Visual Studio Magazine Online
Classic VB Corner

Reacting to the Mousewheel

ONLINE ONLY

ClassicVB was created in a pre-Mousewheel era, hard as that may be to conceive. Here's how you can watch for, and react to, the user spinning that now-ubiquitous device.

August 11, 2009 · by Karl E. Peterson

Can you handle yet another article on cool tricks you can do with SetWindowsSubclass, which has rapidly become my shiniest new toy? This one's about watching for mousewheel messages. Again, we'll write a drop-in ready class that takes advantage of the HookXP infrastructure I wrote up a short while ago.

Plop the CHookMouseWheel class into a project right alongside any of the others I've already written about and it just works. Setting-up the class is as simple as declaring an instance using WithEvents and handing the host form's hWnd off to begin processing:

```
Private WithEvents m_MW As CHookMouseWheel

Private Sub Form_Load()
   ' Delegate mousewheel handling to class.
   Set m_MW = New CHookMouseWheel
   m_MW.hWnd = Me.hWnd
End Sub
```

The history of mouse wheel support is pretty ragged. Originally, Windows 95 support was non-existent -- only Intellipoint provided a registered message to work with it. Windows 98 and Windows NT 4.0 added some native support, in the form of WM_MOUSEWHEEL messages. Much later, a new message was added for horizontal scrolling. Since so few controls knew about these messages initially, Microsoft adopted the strategy of sending the same message to a window's parent if the window that initially got the message failed to respond to it. Therein lays our opportunity.

We can hook the message stream for a form, and sink mouse wheel messages for all the controls that don't process these messages themselves. I haven't spent any serious time looking at these messages in close to a decade, and was actually caught by surprise that many of the intrinsic VB controls have acquired mouse wheel smarts as the years have gone by. This can only be because the underlying base class was updated, but it's fun to see that in XP almost all of them work just fine. Others, like the native scrollbar controls, still need a power boost.

Details on interpreting the WM_MOUSEWHEEL message are all over the Internet, but as a refresher here's how I handle it (complete Classic VB source) in my IHookXP_Message method:

```
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

   Dim EatIt As Boolean
   Dim Delta As Long
   Dim pt As POINTAPI
   Dim hWndOver As Long
   Dim Button As Long
   Dim Shift As Long
   Dim Cancel As Boolean

   ' Special processing for messages we care about.
   Select Case uiMsg
      Case WM_MOUSEWHEEL, WM_MOUSEHWHEEL
         If m_Enabled Then
            ' Gather all available information about event.
            Button = ReadButtonStates()
            Shift = ReadKeyStates()
            Delta = WordHi(wParam)
            pt.X = WordLo(lParam)
            pt.Y = WordHi(lParam)
            hWndOver = WindowFromPoint(pt.X, pt.Y)

            ' Alert client that wheel event occurred.
            If uiMsg = WM_MOUSEWHEEL Then
               RaiseEvent MouseWheel(hWndOver, _
                  Delta, Shift, Button, pt.X, pt.Y, Cancel)
            Else
               RaiseEvent MouseHWheel(hWndOver, _
                  Delta, Shift, Button, pt.X, pt.Y, Cancel)
            End If

            ' Fire default handler, just in case, but tell Windows
            ' that we handled it regardless.  VB Forms don't react
            ' at all to these messages, but the baseclass for some
            ' controls (eg, textbox) will use it, so it depends on
            ' what the client is subclassing how this will play.
            If Cancel = False Then
               Call HookDefault(hWnd, uiMsg, wParam, lParam)
            End If
            IHookXP_Message = 1  'True
            EatIt = True
         End If

      Case WM_NCDESTROY
         Call Unhook  ' !!!
   End Select

   ' Pass back to default message handler.
   If EatIt = False Then
      IHookXP_Message = HookDefault(hWnd, uiMsg, wParam, lParam)
   End If
End Function
```

Be sure to read my earlier *Subclassing the XP Way* article for a full background on how that fits into the technique I've developed to allow multiple message handlers for any window in your app.

