)
```

What can we do to avoid LineTo crashes like this?

The QuickDraw Engineering group is aware of the problem you described. The bug probably is going to be fixed in the next release that includes bug fixes. Given that waiting for a system solution may demand more patience than is reasonable, you may want to consider including in your software some form of workaround that will prevent your users from crashing every time an operation takes the software to the limits of QuickDraw.

One way to approach this problem is to replace the lineProc bottleneck. All you need to do is to check the distance between the current pen position and the line's end, and when the distance becomes too big (let's say more than 32000) your procedure will call StdLine a couple of times, splitting the operation in two.

Replacing the bottlenecks is a very straightforward operation (which you are probably already using) and in most of the cases will only result in another level of indirection into StdLine but that will prevent your program from calling QuickDraw with parameters that are guaranteed to cause crashes.

Back to top

Use crsrNew flag to unobscure cursor without mouse move

Date Written: 3/3/92

Last reviewed: 6/14/93

The Macintosh QuickDraw routine `ObscureCursor` hides the cursor until the next time the mouse is moved, but it isn't affected by `HideCursor` or `ShowCursor`. Our application needs to use `ObscureCursor` while the user is typing but needs the cursor to be visible after no typing has occurred for a short period. How do we "undo" `ObscureCursor`, since we can't rely on the user to move the mouse?

The only way (besides actual mouse movement) to make an obscured cursor visible again is to convince the system that the mouse has moved. There's no really good way to do this via Toolbox calls, so you're going to have to do it the hard way and simply update the low-memory cursor information to tell the system the cursor moved (even though you don't need to update the actual position).

To tell the system the cursor has changed location, simply set the `crsrNew` flag (a byte located at $`08CE`) to 1. When the system sees this byte is 1, it will assume the cursor has moved and redraw the unobscured cursor at the appropriate place (where it was all along), and reset `CrsrNew`, waiting for the mouse to move again.

Back to top

Macintosh CalcMask and CopyMask code sample

Date Written: 2/27/92

Last reviewed: 6/14/93

I can't get the black-and-white version of my lasso-type tool to work correctly with `CalcMask` and `CopyMask`. With `CalcCMask` it seems to work fine. What could I be doing wrong?

`CalcMask` and `CalcCMask` are similar in that they both generate a 1-bit mask given a source bitmap. With `CalcCMask`, though, a pixMap can be used in place of the source bitmap; the `seedRGB` determines which color sets the bits in the mask image. An easy mistake to make is to forget that `CalcCMask` accepts a pointer to a bitmap data structure while `CalcMask` expects a pointer to the actual bit image. And unlike `CalcCMask`, which uses bounding `Rects` for the image's dimensions, `CalcMask` uses the bitmap's `rowBytes` and pixel image offsets to determine the bounding Rects for the image. A typical call to these routines would be:

```
    BitMap source, mask;
    CalcMask (source.baseAddr, mask.baseAddr, source.rowBytes,
              mask.rowBytes, source.bounds.bottom-source.bounds.top,
              source.rowBytes>>1);
    CalcCMask (&source, &mask, &(*source).bounds, &(*mask).bounds,
               &seedRGB, nil, 0);
```

One last thing to note when using `CalcMask` is that the width of the image is in words and not bytes. To learn more about these routines, see page 24 of *Inside Macintosh* Volume IV and page 72 of *Inside Macintosh* Volume V. Also, the *Developer CD Series* disc contains a sample, CalcCMask&CalcMask, that shows how to use both these routines.

Back to top

Code for filling an area fully bounded by polygon

Date Written: 2/21/92

Last reviewed: 6/14/93

Currently, when a polygon is filled, an even-odd rule is applied to determine which areas of the polygon are to be filled. For our application, we also need to fill all the areas of the defined polygon. Is there a relatively easy way to accomplish this?

There are many different ways to fill polygons, as you may know. If you do not want to use QuickDraw's standard FillPoly routine, you'll have to create your own. The following sample illustrates one technique that might be used to fill the area fully bounded by a polygon. It can be dropped right into the traffic light sample (sample.p) that ships with MPW as a replacement for its DrawWindow procedure. The green star is drawn using FillPoly and the black star is drawn using my filling technique that uses an offscreen bitmap and calcMask to fill in the poly the desired way, then CopyBits to transfer it to the onscreen port. The drawbacks of this method are that it is not as fast as writing a specialized poly routine; the benefits are that it's small, fast enough for most operations, and can be used for more than just polygons.

