The dialog manager, part 2: Creating the frame window

devblogs.microsoft.com/oldnewthing/20050330-00

March 30, 2005

Raymond Chen

The dialog template describes what the dialog box should look like, so the dialog manager walks the template and follows the instructions therein. It's pretty straightforward; there isn't much room for decision-making. You just do what the template says.

For simplicity, I'm going to assume that the dialog template is an extended dialog template. This is a superset of the classic DLGTEMPLATE, so there is no loss of generality.

Furthermore, I will skip over some of the esoterica (like the WM_ENTERIDLE message) because that would just be distracting from the main point.

I am also going to ignore error-checking for the same reason.

Finally, I'm going to assume you already understand the structure of the various dialog templates and ignore the parsing issues. (If you've forgotten, you can go back and re-read my series from last June. Most important are parts 2 and 4, and the summary table is a handy quick-reference.)

Okay, here we go.

The first order of business is to study the dialog styles and translate the `DS_*` styles into `WS_*` and `WS_EX_*` styles.

Dialog style	Window style	Extended window style
`DS_MODALFRAME`		add `WS_EX_DLGMODALFRAME` add `WS_EX_WINDOWEDGE`
`DS_CONTEXTHELP`		add `WS_EX_CONTEXTHELP`
`DS_CONTROL`	remove `WS_CAPTION` remove `WS_SYSMENU`	add `WS_EX_CONTROLPARENT`

Question: Why does the `DS_CONTROL` style remove the `WS_CAPTION` and `WS_SYSMENU` styles?

Answer: To make it easier for people to convert an existing dialog into a `DS_CONTROL` sub-dialog by simply adding a single style flag.

If the template includes a menu, the menu is loaded from the instance handle passed as part of the creation parameters.

```
  hmenu = LoadMenu(hinst, <resource identifier in template>);
```

This is a common theme in dialog creation: The instance handle you pass to the dialog creation function is used for all resource-related activities during dialog creation.

The algorithm for getting the dialog font goes like this:

```
  if (DS_SETFONT) {
    use font specified in template
  } else if (DS_FIXEDSYS) {
    use GetStockFont(SYSTEM_FIXED_FONT);
  } else {
    use GetStockFont(SYSTEM_FONT);
  }
```

Notice that `DS_SETFONT` takes priority over `DS_FIXEDFONT`. We saw the reason for this a few weeks ago.

Once the dialog manager has the font, it is measured so that its dimensions can be used to convert dialog units (DLUs) to pixels. Everything in dialog box layout is done in DLUs. Here's a reminder if you've forgotten the formula that converts DLUs to pixels. In explicit terms:

```
// 4 xdlu = 1 average character width
// 8 ydlu = 1 average character height
#define XDLU2Pix(xdlu) MulDiv(xdlu, AveCharWidth, 4)
#define YDLU2Pix(ydlu) MulDiv(ydlu, AveCharHeight, 8)
```

The dialog box size come from the template.

```
cxDlg = XDLU2Pix(DialogTemplate.cx);
cyDlg = YDLU2Pix(DialogTemplate.cy);
```

The dialog size in the template is the size of the *client area*, so we need to add in the nonclient area too.

```
  RECT rcAdjust = { 0, 0, cxDlg, cyDlg };
  AdjustWindowRectEx(&rcAdjust, dwStyle, hmenu != NULL, dwExStyle);
  int cxDlg = rcAdjust.right - rcAdjust.left;
  int cyDlg = rcAdjust.bottom - rcAdjust.top;
```

How do I know that it's the client area instead of the full window including nonclient area? Because if it were the full window rectangle, then it would be impossible to design a dialog! The template designer doesn't know what nonclient metrics the end-user's system will be set

to and therefore cannot take it into account at design time.

(This is a special case of a more general rule: If you're not sure whether something is true, ask yourself, "What would the world be like if it were true?" If you find a logical consequence that is obviously wrong, then you have just proven [by contradiction] that the thing you're considering is indeed not true. This is an important logical principle that I will come back to again and again. In fact, you saw it just a few days ago.)

Assuming the `DS_ABSALIGN` style is not set, the coordinates given in the dialog template are relative to the dialog's parent.

```
POINT pt = { XDLU2Pix(DialogTemplate.x),
             YDLU2Pix(DialogTemplate.y) };
ClientToScreen(hwndParent, &pt);
```

But what if the caller passed `hwndParent = NULL`? In that case, the dialog position is relative to the upper left corner of the primary monitor. But **don't do this**.

- On a multiple-monitor system, it puts the dialog box on the primary monitor, even if your program is running on a secondary monitor.
- The user may have docked their taskbar at the top or left edge of the screen, which will cover your dialog.
- Even on a single-monitor system, your program might be running in the lower-right corner of the screen. Putting your dialog at the upper left corner doesn't create a meaningful connection between the two.
- If two copies of your program are running, their dialog boxes will cover each other precisely. We saw the dangers of this in a previous entry.

Moral of the story: Always pass a `hwndParent` window so that the dialog appears in a meaningful location relative to the rest of your program. (And don't just grab `GetDesktopWindow` either!)

Okay, we are now all ready to create the dialog: We have its class, its font, its menu, its size and position…

Oh wait, we have to deal with that subtlety of dialog box creation discussed earlier: The dialog box is always created initially hidden.

```
BOOL fWasVisible = dwStyle & WS_VISIBLE;
dwStyle &= ~WS_VISIBLE;
```

The dialog class and title come from the template. Pretty much everyone just uses the default dialog class, although I explained in an earlier article how you might use a custom dialog class.

Okay, now we have the information necessary to create the window.

```
HWND hdlg = CreateWindowEx(dwExStyle, pszClass,
     pszCaption, dwStyle & 0xFFFF0000, pt.x, pt.y,
     cxDlg, cyDlg, hwndParent, hmenu, hinst, NULL);
```

Notice that we filter out all the low style bits (per-class) since we already translated the `DS_*` styles into "real" styles.

This is why your dialog procedure doesn't get the window creation messages like `WM_CREATE`. At the time the frame is created, the dialog procedure hasn't yet entered the picture. Only after the frame is created can the dialog manager attach the dialog procedure.

```
// Set the dialog procedure
SetWindowLongPtr(hdlg, DWLP_DLGPROC, (LPARAM)lpDlgProc);
```

The dialog manager does some more fiddling at this point, based on the dialog template styles. The template may have asked for a window context help ID. And if the template did not specify window styles that permit resizing, maximizing or minimizing, the associated menu items are removed from the dialog box's system menu.

And it sets the font.

```
 SetWindowFont(hdlg, hf, FALSE);
```

This is why the first message your dialog procedure receives happens to be `WM_SETFONT`: It is the first message sent after the `DWLP_DLGPROC` has been set. Of course, this behavior can change in the future; you shouldn't rely on message ordering.

Okay, the dialog frame is now open for business. Next up: Creating the controls.

Raymond Chen

Follow