What does the CS_OWNDC class style do?

devblogs.microsoft.com/oldnewthing/20060601-06

June 1, 2006

Raymond Chen

Recall that window DCs are most commonly used only temporarily. If you need to draw into a window, you call `BeginPaint` or, if outside of a paint cycle, `GetDC`, although painting outside of a paint cycle is generally to be avoided. The window manager produces a DC for the window and returns it. You use the DC, then restore it to its original state and return it to the window manager with `EndPaint` (or `ReleaseDC`). Internally, the window manager keeps a small cache of DCs which it dips into when people come asking for a window DC, and when the DC is returned, it goes back into the cache. Since window DCs are used only temporarily, the number of outstanding DCs is typically not more than a handful, and a small cache is sufficient to satisfy DC demands in a normally-running system.

If you register a window class and include the `CS_OWNDC` flag in the class styles, then the window manager creates a DC for the window and puts it into the DC cache with a special tag that means "Do not purge this DC from the DC cache because it's the `CS_OWNDC` for this window." If you call `BeginPaint` or `GetDC` to get a DC for a `CS_OWNDC` window, then that DC will always be found and returned (since it was marked as "never purge"). The consequences of this are good, bad, and worse.

The good part is that since the DC has been created specially for the window and is never purged, you don't have to worry about "cleaning up the DC" before returning it to the cache. Whenever you call `BeginPaint` or `GetDC` for a `CS_OWNDC` window, you always get that special DC back. Indeed, that's the whole point of `CS_OWNDC` windows: You can create a `CS_OWNDC` window, get its DC, set it up the way you like it (selecting fonts, setting colors, *etc.*), and even if you release the DC and get it again later, you will get that same DC back and it will be just the way you left it.

The bad part is that you're taking something that was meant to be used only temporarily (a window DC) and using it permanently. Early versions of Windows had a very low limit for DCs (eight or so), so it was crucial that DCs be released as soon as they weren't needed. That limit has since been raised significantly, but the underlying principle remains: DCs should not be allocated carelessly. You may have noticed that the implementation of `CS_OWNDC` still uses the DC cache; it's just that those DCs get a special marking so the DC manager knows to

treat them specially. This means that a large number of `CS_OWNDC` DCs end up "polluting" the DC cache, slowing down future calls to functions like `BeginPaint` and `ReleaseDC` that need to search through the DC cache.

(Why wasn't the DC manager optimized to handle the case of a large number of `CS_OWNDC` DCs? First, as I already noted, the original DC manager didn't have to worry about the case of a large number of DCs since the system simply couldn't even create that many in the first place. Second, even after the limit on the number of DCs was raised, there wasn't much point in rewriting the DC manager to optimize the handling of `CS_OWNDC` DCs since programmers were already told to use `CS_OWNDC` sparingly. This is one of the practicalities of software engineering: You can do only so much. Everything you decide to do comes at the expense of something else. It's hard to justify optimizing a scenario that programmers were told to avoid and which they in fact were already avoiding. You don't optimize for the case where somebody is abusing your system. It's like spending time designing a car's engine so it maintained good gas mileage when the car has no oil.)

The worse part is that most windowing framework libraries and nearly all sample code assume that your windows are not `CS_OWNDC` windows. Consider the following code that draws text in two fonts, using the first font to guide the placement of characters in the second. It looks perfectly fine, doesn't it?

```
void FunnyDraw(HWND hwnd, HFONT hf1, HFONT hf2)
{
 HDC hdc1 = GetDC(hwnd);
 HFONT hfPrev1 = SelectFont(hdc1, hf1);
 UINT taPrev1 = SetTextAlign(hdc1, TA_UPDATECP);
 MoveToEx(hdc1, 0, 0, NULL);
 HDC hdc2 = GetDC(hwnd);
 HFONT hfPrev2 = SelectFont(hdc2, hf2);
 for (LPTSTR psz = TEXT("Hello"); *psz; psz++) {
  POINT pt;
  GetCurrentPositionEx(hdc1, &pt);
  TextOut(hdc2, pt.x, pt.y + 30, psz, 1);
  TextOut(hdc1, 0, 0, psz, 1);
 }
 SelectFont(hdc1, hfPrev1);
 SelectFont(hdc2, hfPrev2);
 SetTextAlign(hdc1, taPrev1);
 ReleaseDC(hwnd, hdc1);
 ReleaseDC(hwnd, hdc2);
}
```

We get two DCs for the window. In the first we select our first font; in the second, we select the second. In the first DC, we also set the text alignment to `TA_UPDATECP` which means that the coordinates passed to the `TextOut` function will be ignored. Instead the text will be drawn starting at the "current position" and the "current position" will be updated to the end of the string, so that the next call to `TextOut` will resume where the previous one left off.