```
{$S Main}
PROCEDURE DrawWindow(window: WindowPtr);

var        MyPoly:PolyHandle;
        MyRgn :RgnHandle;
        OffPort,OnPort:GrafPtr;

    Function    CreateOffport(VAR newOffscreen:grafPtr;
inBounds:Rect):Boolean;

    var    SavePort,NewPort:Grafptr;

    begin
        GetPort(SavePort);
        NewPort:=GrafPtr(NewPtr(sizeof(grafport)));
        If MemError<>noErr then Begin
            CreateOffport:=false;
            EXIT(CreateOffport);
        END;

        OpenPort(newPort);
        With newPort^ do begin
            portRect :=Inbounds;
            RectRgn(ClipRgn,inBounds);
            RectRgn(visRgn, inBounds);
        End;

        With newPort^.PortBits DO BEGIN
            Bounds:=Inbounds;
            rowBytes:= ((inBounds.right-inBounds.Left+15) DIV 16) *2;
            baseAddr:= NewPtr(rowBytes
                            * LONGINT(inBounds.Bottom-inBounds.Top));
        End;
        If MemError <>noErr THEN BEGIN
            SetPort(SavePort);
            ClosePort(newPort);
            DisposPtr(ptr(newPort));
            CreateOffport:=false;
          END
        ELSE  BEGIN
            EraseRect(inBounds);
            newOffscreen :=newPort;
            setPort(SavePort);
            CreateOffPort:=true;
        end;
    end;

    Procedure    KillOffPort(oldOffscreen :GrafPtr);
    Begin
        ClosePort(oldOffscreen);
        DisposPtr(OldOffscreen^.portBits.baseAddr);
        DisposPtr(ptr(OldOffScreen));
    End;

BEGIN
    If NOT (CreateOffPort(offPort,window^.portRect)) THEN Exit(DrawWindow);
    If NOT (CreateOffPort(onPort,window^.portRect)) THEN Exit(DrawWindow);

    SetPort(window);

    MyRgn:=NewRgn;
    OpenRgn;
        MoveTo(10,25);
        Lineto(70,25);
        Lineto(15,70);
        Lineto(40,10);
        Lineto(65,70);
        Lineto(10,25);
    CloseRgn(MyRgn);

    MyPoly:=OpenPoly;
        MoveTo(10,25);
        Lineto(70,25);
        Lineto(15,70);
        Lineto(40,10);
        Lineto(65,70);
        Lineto(10,25);
    ClosePoly;
    OffsetPoly(MyPoly,0,100);

    SetPort(OffPort);
    FramePoly(MyPoly);
    { Now "Fill the poly" the right way }
    CalcMask(    Offport^.portBits.BaseAddr,OnPort^.portBits.BaseAddr,
                OffPort^.portBits.RowBytes, OnPort^.portBits.RowBytes,
                OffPort^.portRect.bottom-OnPort^.portRect.Top,
                OffPort^.portBits.RowBytes DIV 2);
    SetPort(OnPort);
    SetPort(Window);

    If gStopped then
        CopyBits(    OnPort^.portBits, Window^.portBits,
                    OnPort^.portRect, Window^.portRect, srcCopy, NIL)
    ELSE
        CopyBits(    OffPort^.portBits, Window^.portBits,
                    OffPort^.portRect, Window^.portRect, srcCopy, NIL);

