Out of Context

Click & Retrieve Source CODE!

Replacing default context menus offers new options.

by Karl E. Peterson

Windows 95 and Windows NT 4.0 provide default context menus for nearly everything. Users have grown to expect option menus to appear when they right-click on objects. In many cases, Windows programmers can appreciate the default context menu options because they no longer need to code them. However, in some cases it would be nice to add a few more options to the defaults, or to prevent the context menu from even appearing. Visual Basic offers no native control over context menus. So, we must step outside the boundaries and communicate directly with Windows to accomplish this task.

Windows sends the WM_CONTEXTMENU message to a window as notification that the user has right-clicked the mouse. A program then may pass the message to that window's default window procedure, which will in turn display a default context menu. Alternately, if you design the program to provide a custom context menu, this would be the signal to do so. The problem is, Visual Basic doesn't offer an event to receive this message. I will present a method that uses Zane Thomas' MsgHook OCX, which is available with *Visual Basic 4 How-To*. A freeware version of this control, along with the sample code from this column, is available online in a file called PT0796.ZIP. Download the file from *VBPJ's* Development Exchange on the World Wide Web at http://www.windx.com, or from the *VBPJ* CompuServe Forum, or MSN site. For details, see "How to Reach Us" in Letters to the Editor. Although the code presented uses VB4, the same technique is equally viable in VB3 with any subclassing control. For those using VB3, I've also included the VB3 code for this technique in PT0796.ZIP.

Setting up MsgHook to intercept this message requires just two lines of code in your Form_Load procedure:

```
Msghook1.HwndHook = Text1.hwnd
Msghook1.Message(WM_CONTEXTMENU) = True
```

The first line tells MsgHook which control's messages to monitor. The second line specifies that you want to be notified whenever a window receives the WM_CONTEXTMENU message. If there are multiple messages you want to hook for a single window, you may add more by setting the Message property for each message desired.

From that point on, whenever the user right-clicks the mouse on the control being monitored, MsgHook will fire its Message event. During this event, you have the option of allowing Windows to display a default context menu, providing your own custom context menu, or simply inhibiting display of any context menu (see Listing 1). The first parameter passed to the Message event identifies the message received. Normally, you'd want to test this value to ensure it's the one you want. But because you're only hooking one

VB4

```
Private Sub Msghook1_Message(ByVal msg As Long, _
 ByVal wp As Long, ByVal lp As Long, result As Long)
 Select Case MenuOption
    Case mdDefault

       '
       ' Invoke default window procedure.
       '

       Call Msghook1.InvokeWindowProc(msg, wp, lp)
    Case mdCustom

       '
       ' Pop hidden custom menu, taking the
       ' new MS keyboard into account.
       '

       PreparePopup Text1
       If lp = &HFFFFFFFF& Then
          Me.PopupMenu mPopup, , 0, 0, _
             mCustom(mcSurprise)
       Else
          Me.PopupMenu mPopup, , , , _
             mCustom(mcSurprise)
       End If
    Case mdNone

       '
       ' Do nothing. <g>
       '

 End Select
End Sub
```

LISTING 1 ***What to Do?*** *When the user right-clicks on an object, the application can decide whether to display the default context menu, provide a custom context menu, or inhibit display of any context menu. Windows 95 and Windows NT 4.x provide the WM_CONTEXTMENU message to notify both 16- and 32-bit applications that it's time to decide.*

Karl E. Peterson is a GIS Analyst with a regional transportation planning agency and a member of the Visual Basic Programmer's Journal *Technical Review Board. Based in Vancouver, Washington, he's also an independent programming consultant and a writer. Karl coauthored* Visual Basic 4 How-To, *from Waite Group Press. Online, he's a section leader in the* VBPJ *Forum 32-Bit Bucket and a Microsoft MVP@Large in the MSBASIC Forum. Contact Karl in either CompuServe location at 72302,3707.*

message, you don't need to do that this time.

The WM_CONTEXTMENU message uses wParam to identify the window the message is targeted for, and lParam provides the mouse position—x in the loword and y in the hiword. One gotcha that you do need to be aware of is the new Microsoft Natural Keyboard. This keyboard provides a special key used just to trigger context menus. If the user presses that key, while the mouse isn't over your window, the mouse coordinates passed with WM_CONTEXTMENU are both -1.

