Subclass Your Way Around VB's Limitations

These eight subclassing techniques enable you to customize your apps in ways that Visual Basic native code doesn't allow.

BY JONATHAN WOOD AND KARL E. PETERSON

The thing that makes Visual Basic great is the fact that it allows you to write full-fledged Windows programs without requiring that you understand how Windows operates. Because Visual Basic hides many of these complexities, program development is much faster and easier. However, if you've been programming Windows for a while, you know that with this ease of use come some limitations. In order to grab all the functionality that Windows provides, it is sometimes necessary to dig a little deeper into how Windows works.

One example of these inner details is the Windows messaging system. When an event occurs, Windows sends messages to the affected windows. For example, when the user resizes a window, Windows sends a WM_SIZE message to the window being resized. Visual Basic does not expose this message directly but instead provides a Resize event handler that Visual Basic calls when it receives this message.

However, other messages, such as WM_SYSCOMMAND, which is sent when the user selects a command from the window's system menu, have no corresponding Visual Basic event handler. Visual Basic provides no direct support for responding to these messages.

Fortunately, Visual Basic's design is made considerably more flexible by its support for VBX custom controls. A number of such controls have been developed to allow you to respond to any message sent to a particular window. These controls are commonly called subclassing controls. A subclassing control provides a Message event handler that is called when Windows sends selected messages to the window you specify.

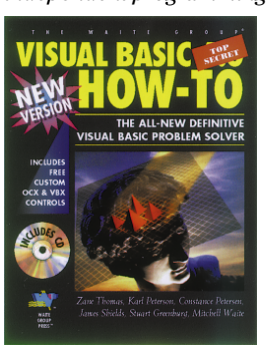
In this article, we'll show you our eight favorite subclassing techniques. These techniques demonstrate tasks that are only possible in Visual Basic using a subclassing control. But before we present these examples, let's discuss how Windows messages and subclassing controls work.

What exactly does it mean when we say "send a message to a window?" Under Microsoft Windows, each window has an associated window procedure that defines the behavior of that window. Often, the same procedure is used by more than one window. For example, all standard text boxes are associated with the same window procedure. When Windows detects an event such as a key being pressed, or the user pressing the left mouse button, Windows calls the procedure associated with the affected window. Window procedures are normally written in C and can't be written in Visual Basic. If you could write a window procedure in Visual Basic, it might look something like this:

```
Function WinProc (hWnd As Integer, _
     msg As Integer, wParam As _
     Integer, lParam As Long) As Long
     Select Case msg
          Case WM_PAINT
          ' Code to paint the window
          Case WM_LMOUSEDOWN
```

Jonathan Wood writes commercial and custom software in Visual Basic, Visual C++, and assembly language. His company, SoftCircuits, is located in Irvine, California. Reach him on the VBPJ *Forum (he's the section leader of the DLL/API Lab section) and MSBASIC Forum on CompuServe at 72134,263.*

Karl E. Peterson is a GIS Analyst with a regional transportation planning agency, an independent programming consultant, and a writer based in Vancouver, Washington. He recently coauthored Visual Basic 4.0 How-To, *from the Waite Group Press. He's the 32-Bit Bucket Section Leader for the* VBPJ *Forum and a Microsoft MVP in the MSBASIC Forum. Contact Karl in either CompuServe location at 72302,3707*



The code accompanying this article (and the subclassing tool used here) is available on the VBPJ *Forum on CompuServe—GO VBPJFORUM to get there. All code for this issue will be in the Magazine Library). CompuServe is now available directly from the Internet. Go to CompuServe's home page to find out how at http://www.compuserve.com. If you don't have access to CompuServe, the code will be available in a future issue of the VB-CD Quarterly. Call 800-848-5523 to order a subscription to the VB-CD Quarterly.*

```
        ' Code to handle left mouse
        ' press

        ' and so on...

        Case Else
        ' Pass unprocessed messages
        ' back to Windows by calling
        ' the DefWindowProc API
        ' function
            WinProc = _
                DefWindowProc(hWnd, _
                 msg, wParam, lParam)
    End Select
End Function
```

The first argument, hWnd, is a handle to the window the message is directed at. If the procedure needs to perform any window-related tasks, it can use this handle to identify the window. The second argument, msg, is the actual message being sent: this is simply one of the predefined values used by Windows that indicate which event has occurred. It can also be a value defined and used internally by a particular application.