    IF gStopped THEN
      begin
        ForeColor(greenColor);
        FrameRgn(MyRgn);
      end
    ELSE
      begin
        ForeColor(greenColor);
        PaintRgn(MyRgn);
      end;
    ForeColor(blackColor);
    DisposeRgn(MyRgn);
    KillPoly(MyPoly);
    KillOffPort(Offport);
    KillOffPort(OnPort);
END; {DrawWindow}
```

Back to top

Inside Macintosh Vol. V PICT opcode size should be fixed

Date Written: 1/22/92

Last reviewed: 2/28/92

The definition of PICT version 2 on pages 92-105 of *Inside Macintosh* Volume V says that the data size of the opcodes `$001A` and `$001B` is variable, but also that the data is an `RGBColor`. This is confusing, since the size of an `RGBColor` is fixed at six bytes. How can these two opcodes vary in the amount of associated data?

Seems like you've run into a cut/paste problem. All the opcodes that refer to Table 4 are new for Color QuickDraw. Also, most of them are variable in length, so the author simply had a standard notation for anything that was explained further in table 4 (page V-103). The information contained in table 4 is, in fact, accurate. The size information of several of the opcodes listed is not variable even though the preceding pages told you they were.

All you gotta do is believe Table 4 and you will be fine.

Back to top

PICTs with PostScript PICT comments and memory use

Date Written: 1/10/92

Last reviewed: 6/14/93

Why does my PICT (including dotted lines) use so much memory when drawn in MacDraw, and even more when drawn in SuperPaint? Do they include `PicComments` for PostScript?

Your guess that it has to do with PicComments is quite right; both MacDraw and SuperPaint include a PostScript representation of the dotted (dashed) lines and some other graphic operations in the PICT, together with the QuickDraw commands. During printing, this allows the LaserWriter driver to take advantage of specific PostScript capabilities that are unavailable in QuickDraw, like primitives for dashed lines.

On the other hand, the PostScript representation for dashed lines is much shorter than the QuickDraw representation, which requires a (long, very long ...) sequence of `ShortLine` opcodes. So, another piece of explanation for the large PICT size basically is that QuickDraw does not have facilities to describe dotted lines in an economic way.

SuperPaint also includes a copy of a proprietary dictionary, which adds substantially to the size of a PICT. On the other hand, the code that resides in that dictionary makes the picture's PostScript representation that much better. Ultimately, WYSIWYG is the goal, and sometimes it takes a little extra code to make that happen. (Incidentally, the PostScript dictionary contained in pictures created by older versions of SuperPaint makes assumptions about the contents of the LaserPrep file which are not true for the recent versions of the LaserWriter driver. Documents containing such pictures will not print correctly any more.)

To determine the primitives that define other nonstandard QuickDraw objects found in drawing applications, you can use MPW's `DeRez` function or a third-party utility such as Palomar Software's PICT Detective on the resource PICT. These tools will provide the opcodes that define the PICT.

Back to top

Where CopyBits looks for memory to use

Date Written: 1/3/92

Last reviewed: 1/27/92

Where does CopyBits look for the memory it needs?

CopyBits checks the stack to determine if there is enough stack space for it to copy the whole image, which in some cases may be roughly up to 5 extra rowbytes of special effects per row, depending on what special effects such as dithering or scaling are being used. If there is not enough stack space for the whole image, CopyBits then tries for half the image, and keeps halving until it gets down to one row of the image (plus the room for the special effects rows). If there is not enough stack space for one row of the image, then CopyBits tries to allocate temporary memory.

Before allocating temporary memory, CopyBits checks if the temporary memory traps are available. (They are available under both System 6 MultiFinder and System 7.) If the traps are available, CopyBits tries to allocate a 256K byte buffer for use as a "fake" stack. (CopyBits used to try for a 64K block, but this has been changed, and it may change again.) If this succeeds, then all is well and the image is copied. If the temporary memory traps do not exist, or if CopyBits cannot allocate a 256K buffer, then the image is not copied and CopyBits returns.

CopyBits does not check in the application heap for free memory, at least not for its work buffer. For its work buffer it will only use the stack, and after that it resorts to temporary memory, if available. There are some circumstances that may cause memory allocations in the application heap, but this memory is not used for CopyBits's image buffer.

Also, please note that the implementation of CopyBits is subject to change in future versions of QuickDraw.

Back to top

GrafPort patStretch: valid values

Date Written: 12/19/91

Last reviewed: 6/14/93

I'd like to know more about that PatStretch field inside a GrafPort or CGrafPort. If I stuff a values in PatStretch(4) then nothing happens; prints look the same, even using a standard bottleneck. Please tell me how I can get this to work.

PatStretch only works with values of 2 or 3. With any other value, it defaults to no stretching. The "2" case was created because of the ImageWriter (72->144 dpi) situation. The "3" case was added to support the ImageWriter LQ and the AppleFax modem.

So why wasn't a "4" (72->300 dpi) handler added for the LaserWriter driver? Good question. Somehow or other it was decided that pattern stretching for the LaserWriter driver would be done completely by the driver itself. The LaserWriter driver actually does pattern stretching by using a pattern 4 times as large, rather than 4.17. In other words, it really scales the 72 dpi pattern to 288 dpi rather than 300 dpi. You may want to take a similar approach, since you'd only have to work with whole numbers this way.

So, if you want to do 4-times pattern stretching, you must scale the pattern yourself. If you copy the original pattern into an area that's twice as wide and twice as tall and use that, you should be all set. You'll need to use PrGeneral to set the printer to the appropriate resolution and CopyBits to copy the pattern into the object that needs to be filled, using the "cookie cutter" approach to fill the object.

X-Ref:*Inside Macintosh* Volume I, page I-150

Back to top

How to tell whether GetPictInfo is available

Date Written: 12/16/91

Last reviewed: 6/14/93

How do you determine whether the Picture Utilities Package function GetPictInfo is available? Gestalt doesn't seem to have the right stuff!

To determine whether the GetPictInfo routine is available, check the system version number with the Gestalt function. GetPictInfo is available in system software version 7.0 and later. Use the Gestalt selector gestaltSystemVersion to determine the version of the system currently running. Usually it's best not to rely on the system version to determine whether features are available, but in this case, it's the only way to determine whether the Picture Utilities Package is available.

For example, the following C function will determine whether the GetPictInfo call is available:

```
#include <GestaltEQU.h>

Boolean IsGetPictInfoAvail()

{

OSErr err;

long feature;

err = Gestalt(gestaltSystemVersion,&feature);

/* Check for System 7 and later */

return (feature >= 0x00000700);