Once the two DCs are set up, we draw our string one character at a time. We query the first DC for the current position and draw the character in the second font at that same *x*-coordinate (but a bit lower), then we draw the character in the first font (which also advances the current position).

After the text drawing loop is done, we restore the states of the two DCs as part of the standard bookkeeping.

The intent of the function is to draw something like this, where the first font is bigger than the second.

```
H e l l o
---------
H e l l o
```

And if the window is not `CS_OWNDC` that's what you get. You can try it out by calling it from our scratch program:

```
HFONT g_hfBig;
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
 LOGFONT lf;
 GetObject(GetStockFont(ANSI_VAR_FONT),
           sizeof(lf), &lf);
 lf.lfHeight *= 2;
 g_hfBig = CreateFontIndirect(&lf);
 return g_hfBig != NULL;
}
void
OnDestroy(HWND hwnd)
{
 if (g_hfBig) DeleteObject(g_hfBig);
 PostQuitMessage(0);
}
void
PaintContent(HWND hwnd, PAINTSTRUCT *pps)
{
 FunnyDraw(hwnd, g_hfBig,
                 GetStockFont(ANSI_VAR_FONT));
}
```

But if the window is `CS_OWNDC`, then bad things happen. Try it yourself by changing the line `wc.style = 0;` to `wc.style = CS_OWNDC;` You get the following unexpected output:

```
HHeellllloo
```

Of course, if you understand how `CS_OWNDC` works, this is hardly unexpected at all. The key to understanding is remembering that when the window is `CS_OWNDC` then `GetDC` just returns the same DC back no matter how many times you call it. Now all you have to do is walk through the `FunnyDraw` function remembering that `hdc1` and `hdc2` are in fact **the same thing**.

```
void FunnyDraw(HWND hwnd, HFONT hf1, HFONT hf2)
{
 HDC hdc1 = GetDC(hwnd);
 HFONT hfPrev1 = SelectFont(hdc1, hf1);
 UINT taPrev1 = SetTextAlign(hdc1, TA_UPDATECP);
 MoveToEx(hdc1, 0, 0, NULL);
```

So far, execution of the function is pretty normal.

```
 HDC hdc2 = GetDC(hwnd);
```

Since the window is a `CS_OWNDC` window, the DC that is returned in `hdc2` is the same one that was returned in `hdc1`. In other words, `hdc1 == hdc2`! Now things get exciting.

```
 HFONT hfPrev2 = SelectFont(hdc2, hf2);
```

Since `hdc1 == hdc2`, what this really does is deselect the font `hf1` from the DC and select the font `hf2` instead.

```
 for (LPTSTR psz = TEXT("Hello"); *psz; psz++) {
  POINT pt;
  GetCurrentPositionEx(hdc1, &pt);
  TextOut(hdc2, pt.x, pt.y + 30, psz, 1);
  TextOut(hdc1, 0, 0, psz, 1);
 }
```

Now this loop completely falls apart. At the first iteration, we retrieve the current position from the DC, which returns (0, 0) since we haven't moved it yet. We then draw the letter "H" at position (0, 30) into the second DC. But since the second DC is the same as the first one, what really happens is that we are calling `TextOut` into a DC that is in `TA_UPDATECP` mode. Thus, the coordinates are ignored, the letter "H" is displayed (in the second font), and the current position is updated to be after the "H". Finally, we draw the "H" into the first DC (which is the same as the second). We think we're drawing it with the first font, but in fact we're drawing with the second font. We think we're drawing at (0, 0), but in fact we're drawing at (x, 0), where x is the width of the letter "H", because the call to `TextOut(hdc2, ...)` updated the current position.

Thus, each time through the loop, the next character in the string is displayed twice, all in the second font.

But wait, the disaster isn't finished yet. Look at our cleanup code:

```
SelectFont(hdc1, hfPrev1);
```

This restores the original font into the DC.

```
SelectFont(hdc2, hfPrev2);
```

This re-selects the first font! We failed to restore the DC to its original state and ended up putting a “corrupted” DC into the cache.

That’s why I described `CS_OWNDC` as “worse”. It takes code that used to work and breaks it by violating assumptions that most people make (usually without realizing it) about DCs.

And you thought `CS_OWNDC` was bad. Next time I’ll talk about the disaster that is known as `CS_CLASSDC`.

Raymond Chen

Follow