The remaining arguments are generic. The contents of the arguments depend on the message being sent and are used to describe information about the event that has occurred. Sometimes one or both of these last two arguments are not used. All unprocessed messages should be passed back to Windows for default processing by calling the DefWindowProc API function.

To provide this same functionality in Visual Basic, a subclassing control allows you to specify the handle of the window you want to subclass. This is usually the hWnd property for a form, but it can also be the handle for a control or other window. The subclassing control also lets you specify the messages you want to intercept. When Windows sends the specified messages to the specified window, the subclassing control fires the Message event. To respond to these messages, place your code in the Message event handler, which looks somewhat similar to the WinProc function.

The examples presented in this article were written to use a subclassing control called MsgHook. This VBX, written by Zane Thomas, is a modified version of the MsgHook control distributed with the second edition of the Waite Group Press book, *Visual Basic How-To*. This control was recently rewritten to be compatible with a new 32-bit MSGHOOK.OCX, to be included with the book *Visual Basic 4.0 How-To,* also from Waite Group Press. This means that code written to use this control can be ported to the 32-bit world of Visual Basic 4.0 with only minimal modification to function parameter data types—changing Integers to Longs, gen-

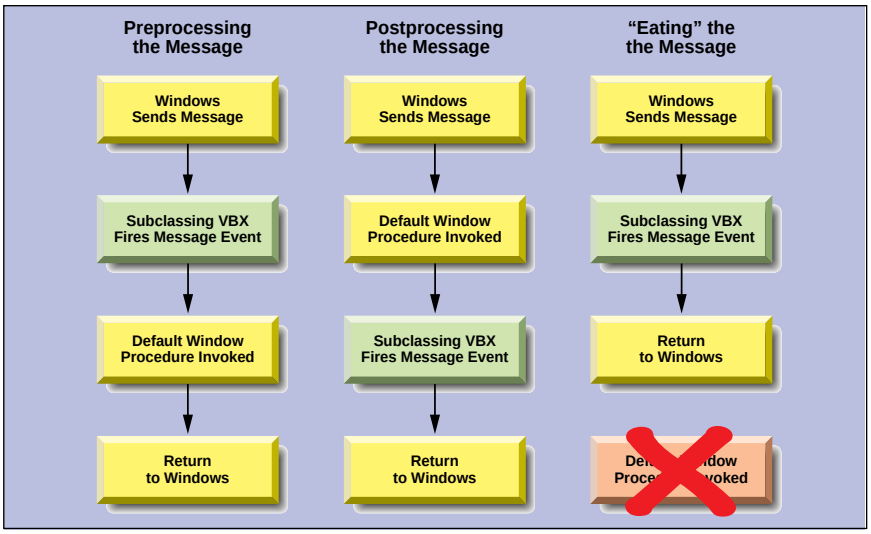


FIGURE 1 ***Three Common Controls.*** *These diagrams show the three most common methods subclassing controls offer for responding to window messages. A control may insert itself so that it receives messages either before or after the default window procedure does. It may also simply "eat" the message, never allowing it to pass through to the default window procedure. MsgHook offers the most flexibility: you can invoke the default window procedure at any point. This allows you to combine the first two methods, incorporating both pre- and post-processing.*

```
Sub Form_Load ()

    ' Setup MsgHook
    MsgHook.HwndHook = Me.hWnd
    MsgHook.Message(WM_MENUSELECT) = True

    DefaultText = "Ready"
    lblStatusBar = DefaultText

End Sub

Sub MsgHook_Message (msg As Integer, wParam As Integer, _
   lParam As Long,result As Long)
   Dim txtStatus As String

   If msg = WM_MENUSELECT Then
       Select Case wParam
          Case 0
             txtStatus = DefaultText
          Case 2
             txtStatus = "Exits this program"
          Case 4
             txtStatus = "Cuts the selected items" & _
                 " to the clipboard and deletes them"
          Case 5
             txtStatus = "Copies the selected " & "items to the clipboard"
          Case 6
             txtStatus = "Pastes the contents of " & _
                "the clipboard to the current location"
          Case 7
             txtStatus = "Deletes the selected items"
          Case Else
             txtStatus = ""
       End Select
       lblStatusBar = txtStatus
       result = 0
    End If

End Sub
```

LISTING 1 ***Displaying Information About Menu Commands in a Status Bar.*** *As each menu command is highlighted, Windows sends a WM_MENUSELECT message to the window that owns the menu. Once you figure out the method Visual Basic uses to determine menu IDs, you can have your application display a brief tip in a status bar that corresponds to the currently selected command.*