}
```

In *Inside Macintosh* Volume VI, see page 3-42 for information on using Gestalt to check the system version number, and see page 18-3 for information on the Picture Utilities Package.

Back to top

Detecting whether application window is partially hidden

Date Written: 9/26/92

Last reviewed: 6/14/93

We draw directly to the screen to gain the fastest possible animation speed, and when we need compatibility--such as when windows overlap or for multiple screens--we do use CopyBits. How do we tell whether the window is hidden or that the visible part is not rectangular?

If your window is covered partially by another applications window or if your layer has been hidden by the process menu, the visRgn of your window's grafport will not be the portRect anymore. (Keep in mind that if you scroll by modifying the portRect of the grafport, then you'll have to do a more complex calculation...) Here is a small Pascal routine that returns this information:

```
Function UseCopyBits(thePort:grafptr):Boolean;

begin

     UseCopyBits:= NOT( (thePort^.VisRgn^^.rgnSize=10) and
        (thePort^.visRgn^^.RgnBBox=thePort^.PortRect) );
end;

The rect strucRgn^^.rgnBBox will be zero for a visible window if the system
has hidden the application.
```

Back to top

CopyBits bug and workaround

Date Written: 6/26/91

Last reviewed: 6/14/93

Has anyone run across what I'm told is a bug in CopyBits? It works like this: In the deep, dark workings of CopyBits, some routine tries to read the two bytes preceding the baseAddress of the source PixMap. If the baseAddress is at the start of a card's NuBus space and there isn't a card filling the adjacent space, this causes a bus error! Has anyone found a good workaround?

The short answer is: you're right. QuickDraw inadvertently reads from memory below the base address of a pixmap. The workaround is to place the video base address 32 bytes into the slot memory space for the card; if the card you're using doesn't have this workaround, there's nothing you can do other than making sure there's a card in the next-lower slot.

Back to top

Macintosh animation samples

Date Written: 11/6/91

Last reviewed: 6/14/93

Do you have an example of flicker-free animation on the Macintosh?

We have some good stuff that's written in MPW Pascal. It's DTS Sample Code #16, OffSample, and this uses some routines defined in DTS Sample Code #15, OffScreen. Also, the System 7.0 CD sample code folder contains a smaller sample called "GMonde" that uses GWorlds.

Back to top

System 7 QuickDrawDrawText performance

Date Written: 11/4/91

Last reviewed: 11/27/91

We've noticed that using DrawText is much slower in System 7, especially when drawing in color (anything other than black on white). What can be done to restore the drawing speed to System 6 levels?

A QuickDraw function like DrawString or DrawText will be slower under certain circumstances in System 7 than System 6. Specifically, if you are drawing in srcCopy mode and you colorize the text--that is, foreground color is not black and background color is not white *(Inside Macintosh* Volume VI, page 17-16)--then QuickDraw really slows down as you have noticed. *Sometimes,* the speed of drawing is 6 times as slow as System 6.

The cause of this slowness is a known System 7 bug. The bug has concerned the engineers greatly and will be responded to in an appropriate manner in the future.

There are a few workarounds: One, you can avoid using the `srcCopy` mode and use the default `srcOr` mode instead. However, this is not a real workaround, since you may have essential reasons to use `srcCopy`. The other option is to create an offscreen pixmap or `GWorld` and perform a `DrawText` with `srcOr` to this `GWorld` with colorization. Then, you can perform a `CopyBits` from the offscreen to the screen with `srcCopy` mode and no colorization. Using `CopyBits` will not cost you much time. Again, this is a workaround and is not ideal.

The srcOr is a bit slower than in System 6.0.x, but it does not have a bug; rather it is a side effect of system enhancements. The slow speed is a trade-off taken to receive the host of other benefits.

Back to top

Updating Macintosh cursor without mouse competition

Date Written: 6/12/91

Last reviewed: 6/14/93

How can I programmatically move the Macintosh mouse without the real mouse interfering?

The real answer to your question is twofold: First, you can do exactly what you want to do with the sample included below. *However,* this is not a good thing to do, it would be better if you took the solution used in Apple's Guided Tour disks: Always hide the cursor and then decouple the cursor from the mouse. Then, instead of using the system's cursor, simply draw your own "cursor" using QuickDraw and treat it as a little animated bitmap on the screen. This avoids all the problems that you have with the mouse competing. (Apple does update the mouse globals with the mouse position so that other things function correctly.)

Now, as promised, here is the way to do what you want using the real cursor. As you have discovered, setting the `crsrCouple` variable to false prohibits the mouse from affecting the cursor; unfortunately, it also prohibits the `jcrsrTask` routine from drawing the cursor. The solution to this is to set `crsr` couple to TRUE, call the cursor drawing routine `jCrsrTask` yourself, and then set the `crsrCouple` variable to false, as shown below:

```
procedure callcrsr;
              inline $2078 ,$08EE ,$4E90;
{            move.L    jcrsrTask,A0
            jsr    (A0) }