To allow display of the default context menu, call MsgHook's InvokeWindowProc method. This method calls the default window procedure for the control, and internal processing proceeds as expected. If you want to provide a custom context menu, it's important that you do *not* allow the default window procedure to receive this message. That's easy enough—just don't invoke the method. Instead, use the PopupMenu method provided by Visual Basic to call up a hidden menu from your form. PopupMenu will use the current mouse position as the default for the menu, but you do need to test whether WM_CONTEXTMENU was generated by the keyboard. Otherwise, your context menu will appear wherever the mouse is, which may be nowhere near your form. If you don't want any context menu to appear, it's even easier—do nothing at all! As an interesting aside, the WM_CONTEXTMENU message is received *between* Visual Basic MouseDown and MouseUp events.

WHAT'S THE CONTEXT?

Providing a custom context menu for edit controls (text boxes) will generally require you to offer the standard options as well. It would be nice if there were a simple way to append options to the default context menu, but there isn't. So you will need to determine which standard options are appropriate at the moment this menu is requested. In this case, you can call the PreparePopup routine when the WM_CONTEXTMENU message is received (see Listing 2). You could code a similar routine to handle your application's Edit menu as well. Call that routine during the Click event for the top-level menu associated with the Edit submenu.

Certain editing options are straightforward. It wouldn't make sense to have Cut, Copy, or Delete enabled if there were no text highlighted. Testing the text box's SelLength property will quickly tell you what to do about those. It also wouldn't make sense, or would certainly frustrate your users, if Paste were enabled when there wasn't text on the clipboard. Use the Clipboard object's GetFormat method to determine this.

Visual Basic provides zero support for the ubiquitous Undo option, however. For this, you must turn to the API. You can send the EM_CANUNDO message to an edit control to determine whether the previous action can be undone. SendMessage will return True if it can, and False if it can't. Set the Enabled property for this menu entry accordingly. If the user selects Undo, use SendMessage again to send an EM_UNDO message to the control. Neither EM_CANUNDO nor EM_UNDO makes use of the wParam or lParam parameters to SendMessage, so you must set both of these to zero.

NOT AGAIN!

Although this problem has been published many times before, it remains one of the more frequently asked questions on CompuServe: how can you prevent multiple instances of your application from running? You can readily determine if another instance of your application is already running by querying the App object's PrevInstance property. Check this value as your first form loads. If it's True, then simply Unload. Problem solved!

It gets slightly more complicated when you want to first transfer focus to the running instance before you unload the new one. You'll need to find the previous instance, then manually activate it. The AppActivate statement comes very close to providing the required functionality. However, it doesn't alter the windowstate of the targeted application, so if it's running as an icon it will remain that way. Again, the API comes to our rescue.

You can use FindWindow to retrieve the window handle of the previous instance's main form if you know its caption (if you know only part of the caption, as with an MDI application, use the FindWindowPartial function I presented on page 113 in the September 1995 Programming Techniques column). Before doing so, make sure the current form's caption isn't identical to the one you're searching for. Once you obtain the handle, you need two more API calls to ensure that the previous instance isn't running as an icon, and to bring it to the foreground. Calling ShowWindow with the SW_RESTORE option meets the first criteria, and SetForegroundWindow takes care of the second. Note that in Win16, SetForegroundWindow doesn't exist, so you'll need to alias SetActiveWindow if you're compiling for both Win16 and Win32 (see Listing 3). If you're using VB3, omit all the 32-bit declarations and conditionals.

WHAT'S A SYSINT?

