Restating the obvious about the WM_NOTIFY message

devblogs.microsoft.com/oldnewthing/20090821-00

August 21, 2009

Raymond Chen

It appears that people seemed to appreciate restating the obvious about the `WM_COMMAND` message, so I'll try it again with the `WM_NOTIFY` message.

The `WM_NOTIFY` message is typically used by a control to communicate with its parent, either to provide information, request it, or both. Although that is the typical use, there are exceptions. For example, property sheets send the `WM_NOTIFY` to their *children*. Property sheets are this sort-of backwards model, where the common controls provide the parent window (the property sheet) and applications provide the child windows (the property sheet pages). The window manager doesn't care who sends the message to whom, as long as the sender and recipient are in the same process.

The message cannot cross a process boundary because `WM_NOTIFY` is basically a sender-defined version of `WM_USER`. Anybody can define a new notification code and associate it with any structure they want (as long as the structure begins with a `NMHDR`). The window manager can't marshal the structure between processes because the structure is defined by the control, not the window manager.

A little elaboration of that "sender-defined version of `WM_USER`": As we saw, the meaning of `WM_USER` messages is determined by the implementor of the window class. In other words, the code *receiving* the message decides what `WM_USER` means. That works great if you're some external code that wants to send a message to a known window class. But what if you're external code that wants to send a message to an unknown window class? For example, you're a list view control and you want to tell your parent about some event. You want to send a message to the parent window, but which message? You can't send anything in the `WM_USER` range because each parent window defines independently what those messages mean, and it's highly unlikely that all the parent windows are going to agree that `WM_USER+205` means the same thing. For similar reasons, the `WM_APP` range is no good. A registered message would work, but if you have hundreds of potential events, then a hundred registered messages is a bit heavy-handed. The old-school answer to this was the `WM_COMMAND` message, whose notification code is defined by the sending control. Unfortunately, the notification code is all you get; the other parameters are busy doing other things. Enter `WM_NOTIFY`, which is basically `WM_COMMAND` on steroids: The `NMHDR` structure contains everything that was in the `WM_COMMAND` message, and since it's a structure, you can embed the `NMHDR` inside a larger structure to provide (and possibly receive) more information.

Okay, end of strange digression.

The `NMHDR` structure itself is a convention, in the same way that the parameters to `WM_COMMAND` are a convention. The `hwndFrom` member is supposed to be the control that generated the notification, but there's no enforcement.

First, there's no way to enforce it. A window doesn't send a message; code sends a message. You can check the thread that is executing the code that is sending a message, but you don't know which window that code is associated with.

"Well, the window that is sending the message is the one that most recently received a message."

That doesn't work because you can have code associated with one window call code associated with another window without actually sending a message. In fact, you probably do this all the time:

```
class CFrame : public CWindow {
...
 LRESULT OnCommand(...);
...
 CGraphWindow *m_pwndGraph;
};
LRESULT CFrame::OnCommand(...)
{
 switch (idFrom) {
 case IDC_CPU: // user clicked the "CPU" button
  m_pwndGraph->ChangeMode(CPU); // change to a CPU graph
  ...
}
```

Suppose that `CGraphWindow::ChangeMode` function calls `SendMessage` as part of its processing. Which window "sent" this message? Since you have the power to read code, the message was conceptually sent by `CGraphWindow`, but the most recently received message is a `WM_COMMAND` sent to the frame window.

Your method call is just a transfer of control inside your program. The window manager doesn't know what's going on. All it knows is that it delivered a `WM_COMMAND` message to the frame window, and then some mystery code executed, and the next thing you know, somebody is sending a message. It doesn't have the source code to your program to know that "Oh, that's coming from `CGraphWindow::ChangeMode`, and to get the window handle for `CGraphWindow`, I should call `CGraphWindow::operator HWND()`." (And even if it did, imagine your surprise when your breakpoint on `CGraphWindow::operator HWND()` gets hit because `SendMessage` called it!)

Second, even if there were some psychic way for the window manager to figure out which window is sending the message, you still wouldn't want that. It is common for `WM_NOTIFY` handlers of complex controls to forward the message to another window. For example, the list view control in report mode receives `WM_NOTIFY` messages from the header control and forwards them back out to its own parent, so that the list view parent can respond to header notifications. (The parent normally should just let the list view handle it, but the operation is performed in case you're one of those special cases that needs it.)

Okay, back to what the fields of `NMHDR` mean. There are only three fixed fields to `NMHDR` and they pretty much match up with the parameters to `WM_COMMAND`:

- `hwndFrom` is the handle to the window that is the logical source of the notification.
- `idFrom` is the control ID corresponding to the window specified by `hwndFrom`. In other words, `idFrom = GetDlgCtrlID(hwndFrom)`.
- `code` is the notification code. The meaning of this notification code depends on the window class of `hwndFrom`.

It is an unwritten convention that the notification codes for the common controls are all negative numbers. This leaves positive numbers for applications to use for their own purposes. Not that applications strictly speaking needed the help, because the meaning of the notification code depends on the window that generated the notification, so if you want a brand new 32-bit message number namespace, just register a new window class, and boom, a whole new range of codes becomes available just to you. (Even though the notification code values do not need to be unique across window classes, the common controls team tries to keep the system-defined notification codes from overlapping, just to make debugging easier.)

The `idFrom` member is provided as a convenience to the window receiving the message so that it can use a simple `switch` statement to figure out who is sending the notification.

Once you figure out which notification you're receiving, you can use the documentation for that notification to see which structure is associated with the notification. This answers Norman Diamond's complaint that he couldn't figure out what to cast it to. For example, if the notification is `LVN_ITEMCHANGING`, well, let's see, the documentation for `LVN_ITEMCHANGING` says,

> ```
> LVN_ITEMCHANGING
> pnmv = (LPNMLISTVIEW) lParam;
> ```
>
> *pnmv*: Pointer to an NMLISTVIEW structure that identifies the item and specifies which of its attributes are changing.

In other words, your code goes something like this:

```
case LVN_ITEMCHANGING:
 pnmv = (LPNMLISTVIEW) lParam;
 ... do stuff with pnmv ...
```

I’m not sure how much more explicit the documentation could be made to be. All it was missing was the word `case` in front.

Raymond Chen

Follow