Procedure FudgeMouse;

type    PointPtr=^Point;

var        RawMouse:PointPtr;
        MTemp:PointPtr;
        RandPt:Point;
        CrsrNew:ptr;
        CrsrCouple:ptr;
        fred:Longint;

begin
        RawMouse:=PointPtr($82C);
        MTemp:=PointPtr($828);
        CrsrNew:=ptr($8CE);
        CrsrCouple:=ptr($8CF);
        RandPt:=RawMouse^;
        repeat
            RandPt.h:=RandPt.h+1;
            RandPt.V:=RandPt.v+1;
            RawMouse^:=RandPt;
            MTemp^:=RandPt;
            CrsrNew^:=1;
            CrsrCouple^:=1;
            callCrsr;
            crsrCouple^:=0;
            repeat until fred<tickCount;
            fred:=tickCount+3;
        until Button;
        crsrCouple^:=1;
end;
```

Back to top

Techniques for graying Macintosh text

Date Written: 6/3/91

Last reviewed: 6/14/93

How do I draw grayed-out text on the Macintosh, like the text for disabled buttons or menu items?

There are currently two different kinds of grayed text: First, there's "patterned" gray, where every other dot is missing. This really only looks good with Chicago or other heavy fonts and was always used for graying out menus and controls in system software through 6.0.x, and is still used in 7.0 when the screen is set to less than 4 bits deep. This is done by first drawing the text in a normal, `srcCopy` transfer mode. Then a gray rectangle is drawn over the text using the `patBic` mode. This "erases" half the bits in the text, and is rapid enough that there is very rarely any flicker.

The second kind of text is the actually gray text, which is used in System 7 on screens that are 4 bits deep or deeper for menus, controls, and other grayed text. To draw this text, just call `GetGray` (as documented on page 17-27 of *Inside Macintosh* Volume VI) to get an appropriate gray. Then draw the text in that color.

Back to top

Use srcOr instead of srcCopy for Macintosh text drawing

Date Written: 6/4/91

Last reviewed: 10/9/91

DrawText with srcCopy takes six times as long as with srcOr now that my Macintosh is running System 7. Why is this so slow? Is this a bug in System 7?

It's true that srcCopy is slower than srcOr when handling text, especially in color mode. This loss in speed occurs because CopyBits is a lot smarter than it used to be. It can handle foreground and background colors a lot better, but that improvement came at the cost of speed. Our recommended method for drawing text is to erase before drawing, and use srcOr to draw, not srcCopy. Alternatively, you could draw colorized text in srcOr mode off screen and then use CopyBits to draw it on the screen in srcCopy mode without colorization.

Back to top

Code for reversing Macintosh PICT images

Date Written: 3/4/91

Last reviewed: 6/14/93

Is there a simple way to put PICT images up in mirror image format, or is there sample code showing how to flip an offscreen bitmap?

There is no easy way to do this, nor do we have sample code showing how to flip an offscreen bitmap. Indeed, the best way to do what you want is to draw it to an offscreen pixel map and reverse it.

If you are using Color QuickDraw, always draw it to an 8-bit-per-pixel offscreen bitmap, and then the reverse is a very simple task. Here is some sample Pascal code that might roughly do what you want, with the following assumptions:

1. You are going to add error checking where appropriate.
2. Rowbytes correspond exactly to pixel width of the port.
3. The port is 8 bits deep.
4. You add the code to make this sketch work.
5. The origin of your offscreen port is (0,0).

```
Procedure FlipScanLine(theV:Integer; thePort:cGrafPtr);
{ Given any scan line number in the indicated port, this routine will flip }
{ that scan line horizontally. This routine assumes that you have made     }
{ sure that scan line theV exists. }

type ScanLn=Packed Array [0..0] of Byte;
   ScanPtr=^ScanLine;
var thePixMap:PixMapPtr;
   Index,Size:Integer;
   ThisScanLine:ScanPtr;
   TempPixel:Byte;

Begin
 thePixMap:=thePort^.PortPixMap^;
{ First create a pointer to the scan line we are currently reversing. }
 ThisScanLine:=ScanPtr(thePixMap^.BaseAddr);
 ThisScanLine:=ScanPtr(ord4(ThisScanLine)+(thePixMap^.RowBytes*theV));