So, now that you have a way to watch for all the mouse wheel messages that fall through the cracks, you need to consider how to handle them when they arrive. If you see a mouse wheel message, in this scheme, it means the control with focus and/or under the cursor didn't react to it. The native scrollbar controls are a good example of this non-functionality. Here's one approach you may want to consider:

```
Private Sub m_MW_MouseWheel(ByVal hWnd As Long, _
   ByVal Delta As Long, ByVal Shift As Long, _
   ByVal Button As Long, ByVal X As Long, _
   ByVal Y As Long, Cancel As Boolean)
      Call AutoScroll(hWnd, Delta, Shift)
End Sub

Private Sub AutoScroll(ByVal hWnd As Long, _
   ByVal Delta As Long, ByVal Shift As Long)

   Dim obj As Object
   ' See what sort of object this handle belongs to,
   ' and act accordingly if it's a scrollbar.
   Set obj = hWndToObject(hWnd)
   If obj Is Nothing Then Exit Sub
   ' If the object is a form, use active control instead.
   If TypeOf obj Is Form Then
      Set obj = obj.ActiveControl
   End If
   ' Act appropriately, if we have a scrollbar.
   Select Case TypeName(obj)
      Case "HScrollBar", "VScrollBar"
         With obj
            On Error Resume Next
            If Shift = vbShiftMask Then
               .Value = .Value + -Sgn(Delta) * .LargeChange
            Else
               .Value = .Value + -Sgn(Delta) * .SmallChange
            End If
         End With
   End Select
End Sub
```

Here, I've written an AutoScroll routine that first determines what sort of control was passed to it, based on the hWnd. Then, if the control is either a vertical or horizontal scrollbar, it adjusts its Value property based on how much the user spun the mouse wheel. The Delta parameter indicates magnitude and direction, with a positive value indicating the wheel was scrolled forward and a negative value indicating the wheel was scrolled backward. It will also be a greater absolute value based on how far the wheel was turned.

In this case, I simply decided to use the sign to indicate direction, and based magnitude on whether or not the user was depressing the Shift key. If not, the scrollbar's Value was incremented by SmallChange, and if Shift was pressed a LargeChange was notched instead. You can make decisions like this based entirely on how it makes most sense within your own application. You could, of course, add other control types to the Select Case block, and handle any variety of controls in a similar manner.

Converting hWnd to Control Reference

So how did I come to know the passed hWnd was a native scrollbar? I had to devise a small set of utility functions that would accept an hWnd and quickly scan the current project for a match. I found situations where I needed to know different sorts of information about the window an hWnd pointed to, so I wrote routines that would return an object's Name, an object's TypeName, or a reference to the object itself:

```
Private Function hWndToObject(ByVal hWnd As Long) As Object
   Dim frm As Form, ctl As Control
   ' Loop all forms and controls in project, looking for a match.
   For Each frm In Forms
      If frm.hWnd = hWnd Then
         Set hWndToObject = frm
         Exit Function
      Else
         On Error Resume Next
            For Each ctl In frm.Controls
               If ctl.hWnd = hWnd Then
                  Set hWndToObject = ctl
                  Exit Function
               End If
            Next ctl
         On Error GoTo 0
      End If
   Next frm
End Function

Private Function hWndToName(ByVal hWnd As Long) As String
   Dim obj As Object
   Set obj = hWndToObject(hWnd)
   On Error Resume Next
   hWndToName = obj.Name
End Function

Private Function hWndToType(ByVal hWnd As Long) As String
   Dim obj As Object
   Set obj = hWndToObject(hWnd)
   hWndToType = TypeName(obj)
End Function
```

In order to determine either the Name or TypeName of an object, you first need to obtain a reference to the object itself, so that's always the first step. The hWndToObject function above works by simply iterating all the controls on all forms in the current application until a match is found. Note the routine isn't limited to controls alone, but also considers whether each form's hWnd is a match.

Given that foundation, the process of determining either Name or TypeName is child's play. Both the hWndToName and hWndToType functions work by calling hWndToObject first, then testing for the desired properties. Error trapping is a must to avoid asking for, say, an hWnd of a windowless control.

I've expanded the HookXP sample on my site to include this new demo of mouse wheel processing. As always when subclassing with native code, be safe. Unhandled errors can be deadly. Save before running.

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.