Technical Note QD08
Old-Style Colors

CONTENTS

This Technical Note covers limitations of the original Macintosh color model (eight-color) that *Inside Macintosh* Volume I, page I-173, QuickDraw, does not document.

[Oct 01 1989]

Introduction

QuickDraw has always been able to deal with color, just on a very limited basis. Most applications have not made use of this feature, since Color QuickDraw-based Macintosh computers come with a better color model. There are, however, a few nice features that come with the old-style color model. With the old-style colors, it is easy to print color on an ImageWriter with a color ribbon. Another advantage is that developers do not have to write special-case code depending on whether or not a machine has Color QuickDraw.

Now that you are ready to convert to the old-style colors, there are a few things you should know about that do not work with old-style colors. This Note covers the limitations of using old-style colors, as well as the best ways to work around these limitations.

Back to top

Limitations

The most obvious limitation is that of only eight colors: black, white, red, green, blue, cyan, yellow, and magenta. This limitation is a problem only if you want to produce a color-intensive application; if this describes your application, then you need not read any further in this Note.

The next limitation is that off-screen buffers are not very useful. You can draw into off-screen buffers, but there is no way to get the colors back from the buffer. This leads into the next limitation, which is that `_CopyBits` cannot copy more than one color at a time.

When you call `_CopyBits` from an off-screen buffer to your window, you need to set the forecolor to the color you want to copy before calling `_CopyBits` (for example, to copy a red object, call `_ForeColor(redColor)`). Now when you copy the object, you can copy only one color. If you copy different colored objects at one time, then you have a problem. The result of a multicolored copy is that all objects copy in the same color, that of the foreground.

It is possible to work with an off-screen buffer and the old-style colors, but it requires a lot of extra work. Unless the objects are really complex, then it is probably easier to just draw the objects directly into your window.

One other limitation does exist. Consider the following code sample. One would assume that this sample would work at all times.

```
SetPort (myPort);
savedFG := myPort^.fgColor;
ForeColor (redColor);            {or any other color}

{...drawing takes place here...}

ForeColor (savedFG);
```

Surprise. It does not always work. The saved value for the `fgColor` field of the `grafPort` is not a classic QuickDraw color if the `grafPort` is actually a `cGrafPort`. If dealing with a `cGrafPort`, the `fgColor` field actually contains the foreground color's entry in the color table, so the second call to `_ForeColor` really messes things up.

The proper way to set and reset the foreground color with classic QuickDraw's `_ForeColor` call is as follows:

```
SetPort (myPort);
savedFG := myPort^.fgColor;
ForeColor (redColor);            {or any other color}

{...drawing takes place here...}

myPort^.fgColor := savedFG;      {Manually stuff the old fgColor back.}
If (32BQD = TRUE) Then           {32BQD is a flag that is made and set by}
    PortChanged (myPort);        {the application; to set it, the application}
                                 {needs to check _Gestalt for 32-Bit QuickDraw.}
```

This Note also applies to the routine `_BackColor`.

Back to top

What Works

The easiest way to work with these limited colors is to use pictures. When you draw the images, you should draw into a picture. Then when you want to draw the images into your window or to a printer, call `_DrawPicture`. Pictures work well with the old-style colors, and you don't need to worry about making sure that the forecolor is current when you draw into your window.

Once you have the picture, you can use it to draw into the screen or to the printer port. You can also set the `WindowRecord`s `windowPic` to equal your `PictureHandle` so updates are handled by the Window Manager.

Back to top

What Do Those Constants Mean Anyway?

The correct values are

`blackColor = 33`

`whiteColor = 30`

`redColor = 205`

`greenColor = 341`

`blueColor = 409`

```
cyanColor = 273

magentaColor = 137

yellowColor = 69
```

The following discussion is theoretical and was based on the color constants for the MPW 3.1 interfaces. Well, those interfaces were wrong as far as the color constants. The discussion will be kept here to prove once and for all that Macintosh programming sometimes is arbitrary and not logical. On the other hand, the information about the color bits is correct.

Each of the constants contains 9 bits of information, and each bit has a special meaning. Figure 1 illustrates the meaning of each of the bits, while Table 1 shows how each of the color constants fills in the appropriate bits.

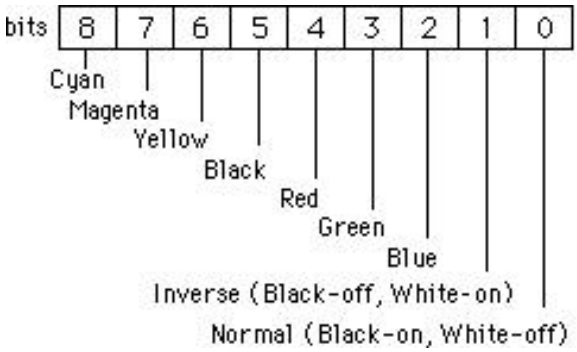

Figure 1. Bit Definitions

Table 1. Color-Bit Correlation

	black (33)	white (30)	red (209)	green (329)	blue (389)	cyan (269)	magenta (149)	yellow (89)
Cyan	0	0	0	1	1	1	0	0
Magenta	0	0	1	0	1	0	1	0
Yellow	0	0	1	1	0	0	0	1
Black	1	0	0	0	0	0	0	0
Red	0	1	1	0	0	0	1	1
Green	0	1	0	1	0	1	0	1
Blue	0	1	0	0	1	1	1	0
Inverse	0	1	0	0	0	0	0	0
Normal	1	0	1	1	1	1	1	1

Back to top

References

Inside Macintosh , Volume I, page I-173, QuickDraw

Technical Note M.IM.Copybits--Of Time and Space and _CopyBits

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