{ Now simply reverse all the bytes. }
{ The scan line is simply an array [0..RowBytes] of Byte, and since this is }
{ 8 bits per pixel, each one is a single pixel.}
 Size:=thePixMap^.RowBytes;
 For Index:=0 to (Size div 2) do
  begin
   tempPixel:=ThisScanLine^[Index];
   ThisScanLine^[Index]:=ThisScanLine[Size-Index-1];
   ThisScanLine^[Index]:=tempPixel;
  end;
end;
```

This same procedure can be used also to swap a 1-, 2- or 4-bit-per-pixel pixmap if you add a function that accepts a byte and swaps the pixels in it.

Back to top

Using dithered drawing mode with QuickDraw

Date Written: 11/28/90

Last reviewed: 12/19/90

When I draw a 32-bit Macintosh PICT image from a file to an 8-bit port via an offscreen GWorld, I use dither mode in the CopyBits call and the results are quite impressive. If there is not enough memory to allocate the GWorld, I draw the image directly to the port. But since there does not seem to be any way to tell QuickDraw to use dithered drawing mode, the image looks horrible.

Do you have any suggestions? I have installed bottleneck procs to allow DrawPicture to get its data from the file instead of the handle in memory. Is there a way, while in the bottlenecks, to find the CopyBits call that comes from the picture and force it to use dithered mode instead of source mode? I don't want to try and parse the PICT myself, but I thought that maybe a QuickDraw global could be modified in my StdBits proc to force dithered drawing for that operation only?

You can install a StdBits or bitsProc bottleneck procedure to get all the CopyBits calls when the picture is being played back. One of the parameters to the StdBits call is the mode. You can install a procedure that saves the current mode, and then passes ditherMode to the original StdBits proc. This is all you should need to do. It's been done here so we know it works, only not in any form that can be sent to you as sample code at this time.

Back to top

Using PicComments to rotate text

Date Written: 11/28/90

Last reviewed: 12/19/90

I have a PostScript routine (using TextBegin/TextEnd) to generate bitmapped rotated text on the screen (which can be later printed on QuickDraw printers). Why do I get duplicate text? I get both bitmapped rotated text and PostScript rotated text when I print on the LaserWriter II, and both bitmapped rotated text and horizontal text on the ImageWriter. When I make a machine dependent check (check type of printer) and call the proper printing procedure, it works fine. Because of the speed and memory considerations of generating the rotated bitmapped text (especially at 300 dpi), is there a way to ensure that the printer will use the PostScript BEFORE generating the bitmap?

We will use the following Macintosh PicComments to hide your QuickDraw calls from the LaserWriter, but the ImageWriter will use them:

```
PostScriptBegin
 >> Put your CopyBits and QuickDraw calls to image your rotated
 >> bitmapped text here....
PostScriptEnd
```

By wrapping your QuickDraw code within the PostScriptBegin and PostScriptEnd PicComments, the code will be ignored by the LaserWriter, but the ImageWriter will use the QuickDraw calls. Basically, the PostScriptBegin and PostScriptEnd PicComments tell the LaserWriter driver to turn "off" QuickDraw. In the ImageWriter case, the ImageWriter does not understand the PicComments. Therefore, it will use the QuickDraw calls to create and image your bitmapped text.

Now, we need to use the rotation PicComments to rotate the text on the LaserWriter, but have the ImageWriter ignore the code:

```
Rect  zeroRect;

SetRect (&zeroRect, 0, 0, 0, 0);

TextBegin
TextCenter
      ClipRect (&zeroRect);

      >> Draw your text to be rotated on the LaserWriter....

