Why do we even have the DefWindowProc function?

devblogs.microsoft.com/oldnewthing/20071105-00

November 5, 2007

Raymond Chen

Some time ago, I looked at two ways of reimplementing the dialog procedure (method 1, method 2). Commenter "8" wondered why we have a `DefWindowProc` function at all. Couldn't window procedures have followed the dialog box model, where they simply return `FALSE` to indicate that they want default processing to occur? Then there would be no need to export the `DefWindowProc` function.

This overlooks one key pattern for derived classes: Using the base class as a subroutine. That pattern is what prompted people to ask for dialog procedures that acted like window procedures. If you use the "Return `FALSE` to get default behavior" pattern, window procedures would go something like this:

```
BOOL DialogLikeWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
 switch (uMsg) {
 ... handle messages and return TRUE ...
 }
 // We didn't have any special processing; do the default thing
 return FALSE;
}
```

Similarly, subclassing in this hypothetical world would go like this:

```
BOOL DialogLikeSubclass(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
 switch (uMsg) {
 ... handle messages and return TRUE ...
 }
 // We didn't have any special processing; let the base class try
 CallDialogLikeWindowProc(PrevDialogLikeWndProc, hwnd, uMsg, wParam, lParam);
}
```

This works as long as what you want to do is override the base class behavior entirely. But what if you just want to augment it? Calling the previous window procedure is analogous to calling the base class implementation from a derived class, and doing so is quite common in object-oriented programming, where you want the derived class to behave “mostly” like the base class. Consider, for example, the case where we want to allow the user to drag a window by grabbing anywhere in the client area:

```
LRESULT CALLBACK CaptionDragWndProc(
    HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
 LRESULT lres;
 switch (uMsg) {
 case WM_NCHITTEST:
  lres = DefWindowProc(hwnd, uMsg, wParam, lParam);
  if (lres == HTCLIENT) lres = HTCAPTION;
  return lres;
 ...
 }
 return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

We want our hit-testing to behave just like normal, with the only exception that clicks in the client area should be treated as clicks on the caption. With the `DefWindowProc` model, we can do this by calling `DefWindowProc` to do the default processing, and then modifying the result on the back end. If we had use the dialog-box-like model, there would have been no way to call the “default handler” as a subroutine in order to make it to the heavy lifting. We would be forced to do all the work or none of it.

Another avenue that an explicit `DefWindowProc` function opens up is **modifying** messages before they reach the default handler. For example, suppose you have a read-only edit control, but you want it to look like a normal edit control instead of getting the static look. You can do this by modifying the message that you pass to `DefWindowProc`:

```
...
 case WM_CTLCOLORSTATIC:
  if (GET_WM_CTLCOLOR_HWND(wParam, lParam) == m_hwndEdit)
  {
   // give it the "edit" look
   return DefWindowProc(hwnd, WM_CTLCOLOREDIT, wParam, lParam);
  }
  ...
```

Another common operation is changing one color attribute of an edit control while leaving the others intact. For this, you can use `DefWindowProc` as a subroutine and then tweak the one attribute you want to customize.

```
case WM_CTLCOLORSTATIC:
 if (GET_WM_CTLCOLOR_HWND(wParam, lParam) == m_hwndDanger)
 {
  // Start with the default color attributes
  LRESULT lres = DefWindowProc(hwnd, uMsg, wParam, lParam);
  // Change text color to red; leave everything else the same
  SetTextColor(GET_WM_CTLCOLOR_HDC(wParam, lParam), RGB(255,0,0));
  return lres;
 }
 ...
```

Getting these types of operations to work with the dialog box model would be a significantly trickier undertaking.

Raymond Chen

Follow