Visual Studio Magazine Online

Classic VB Corner

Subclassing the XP Way

ONLINE ONLY

Windows XP offers a much-improved method to subclass windows. Is it the right call for you to make?

July 16, 2009 · by Karl E. Peterson

I'll start with a little background. I switched over my main machines to XP only a little over two years ago, and that was pretty much kicking and screaming. Well, not really, but there was a bit of grumbling. XP was just goofy-looking, and I couldn't imagine the "business case" for moving from Windows 2000, a far more "serious" operating system for getting "real work" done. But I finally decided I really needed built-in support for hosting terminal server clients to more easily access machines of mine that weren't where I was, so I relented. It's an uneasy peace, but hey, I can download my photos easier now, too. Hmm.

I suppose all these years after its release, I'm also beginning to crack a bit on whether to use "XP or higher" methods. Certainly, it seems that in 2009, that's OK for most professional purposes, although there are still going to be any number of reasons that writing more portable apps appeal to some niches.

The functions I'm going to show you in this column were first documented in XP, but have in fact existed for a very long time (since Windows 98 and/or Internet Explorer version 4.01), although they were exported by ordinal only until XP was released. So perhaps the title of this piece is a bit misleading, as you can use these techniques just about anywhere.

It's with this introduction that I offer a method kind reader Markus Melk recently wrote to me (kudos, Markus!) about. I was actually quite taken aback that I'd never run across it before -- and after a few quick searches, it seems like very few in the Classic VB community have, either. I guess I can blame my own ignorance on simply not paying attention to the XP hype the first few years it was out. Was that a universal pattern followed in the Classic VB community? It appears that way. Well, that's (more than?) enough appetizer. Let's bring on the meat!

One of the ever-present concerns when hooking into a window's message stream is that if there are multiple hooks being set, they must be torn down in the exact opposite order that they were put in place. This was mandatory because in order to set the hook, you had to replace the window's GWL_WNDPROC property using SetWindowLong, storing the old value until you were ready to unhook. In this way, a chain of window handlers was created, with Windows calling the last installed first, that handler calling the previous handler next, and so on until the very first installed handler calls the default handler for that window.

But this is like a child's paper chain; it just tears apart if one of the hooks in the middle is undone first. The Windows shell team recognized this problem and created the SetWindowSubclass function. When you're through subclassing, you simply call RemoveWindowSubclass to unhook safely. This new capability totally eliminates the concern about multiple hooks and strict teardown order. I could hardly believe how magical it appeared as I was suddenly contemplating all the wonderful tricks this would enable.

Each subclass is identified two pieces of data along with the window's hWnd -- those being a pointer to the new message-handling procedure and a unique ID that's up to you to generate. You may also select to pass one Long value along as an extra parameter to each callback, for whatever purpose you may desire. The thought that immediately struck me was to use an ObjPtr() as the unique ID for each subclass. What object? The one that will be handling the callback! This is pure magic, I tell ya. So setting the hook (with code in a standard BAS module) is this easy:

```
Public Function HookSet(ByVal hWnd As Long, _
   ByVal Thing As IHookXP, Optional dwRefData As Long) As Boolean
   ' http://msdn.microsoft.com/en-us/library/bb762102(VS.85).aspx
   HookSet = CBool(SetWindowSubclass _
      (hWnd, AddressOf SubclassProc, ObjPtr(Thing), dwRefData))
End Function

Public Function HookClear(ByVal hWnd As Long, _
   ByVal Thing As IHookXP) As Boolean
   ' http://msdn.microsoft.com/en-us/library/bb762094(VS.85).aspx
   HookClear = CBool(RemoveWindowSubclass _
      (hWnd, AddressOf SubclassProc, ObjPtr(Thing)))
End Function
```

What's that IHookXP *Thing*, you ask? It's a simple little hook interface. This is the entire IHookXP.cls:

```
Option Explicit
' Implement this interface in objects that sink messages
' using the subclassing technique offered by MHookXP.
Public Function Message(ByVal hWnd As Long, _
                        ByVal uiMsg As Long, _
                        ByVal wParam As Long, _
                        ByVal lParam As Long, _
                        ByVal dwRefData As Long) As Long
End Function
```

So, in order to set a hook into a form's message stream and process those messages within the form itself, you'd use code like this:

```
' Subclassing interface
Implements IHookXP

Private Sub Form_Load()
   Call HookSet(Me.hWnd, Me)
End Sub

Private Sub Form_Unload(Cancel As Integer)
   Call HookClear(Me.hWnd, Me)
End Sub
```