      ClipRect (&rPageRect);
TextEnd
```

Wrapping your text drawing call(s) between the ClipRect calls will ensure that the text is drawn only on the LaserWriter. Setting the ClipRect to zero tells the ImageWriter to ignore all QuickDraw calls until the ClipRect is reset to something "real" (actually, a zero ClipRect prevents QuickDraw from drawing anything). After we have completed drawing the rotated text, we reset the ClipRect to the dimensions of rPage (that is, rPage is the image-able area of the currently selected printer--see *Inside Macintosh* Volume II, page 150). This will allow all of your normal drawing to continue on the ImageWriter and LaserWriter. If you did not reset the ClipRect after the TextEnd call, nothing would be drawn on the ImageWriter or LaserWriter.

Back to top

Why grafPort's clipRgn should be changed before OpenPicture

Date Written: 11/1/90

Last reviewed: 12/19/90

On page 189 of *Inside Macintosh* Volume I, in the QuickDraw chapter's description of OpenPicture, is the following warning: "A grafPort's clipRgn is initialized to an arbitrarily large region. You should always change the clipRgn to a smaller region before calling OpenPicture, or no drawing may occur when you call DrawPicture." The "arbitrarily large" clipping region rectangle is set to -32767,- 32767,32767,32767 (top, left, bottom, right) for new ports. This is the largest rectangle possible. If this is not a "valid" clipping rectangle for pictures, what is? Is there some specific limit to the size of the clipping rectangle? Does it depend on either available memory or the size of the picture?

Inside Macintosh' s warning is based on truth but it's incomplete. It didn't actually say that this rectangle is invalid as a clipping region, because this is in fact a perfectly valid clipping region. But, you could run into problems if you use this as a clipping region when creating a QuickDraw picture. It's not a matter of available memory or size; it's a simple matter of 16-bit signed integer overflow and underflow.

When you open a picture, the current clip region is recorded in the picture (this wasn't necessarily true in some early versions of QuickDraw). When you draw the resulting picture using the picture's picFrame as the destination rectangle, there won't be any problems. But if you use a destination rectangle that's larger than the picFrame, QuickDraw scales everything in the picture proportionately, including the clip region. If you allowed the default clip region to be recorded into the picture, then its rgnBBox, already as large as possible, will be made even larger. That means that the -32767 coordinates might wrap around to the positive number range, and the 32767 coordinates might wrap around to the negative number range. This leaves you with an empty clip region. Nothing at all gets drawn when the current port's clip region is empty.

If the destination rectangle is smaller than the picture's picFrame, you won't have any problems because the default clip region will be made smaller, and that's no problem.

This is why *Inside Macintosh* suggests that you make the clip region smaller than the default clip region before opening a picture. By doing this, you're almost guaranteed that the clip region won't get scaled to the point that it turns inside out. What size should you make it? Small enough so that the risk of the clip region's coordinates being scaled out of QuickDraw coordinate space is minimal. I usually just set the clip region to the picFrame of the picture. It's hard to go wrong this way.

Back to top

Calling InitCursor instead of SetCursor

Date Written: 10/23/90

Last reviewed: 6/14/93

Is it legal to call InitCursor instead of SetCursor(arrow) when I want to set the cursor to an arrow (after my normal one-time program initialization code, in my UpdateCursor routine)? The only reason I'd want to do such a nasty thing is to save code. Calling a trap with no parameters is less code than one with parameters. What, exactly, if anything, does InitCursor do besides setting the cursor to an arrow and setting the cursor level to zero?

There's no problem at all with this, as long as you are aware that the hidden, busy, and obscured states are cleared when you call InitCursor, so if the cursor was hidden or obscured for good reason it'll suddenly reappear. It also gets the arrow from QuickDraw, of course, but that's not a problem.

Back to top

Macintosh PICT-to-PostScript conversion

Date Written: 8/3/90

Last reviewed: 10/8/91

How do I convert PICT format data to PostScript in my printer driver?

Converting PICT files to PostScript involves a detailed understanding of both bitmaps (or pixmaps) and the graphics state in PostScript, which is a data structure defining the context in which other graphic operators in PostScript execute. If you don't know PostScript, the following manuals are a must:

- PostScript Language Tutorial and Cookbook (Addison-Wesley) is an introduction to PostScript.
- PostScript Language Reference Manual (Addison-Wesley).
- PostScript Language Program Design (Addison-Wesley) details designing efficient PostScript programs. It has a lot of useful sample programs on topics like writing a print spooler.

You need to convert all the QuickDraw operations in a PICT to corresponding PostScript operations. To get a feel for this conversion, you can analyze the PostScript dump from a LaserWriter to see how it converts a PICT to PostScript. Under System 6.x, a PostScript dump can be obtained by pressing Command-K while printing. Under System 7.0, you can get a dump by selecting the PostScript File option in the Print dialog.

Some areas of QuickDraw, such as transfer modes, do not have a correspondence in PostScript. The PostScript imaging model is designed so that all areas of a page affected by an image are marked as if with opaque paint. Using image masks can help. See the Graphics chapter in the PostScript reference manual.

PICT-to-PostScript conversion can be a long process, especially if one is unfamiliar with PostScript. Using the above books and the PostScript dump from the LaserWriter (but ONLY as a general guide) should help.

Back to top

Sending PostScript via PostScriptHandle PicComment

Date Written: 5/1/90

Last reviewed: 10/9/91

If I use the `PostScriptHandle PicComment` to send PostScript code to the LaserWriter driver, do I need to open a picture and then draw the picture to the driver, or can I just use the `PicComment` with no picture open while drawing to the printer's `grafPort`?

You don't need to create a picture with your `PicComment` in it and draw the picture to the driver. The best method for sending PostScript code to the LaserWriter is to use the `PostScriptHandle PicComment` documented in the Macintosh Technical Note "Optimizing for the LaserWriter--Picture Comments," as shown below.

```
PrOpenPage(...)
{ Send some QuickDraw so that the Printing Manager gets a }
{ chance to define the clipping region. }
PenSize(0,0);
MoveTo(0,0);
LineTo(0,0);
PenSize(1,1);
PicComment(PostScriptBegin, 0, NIL);
{ QuickDraw representation of graphic. }
MoveTo(100, 100);
LineTo(200, 200);
{ PostScript representation of graphic. }
thePSHandle^^ := '100 100 moveto 200 200 lineto stroke';