You may have noticed something strange in the previous topic's listing. I declared the variable used to store a win-

VB4

```
Private Sub PreparePopup(cntl As TextBox)
  '
  ' Can't cut, copy, or delete without
  ' a selection.
  '
  If cntl.SelLength Then
     mCustom(mcCut).Enabled = True
     mCustom(mcCopy).Enabled = True
     mCustom(mcDelete).Enabled = True
  Else
     mCustom(mcCut).Enabled = False
     mCustom(mcCopy).Enabled = False
     mCustom(mcDelete).Enabled = False
  End If
  '
  ' Check clipboard for text to paste.
  '
  If Clipboard.GetFormat(vbCFText) Then
     mCustom(mcPaste).Enabled = True
  Else
     mCustom(mcPaste).Enabled = False
  End If
  '
  ' See if last action can be undone.
  '
  If SendMessage(cntl.hwnd, EM_CANUNDO, 0, 0) Then
     mCustom(mcUndo).Enabled = True
  Else
     mCustom(mcUndo).Enabled = False
  End If
End Sub
```

LISTING 2 ***Setting Standard Options On a Context Menu.*** *Visual Basic provides native methods to determine all but one of the standard context menu options for edit controls. You must utilize the API when deciding whether Undo should be an enabled menu option.*

dow handle like this:

```
Dim hPrev 'As SysInt
```

I readily admit borrowing this idea from Bruce McKinney, author of *Hardcore Visual Basic* (Microsoft Press). One thing that has *frustrated me to no end* is the need to declare certain variables as Integer in Win16 and as Long in Win32. Windows considers them integers, and that's that. But, Visual Basic insists on making a distinction.

In his book, Bruce offers a devilishly sly workaround. I intend to use it as long as I'm writing code for both Win16 and Win32 environments, and I invite you to consider the impact of also adopting the style. It's extremely unlikely that Microsoft will release another 16-bit version of Visual Basic, so this hack will have a short life. Still, it will save you thousands of keystrokes, and probably make your code much more readable as well.

So, what's the secret? Take a deep breath, then ponder the implications of:

```
#If Win32 Then
   DefLng A-Z
#ElseIf Win16 Then
   DefInt A-Z
#End If
```

Stunning in its simplicity! These few lines declare that all nontyped variables will be Long in Win32 and Integer in Win16.

VB4

```
'
' API Declarations
'
#If Win32 Then
  Private Declare Function FindWindow Lib "user32" _
     Alias "FindWindowA" (ByVal lpClassName As _
     String, ByVal lpWindowName As String) As Long
  Private Declare Function ShowWindow Lib "user32" _
     (ByVal hWnd As Long, ByVal nCmdShow As Long) As
Long
  Private Declare Function SetForegroundWindow Lib _
     "user32" (ByVal hWnd As Long) As Long
#ElseIf Win16 Then
  Private Declare Function FindWindow Lib "User" _
     (ByVal lpClassName As Any, ByVal lpWindowName _
     As Any) As Integer
  Private Declare Function ShowWindow Lib "User" _
     (ByVal hWnd As Integer, ByVal nCmdShow As _
        Integer) As Integer
  Private Declare Function SetForegroundWindow _
     Lib "User" Alias "SetActiveWindow" (ByVal hWnd _
     As Integer) As Integer
#End If
Private Const SW_RESTORE = 9

Private Sub Form_Load()
  Dim SearchFor As String
  Dim hPrev 'As SysInt
  If App.PrevInstance Then
     '
     ' Store aside default caption, then
     ' change caption for this instance.
     '
     SearchFor = Me.Caption
     Me.Caption = "ByteMe!@$%^*&#"
     '
     ' Find handle for previous instance.
     ' Restore if needed, and set focus.
     '
     hPrev = FindWindow(vbNullString, SearchFor)
     Call ShowWindow(hPrev, SW_RESTORE)
     Call SetForegroundWindow(hPrev)
     '
     ' Unload current instance.
     '
     Unload Me
  End If
End Sub
```

LISTING 3 ***Bail Out Gracefully.*** *If your main form has a known caption, you can use this method to restore a previous instance of your application before exiting. The previous instance will be restored from an icon (if necessary) and given foreground focus. This technique is equally viable in VB3.*