erally. You may freely distribute MSGHOOK.VBX with any applications you write (MSGHOOK.VBX along with the full source code for all the examples presented in this article plus a few more will be posted on CompuServe in the *VBPJ* Forum as SUBCLS.ZIP. We weren't able to fit all of our code in the listings printed here).

You should find the code easily adaptable to any of the other subclassing controls currently available, such as SpyWorks-VB from Desaware, or MSGBlaster from WareWithAll Inc. (Both Desaware and WareWithAll have demo versions of their products in the third-party section in the *VBPJ* Forum on CompuServe.)

Such controls will use a similar syntax for the Message event handler but may have slightly different parameters. The syntax for specifying which messages you want to respond to and for specifying how default processing is handled will probably be different.

Default processing specifies when or if the message is sent to the original window procedure (see Figure 1). For example, should the window process the message before or after the Message event is fired? Or should messages you process not be sent to the original window procedure at all? To control default processing, MsgHook provides an InvokeWindowProc function that you call when you want the original window procedure to process a message. You can simply call this function in any order you want, or not at all. Note that this function is new, and is not supported by the VBX that came with the second edition of *Visual Basic How-To*.

1 SHOWING STATUSBAR INFORMATION FOR MENU COMMANDS

A window receives a WM_MENUSELECT message each time a new menu command is highlighted. By processing this message, you can create a status bar that provides the user with a brief description of menu commands as each command is highlighted.

The WM_MENUSELECT message uses the wParam parameter to indicate the ID of the menu command being selected. How do you know which ID value corresponds to which menu command? A little experimenting shows that Visual Basic uses consecutive numbering for menu commands starting from 1. For example, say you have a File menu and an Edit menu, and that each menu has five commands. The ID for File will be 1, and the ID for the File menu's commands will be 2, 3, 4, 5, and 6. The ID for Edit will then be 7, and the ID for the Edit commands will be 8, 9, 10, 11, and 12, and so on. If wParam is zero, no menu command is selected and your program should either clear the status bar, or place some default text in the status bar (see Listing 1). To create the status bar, the code in Listing 1 uses a simple label control. This allows the code to work with both the professional and standard editions of Visual Basic. However, the technique is easily adapted to work with a more professional-looking status bar if you have access to one.

2 ADDING A COMMAND TO A FORM'S SYSTEM MENU

Using the Windows API, it is easy to add a command to a form's system menu or control box (see Listing 2). This can be useful in simple applications that have only one or two menu commands. Because Visual Basic doesn't know about the menu command your program added, no event handler exists to respond when the command is selected. This can be solved using a subclassing control to intercept the WM_SYSCOMMAND message.

Windows sends the WM_SYSCOMMAND message when the user selects a command from the system menu, or clicks on the maximize or minimize buttons. The wParam parameter contains an identifier that indicates the requested system command.