PicComment(PostScriptHandle, GetHandleSize(thePSHandle),
        thePSHandl);
PicComment(PostScriptEnd, 0, NIL);
PrClosePage(...)
```

The above code prints a line on any type of printer, PostScript or not. The first `MoveTo/LineTo` combination is required to give the LaserWriter driver a chance to define a clipping region. The LaserWriter driver replaces the `grafProcs` record in the `grafPort` returned from `PrOpenDoc`. In order for the LaserWriter driver to get execution time, you must execute a QuickDraw drawing routine that calls one of the `grafProcs`. In this case, the

MoveTo/LineTo combination calls the StdLine grafProc. When StdLine executes, it notices that the grafPort has been reinitialized, and therefore initializes the clipping region for the port. Until the MoveTo/LineTo combination is executed, the clipping region for the port is set to (0,0,0,0). If PostScript code is sent via the PostScriptHandle PicComment before executing any QuickDraw routines, all PostScript operations will be clipped to (0,0,0,0).

The next thing that's done is to send the PostScriptBegin PicComment. This comment is recognized only by PostScript printer drivers. When the driver receives this comment, it saves the current state of the PostScript device (by executing the PostScript gsave operator), then disables all QuickDraw drawing operations. This way, the QuickDraw representation of the graphic will be ignored by PostScript devices. In the above example, the second MoveTo/LineTo combination is executed only on non-PostScript devices.

The next PicComment is PostScriptHandle, which tells the driver that the data in thePSHandle is to be sent to the device as PostScript code. The driver then passes this code unchanged to the PostScript device for execution. The PostScriptHandle comment is recognized only by PostScript printer drivers.

The last PicComment, PostScriptEnd, tells the driver to restore the previous state of the device (via a PostScript grestore call), and to enable QuickDraw drawing operations.

Since most PicComments are ignored by QuickDraw devices, only the QuickDraw representation is printed. Since PostScriptBegin tells PostScript drivers to ignore QuickDraw operations, only the PostScript representation is printed on PostScript devices. This is a truly device-independent method for providing both PostScript and QuickDraw representations of a document.

Back to top

Macintosh QuickDraw region quirks

Date Written: 1/1/90

Last reviewed: 11/21/90

I'm working with regions, and I'm having problems with Macintosh QuickDraw trashing the heap and crashing, even though my regions are under 32K.

There are some quirks in the current version of QuickDraw. Here are some the commonly-encountered problems:

1. When doing operations which use more than one region, sduch as UnionRgn, DiffRgn, XorRgn, or SectRgn, the sum of the sizes of the source regions *must be less than* 32K, regardless of the size of the resulting region.
2. FrameRgn will fail if it tries to frame a region bigger than 16K.
3. If CloseRgn fails, the internal region data is already corrupt; there is nothing you can do to recover. CloseRgn will also fail if there isn't at least a 32K block of free space available.

Here are some workarounds:

1. Keep regions small and not too complex. Keep track of the sizes of all regions so you can check the SUM of the sizes before calling a routine that has a 32K limit.
2. Keep 32K free, or allocate a 32K block and release it just before calling CloseRgn.

Apple is working on these problems and expects to fix them in future versions of QuickDraw.

Back to top

How to get Macintosh QuickDraw arc endpoints

Date Written: 1/1/90

Last reviewed: 6/14/93

Is there a way to obtain the endpoints of an arc drawn by the Macintosh QuickDraw arc routines, such as FrameArc and PaintArc?

Given a rectangle R which frames the arc you wish to draw, convert your angles to an absolute coordinate system, where three o'clock is 0 degrees and 12 o'clock is 90 degrees.

Now, let:

```
    x = .5 (+ or -) (R.right - R.left)
    y = .5 (+ or -) (R.bottom - R.top)
```

The endpoint of the curve will be defined by:

```
    EndPoint.h = x (+ or -) cos(ang);
    EndPoint.v = y (+ or -) sin(ang);
```

h & v are relative to center of rectangle R

This calculates only the upper endpoint of the arc, but you can easily calculate the other endpoint using the same formula by calculating the absolute angle for the start point and applying the same formula.

Here is a subroutine which illustrates the algorithm, in ThinkSpeed Pascal:

```
{ DrawCurve: draw an arc from 0 degrees until the point defined }
{ by 'angle'. At that point draw a 4 by 4 crosshair. }

procedure DrawCurve (frame : Rect; angle : integer);

var
  x, y : integer;
  xr, yr : extended;
  rad : extended;

begin

  { Convert angle to radians }
  rad := (90 - angle) / 180 * 3.14159;

  { Find end point }
  xr := (frame.right - frame.left) * cos(rad) / 2;
  yr := (frame.bottom - frame.top) * sin(rad) / 2;
  x := (frame.right + frame.left) / 2 + Num2Integer(xr);
  y := (frame.bottom + frame.top) / 2 + Num2Integer(yr);

  { Draw crosshair }
  MoveTo(x - 4, y);
  LineTo(x + 4, y);
  MoveTo(x, y - 4);
  LineTo(x, y + 4);

  { Draw arc }
  FrameArc(frame, 0, angle);
end;
```

Back to top

Downloadables

Acrobat version of this Note (K). Download

Technical Notes by Date | Number | Technology | Title
Developer Documentation | Technical Q&As | Development Kits | Sample Code