Three lines of code in the form set up and finish the subclassing. All that's left is processing the messages. You'll note that in the HookSet and HookClear procedures, a pointer to SubclassProc is passed to each respective API function. This procedure is defined by the SDK to look like this:

```
typedef LRESULT (CALLBACK *SUBCLASSPROC)(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    UINT_PTR uIdSubclass,
    DWORD_PTR dwRefData
);
```

Here's where another little piece of magic occurs. Notice that uIdSubclass is simply a pointer to an unsigned integer. We need to provide a unique ID for this parameter. We could hardcode values, or generate GUIDs, or any number of other things. But we have an ObjPtr for the IHookXP interface exposed by each form or class that would like to be a message handler. This, too, is almost certainly a unique value as long as we're not continually creating, destroying and recreating hook objects. So look at the call to HookSet in Form_Load -- it's just passing a reference to Me as the Thing parameter. HookSet receives a pointer to the form's IHookXP interface, which it in turn passes to SetWindowSubclass. When the window is to be sent a message from Windows, our SubclassProc is called like this:

```
Public Function SubclassProc( _
   ByVal hWnd As Long, _
   ByVal uiMsg As Long, _
   ByVal wParam As Long, _
   ByVal lParam As Long, _
   ByVal uIdSubclass As IHookXP, _
   ByVal dwRefData As Long) As Long
   ' http://msdn.microsoft.com/en-us/library/bb776774(VS.85).aspx
   SubclassProc = uIdSubclass.Message( _
      hWnd, uiMsg, wParam, lParam, dwRefData)
End Function
```

Magic! Or close enough to make one smile, at any rate. By declaring uIdSubclass to be As IHookXP, VB can hand us a fully realized instance of the object that wants to be notified of each incoming message. All SubclassProc needs to do is fire off the single exposed method of that object's IHookXP interface. So, back in our form, we have:

```
Private Function IHookXP_Message( _
   ByVal hWnd As Long, ByVal uiMsg As Long, _
   ByVal wParam As Long, ByVal lParam As Long, _
   ByVal dwRefData As Long) As Long

   Debug.Print "hWnd: 0x"; Hex$(hWnd), "Msg: 0x"; Hex$(uiMsg), _
               "wParam: 0x"; Hex$(wParam), "lParam: 0x"; Hex$(lParam), _
               "RefData: "; dwRefData
   IHookXP_Message = HookDefault(hWnd, uiMsg, wParam, lParam)
End Function
```

It's here that all the custom message processing can take place. Or not. To let the default handler take over, XP provides a newly documented DefSubclassProc API function. DefSubclassProc safely calls the subclassed window's default message handler, or whatever hook follows yours in line to that procedure. I've wrapped that up in one final method of my standard BAS module:

```
Public Function HookDefault(ByVal hWnd As Long, _
   ByVal uiMsg As Long, ByVal wParam As Long, _
   ByVal lParam As Long) As Long
   ' http://msdn.microsoft.com/en-us/library/bb776403(VS.85).aspx
   HookDefault = DefSubclassProc(hWnd, uiMsg, wParam, lParam)
End Function
```

About the only undocumented caveat I've run into is that you must be sure to unhook your subclass before a window is destroyed. If you use my technique of embedding the handler in a form or class that's destroyed during the Form_Unload method, this should never be a problem. If you really don't want to take any chances, you can insert a simple branch in your handling routine:

```
   Select Case uiMsg
      Case WM_THIS
         '
      Case WM_THAT
         '
      Case WM_NCDESTROY
         Call Unhook  ' !!!
   End Select
```

I'm providing a ready-to-run sample on my site, of course. The absolute best way to wrap your head around this is to actually get into the code, run it and watch what happens. The sample provides two drop-in ready classes which will subclass any form in your project. One handles WM_GETMINMAXINFO, so that you can control how big or how small a user may resize it. The other class monitors WM_WINDOWPOSCHANGING for form movement, snapping the dialog to the edge of the screen if it gets within 15 pixels, using a technique I presented in an earlier column. Plus, the form itself has hooked into its own message stream and just dumps every message it gets to the Immediate window (as shown above).

This HookXP sample shows that you can now set multiple hooks on the same window, have them handled by multiple different objects, and not have to worry at all about how things get torn down. With this technique, you can build all the specialized handler objects you want and drop them into your projects at will, rather than try to combine the functionality of each into a single callback procedure.

As always, be safe. Unhandled errors can be deadly. Save before running. Enjoy!

About the Author

Karl E. Peterson wrote Q&A, Programming Techniques, and various other columns for VBPJ and VSM from 1995 onward, until Classic VB columns were dropped entirely in favor of other languages. Similarly, Karl was a Microsoft BASIC MVP from 1994 through 2005, until such community contributions were no longer deemed valuable. He is the author of VisualStudioMagazine.com's new Classic VB Corner column. You can contact him through his Web site if you'd like to suggest future topics for this column.