The listings use GetSystemMenu to get a handle to the form's system menu and then uses the AppendMenu API function to append a menu separator followed by an About command. Note that the ID assigned to the new menu item must be less than &HF000 so that it does not conflict with any of Windows' system commands.

The code also invokes MsgHook's InvokeWindowProc method to call the form's original window procedure, if WM_SYSCOMMAND was not sent in response to the user selecting the new menu

```
Sub Form_Load ()
    Dim i As Integer
    Dim hMenu As Integer

    ' Add "About..." command to system menu
    hMenu = GetSystemMenu(Me.hWnd, False)
    i = AppendMenu(hMenu, MF_SEPARATOR, 0, 0&)
    i = AppendMenu(hMenu, MF_STRING, IDM_ABOUT, "&About...")

    ' Setup MsgHook
    MsgHook.HwndHook = Me.hWnd
    MsgHook.Message(WM_SYSCOMMAND) = True

End Sub

Sub MsgHook_Message (msg As Integer, wParam As Integer, _
    lParam As Long, result As Long)

    ' Look for WM_SYSCOMMAND message with About command
    If msg = WM_SYSCOMMAND Then
       Select Case wParam
          Case IDM_ABOUT
             frmAbout.Show 1
             result = 0
             Exit Sub
       End Select
    End If

    ' Pass along to default handler if message not processed
    result = InvokeWindowProc(MsgHook.HwndHook, msg, _
       wParam, lParam)

End Sub
```

LISTING 2 ***Adding a Menu Command to a Form's System Menu.*** *Adding a command to a form's system menu is no problem with the Windows API. But having your application respond when the command is selected requires a subclassing control to intercept the WM_SYSCOMMAND message.*

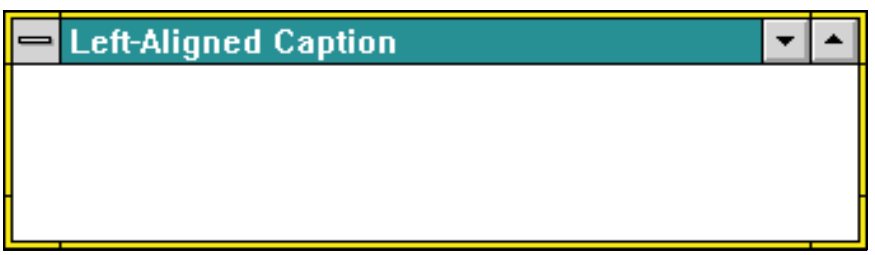


FIGURE 2 ***Left-Justifying Caption Text.*** *By intercepting messages that force a repaint of the nonclient areas of your form, you can create customized title bars. The same techniques could be used to add bitmaps or even buttons to this part of the form.*

command. This allows other instances of this message to be processed normally. Failure to do this would effectively disable all other system commands.

3 SUPPORTING DRAG-AND-DROP FROM FILE MANAGER

Applications that process files seem not quite as polished if they don't accept files dragged from File Manager or the Windows 95 Explorer shell. Adding such support to a Visual Basic app requires modifying the extended window attributes of your form, notifying Windows that you wish to accept dropped files, and finally hooking the WM_DROPFILES message using a subclassing control.

Visual Basic does not set the WS_EX_ACCEPTFILES style attribute for forms, so this must be done first. The AcceptDrops routine called from a Form_Load event (see Listing 3) first reads the extended style attributes for the form, then modifies these to include this new setting. Calling the DragAcceptFiles API registers your form with Windows to receive notification when files are dropped.

Within the Message event, the DragQueryFile API is first used to establish the number of files dropped, then to retrieve their file names one by one and place them into a list box. The Message event's wParam parameter contains a handle that identifies an internal data structure describing the dropped files, and is used with the other Drag API functions. This handle is valid only during the processing of this message. The DragFinish API releases memory Windows allocated for transfering file names to the application. Because Visual Basic doesn't normally support these operations, there is no need to invoke the original window procedure.

