The evolution of the ICO file format, part 1: Monochrome beginnings

devblogs.microsoft.com/oldnewthing/20101018-00

October 18, 2010

Raymond Chen

This week is devoted to the evolution of the ICO file format. Note that the icon resource format is different from the ICO file format; I'll save that topic for another day.

The ICO file begins with a fixed header:

```
typedef struct ICONDIR {
    WORD          idReserved;
    WORD          idType;
    WORD          idCount;
    ICONDIRENTRY  idEntries[];
} ICONHEADER;
```

`idReserved` must be zero, and `idType` must be 1. The `idCount` describes how many images are included in this ICO file. An ICO file is really a collection of images; the theory is that each image is an alternate representation of the same underlying concept, but at different sizes and color depths. There is nothing to prevent you, in principle, from creating an ICO file where the 16×16 image looks nothing like the 32×32 image, but your users will probably be confused.

After the `idCount` is an array of `ICONDIRECTORY` entries whose length is given by `idCount`.

```
struct IconDirectoryEntry {
    BYTE  bWidth;
    BYTE  bHeight;
    BYTE  bColorCount;
    BYTE  bReserved;
    WORD  wPlanes;
    WORD  wBitCount;
    DWORD dwBytesInRes;
    DWORD dwImageOffset;
};
```

The `bWidth` and `bHeight` are the dimensions of the image. Originally, the supported range was 1 through 255, but starting in Windows 95 (and Windows NT 4), the value 0 is accepted as representing a width or height of 256.

The `wBitCount` and `wPlanes` describe the color depth of the image; for monochrome icons, these value are both 1. The `bReserved` must be zero. The `dwImageOffset` and `dwBytesInRes` describe the location (relative to the start of the ICO file) and size in bytes of the actual image data.

And then there's `bColorCount`. Poor `bColorCount`. It's supposed to be equal to the number of colors in the image; in other words,

```
bColorCount = 1 << (wBitCount * wPlanes)
```

If `wBitCount * wPlanes` is greater than or equal to 8, then `bColorCount` is zero.

In practice, a lot of people get lazy about filling in the `bColorCount` and set it to zero, even for 4-color or 16-color icons. Starting in Windows XP, Windows autodetects this common error, but its autocorrection is slightly buggy in the case of planar bitmaps. Fortunately, almost nobody uses planar bitmaps any more, but still, it would be in your best interest not to rely on the autocorrection performed by Windows and just set your `bColorCount` correctly in the first place. An incorrect `bColorCount` means that when Windows tries to find the best image for your icon, it may choose a suboptimal one because it based its decision on incorrect color depth information.

Although it probably isn't true, I will pretend that monochrome icons existed before color icons, because it makes the storytelling easier.

A monochome icon is described by two bitmaps, called *AND* (or *mask*) and *XOR* (or *image*, or when we get to color icons, *color*). Drawing an icon takes place in two steps: First, the mask is ANDed with the screen, then the image is XORed. In other words,

```
pixel = (screen AND mask) XOR image
```

By choosing appropriate values for *mask* and *image*, you can cover all the possible monochrome BLT operations.

mask	image	result	operation
0	0	(screen AND 0) XOR 0 = 0	blackness
0	1	(screen AND 0) XOR 1 = 1	whiteness

1	0	(screen AND 1) XOR 0 = screen	nop
1	1	(screen AND 1) XOR 1 = NOT screen	invert

Conceptually, the *mask* specifies which pixels from the *image* should be copied to the destination: A black pixel in the mask means that the corresponding pixel in the image is copied.

The mask and image bitmaps are physically stored as one single double-height DIB. The image bitmap comes first, followed by the mask. (But since DIBs are stored bottom-up, if you actually look at the bitmap, the mask is in the top half of the bitmap and the image is in the bottom half).

In terms of file format, each icon image is stored in the form of a `BITMAPINFO` (which itself takes the form of a `BITMAPINFOHEADER` followed by a color table), followed by the image pixels, followed by the mask pixels. The `biCompression` must be `BI_RGB`. Since this is a double-height bitmap, the `biWidth` is the width of the image, but the `biHeight` is *double* the image height. For example, a 16×16 icon would specify a width of 16 but a height of 16 × 2 = 32.

That's pretty much it for classic monochrome icons. Next time we'll look at color icons.

Still, given what you know now, the following story will make sense.

A customer contacted the shell team to report that despite all their best efforts, they could not get Windows to use the image they wanted from their .ICO file. Windows for some reason always chose a low-color icon instead of using the high-color icon. For example, even though the .ICO file had a 32bpp image available, Windows always chose to use the 16-color (4bpp) image, even when running on a 32bpp display.

A closer inspection of the offending .ICO file revealed that the `bColorCount` in the `IconDirectoryEntry` for all the images was set to 1, regardless of the actual color depth of the image. The table of contents for the .ICO file said "Yeah, all I've got are monochrome images. I've got three 48×48 monochrome images, three 32×32 monochrome images, and three 16×16 monochrome images." Given this information, Windows figured, "Well, given those choices, I guess that means I'll use the monochrome one." It chose one of images (at pseudo-random), and went to the bitmap data and found, "Oh, hey, how about that, it's actually a 16-color image. Okay, well, I guess I can load that."

In summary, the .ICO file was improperly authored. Patching each `IconDirectoryEntry` in a hex editor made the icon work as intended. The customer thanked us for our investigation and said that they would take the issue up with their graphic design team.

Raymond Chen

Follow