VB3

```
Sub MdiCaptionUpdate (mFrm As MDIForm, Cap$)
  Dim nRet As Integer
  Dim hWndCap As Integer
  Dim hDCCap As Integer
  '
  ' Set new caption into its window
  '
  mFrm.Caption = Cap$
  '
  ' If the form is minimized, then get the caption
  ' window's window handle, and send it the required
  ' message to initiate a repaint.
  '
  If mFrm.WindowState = 1 Then
     hWndCap = GetCaptionHandle((mFrm.hWnd))
     If hWndCap Then
        nRet = SendMessage(hWndCap, WM_SHOWWINDOW, _
           SW_PARENTCLOSING, 0&)
        hDCCap = GetDC(hWndCap)
        nRet = SendMessage(hWndCap, WM_ERASEBKGND, _
           hDCCap, 0&)
        nRet = ReleaseDC(hWndCap, hDCCap)
     End If
  End If
End Sub

Function GetCaptionHandle (hWndIcon As Integer) As
Integer
  Dim hWndCap As Integer
  Dim Buffer As String
  Dim nRet As Integer
  '
  ' Search the master window list for the icon's
  ' caption, which is itself another window. Identify
  ' a caption window by their unique classname --
  ' #32772.
  '
  hWndCap = GetWindow(hWndIcon, GW_HWNDFIRST)
  Do While hWndCap <> 0
     If GetParent(hWndCap) = hWndIcon Then
        Buffer$ = Space$(128)
        nRet = GetClassName(hWndCap, Buffer$, _
           Len(Buffer$))
        Buffer$ = Left$(Buffer$, nRet)
        If Buffer$ = "#32772" Then
           GetCaptionHandle = hWndCap
           Exit Function
        End If
     End If
     hWndCap = GetWindow(hWndCap, GW_HWNDNEXT)
  Loop
End Function
```

LISTING 4 ***Get Rid of the MDI Icon Caption Bug.*** *VB2 and VB3 contained a bug that prevented the caption from updating on the icon of a minimized MDI form. The MdiUpdateCaption routine uses some API trickery to update an MDI form's caption, no matter what windowstate it's currently in.*

This overrides the default data type of Variant that causes so much grief for so many. Without this block of code, you would need to code the previous example like this:

```
#If Win32 Then
   Dim hPrev As Long
#Else
   Dim hPrev As Integer
#End If
```

If you're using a number of API calls in your applications, you'll end up typing these five lines over and over without resorting to the SysInt trick. I know many of you will object to relying on default behaviors, but you must admit just how much better your code will read and how much faster you can write it using this little trick. I'd even wager that you could avoid a number of common bugs with this automatic type-casting in place. Think about it. Plan on seeing it used in my columns as long as *VBPJ* is covering 16-bit development. I like it. Thanks, Bruce!

SWAT A VB3 BUG

A VB bug that exists in Versions 2 and 3 prevents caption updates on minimized MDIForms. Fortunately, Microsoft fixed this bug in VB4. Normally, when a window's window text is altered, it is also invalidated, which causes a repaint. This invalidation does not occur with MDIForm's caption windows under earlier versions of VB, however. You can work around this by finding the handle to the caption window, and sending a pair of messages that causes it to resize and repaint itself.

This workaround relies on the method Windows uses when creating the windows it manages. The caption below an icon is actually a window. All caption windows in Windows 3.x and Windows NT 3.x share the same classname. By searching for this common trait, you can identify top-level windows of this type.

The GetCaptionHandle function uses the GetWindow API to walk through the master window list (see Listing 4). This function checks each window to see if it's an icon caption window by calling the GetClassName API, watching for the "#32772" classname. After finding a caption window, you will need to call the GetParent API. The related icon is the parent of its caption window, so calling the GetParent API will either confirm a good hit or tell you to keep looking. After the GetCaptionHandle function checks all top-level windows, it will return an invalid window handle (0) if none match the criteria.

Setting the Caption property *does* change the associated window text (as can be shown with the GetWindowText API), but Visual Basic isn't forcing the repaint. The MdiCaptionUpdate routine first sets the MDIForm's Caption property. If the WindowState is not minimized, this is sufficient. If the WindowState is minimized, and the call to GetCaptionHandle succeeds, two messages are sent to the caption window. Windows is first fooled into thinking that the parent form is just now being minimized with the WM_SHOWWINDOW message. This causes Windows to resize the caption window to fit its current window text.

Next, the WM_ERASEBKGND message invalidates the entire caption window, and therefore causes a repaint. Because the WM_ERASEBKGND message must include a device context handle in wParam, calls to the GetDC and ReleaseDC APIs are on either side of the SendMessage call. The final result is a correctly updated caption. ×