4 LEFT-JUSTIFYING CAPTION TEXT

Ever feel like altering a Windows standard to do it *your* way? This example hooks all the messages that cause repaints of nonclient areas (title bar, menus, system icons, and borders) to occur. Immediately after allowing the default window processing, the title bar portion of the nonclient area is repainted with a customized Windows 95-style caption (see Figure 2). You could use the same techniques to add to the standard caption, perhaps with the time on the far right, or to otherwise alter the title bar to your taste.

The WM_NCPAINT and WM_NCACTIVATE messages notify applications that their nonclient areas need to be repainted. But because this includes other areas besides the caption, the original window procedure must do the bulk of the work before we step in to paint a custom caption. By setting the form's Caption property to a null string, a temporary flash in the title bar is prevented as the caption is painted centered and then re-

CONTINUED ON PAGE 43.

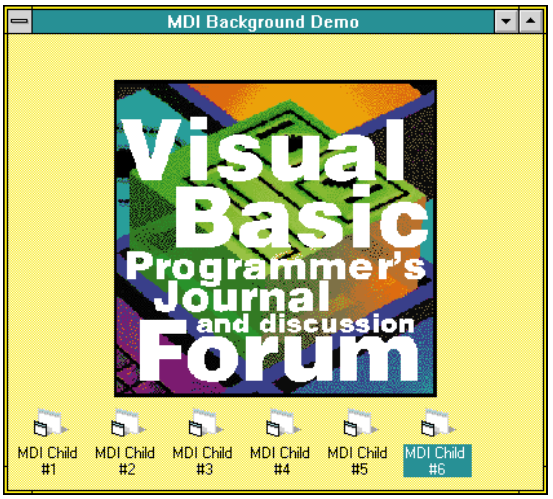


FIGURE 3 ***MDI Wallpaper.*** *You can add your company's logo or other interesting bitmaps or patterns to an MDI background by intercepting WM_PAINT and WM_ERASEBKGND for the MDI client space.*

```
' *** Listing from DROPFILE.BAS ***
Sub AcceptDrops (hWnd As Integer)
  Dim Style As Long
  '
  ' Set to accept dropped files from File Manager
  '
  Style = GetWindowLong(hWnd, GWL_EXSTYLE)
  Style = SetWindowLong(hWnd, GWL_EXSTYLE, Style _
    Or WS_EX_ACCEPTFILES)
  '
  ' Notify system we want to accept dropped files
  '
  DragAcceptFiles hWnd, True
End Sub

' *** Listing from DROPFILE.FRM ***
Sub Form_Load ()
  '
  ' Prepare form to accept dropped files
  '
  Call AcceptDrops((Me.hWnd))
  '
  ' Setup MsgHook control
  '
  MsgHook.HwndHook = Me.hWnd
  MsgHook.Message(WM_DROPFILES) = True
End Sub

Sub MsgHook_Message (Msg As Integer, wParam As _
  Integer, lParam As Long, Result As Long)
  Dim nFiles As Integer
  Dim Buffer As String
  Dim i As Integer
  Dim nRet As Integer
  '
  ' Always test which message was recieved.
  '
  If Msg = WM_DROPFILES Then
    '
    ' Set up buffer to recieve filenames, then
    ' retrieve number of files dropped by passing
    ' -1 as the file number.
    '
    Buffer = Space$(256)
    nFiles = DragQueryFile(wParam, -1, _
      Buffer, Len(Buffer))
    '
    ' Clear list box and reset label.
    '
    List1.Clear
    Label1 = nFiles & " File(s) Dropped:"
    '
    ' Retrieve the name of each file dropped, and
    ' place in listbox.
    '
    For i = (nFiles - 1) To 0 Step -1
      nRet = DragQueryFile(wParam, i, _
        Buffer, Len(Buffer))
      List1.AddItem Left(Buffer, nRet), 0
    Next i
    List1.ListIndex = 0
    '
    ' Tell system we're done.  No need to invoke
    ' original window procedure.
    '
    Call DragFinish(wParam)
    Result = 0
  End If
End Sub
```

LISTING 3 ***Supporting Drag-and-Drop from File Manager.*** *The names of files dragged from File Manager and dropped on your form can be retrieved using this code.*

```
' *** Listing from LEFTCAP.FRM ***
' MsgHook routine which calls default window procedure
Declare Function InvokeWindowProc Lib _
  "MsgHook.vbx" (ByVal hWnd As Integer, _
  ByVal Msg As Integer, ByVal wParam As _
  Integer, ByVal lParam As Long) As Long

' Store Caption as a string
Dim Kaption As String

Sub Form_Load ()
  '
  ' Set Caption to "", storing whatever was
  ' set at design
  '
  Kaption = Me.Caption
  Me.Caption = ""
  '
  ' Setup MsgHook control
  '
  MsgHook.HwndHook = Me.hWnd
  MsgHook.Message(WM_NCPAINT) = True
  MsgHook.Message(WM_NCACTIVATE) = True
  MsgHook.Message(WM_SIZE) = True
End Sub

Sub MsgHook_Message (Msg As Integer, wParam As Integer, _
   lParam As Long, Result As Long)
  Static PrevState As Integer
  '
  ' Fire default window procedure before processing
  ' any of the messages we're interested in for this
  ' task.
  '
  Result = InvokeWindowProc(MsgHook.HwndHook, _
    Msg, wParam, lParam)
  '
  ' Check which message arrived, and act accordingly.
  '
  Select Case Msg
    Case WM_NCPAINT
      '
      ' Check whether to paint as active or inactive
      '
      RefreshCaption Kaption, Me, (GetActiveWindow() = _
         Me.hWnd)
    Case WM_NCACTIVATE
      '
      ' wParam indicates active or inactive
      '
      RefreshCaption Kaption, Me, wParam
    Case WM_SIZE
      '
      ' Supply Caption for minimized icon only
      '
      If wParam = SIZE_MINIMIZED Then 'Minimized
        Me.Caption = Kaption
      ElseIf PrevState = SIZE_MINIMIZED Then
        Kaption = Me.Caption
        Me.Caption = ""
        RefreshCaption Kaption, Me, True
      End If
      '
      ' Store "last" WindowState
      '
      PrevState = Me.WindowState
  End Select
End Sub

' *** Listing from LEFTCAP.FRM ***

Sub RefreshCaption (CapText$, Frm As Form, fActive%)
  Dim nRet As Long
  Dim wDC As Integer
  Dim wr As RECT
  Dim xText As Integer
  Dim yText As Integer
  Static xIcon As Integer
  Static yIcon As Integer
  Static xBorder As Integer
  Static yBorder As Integer
  Static BeenHere As Integer
  '
  ' Bail out if form is minimized
  '
  If Frm.WindowState = SIZE_MINIMIZED Then
    Exit Sub
  End If
  '
  ' Retrieve system metrics if first time here
  '
  If Not BeenHere Then
    xIcon = GetSystemMetrics(SM_CXSIZE)
    yIcon = GetSystemMetrics(SM_CYSIZE)
    If Frm.BorderStyle = 1 Then 'FixedSingle
      xBorder = GetSystemMetrics(SM_CXBORDER)
      yBorder = GetSystemMetrics(SM_CYBORDER)
    ElseIf Frm.BorderStyle = 2 Then 'Sizable
      xBorder = GetSystemMetrics(SM_CXFRAME)
      yBorder = GetSystemMetrics(SM_CYFRAME)
    ElseIf Frm.BorderStyle = 3 Then 'FixedDouble
      xBorder = GetSystemMetrics(SM_CXDLGFRAME)
      yBorder = GetSystemMetrics(SM_CYDLGFRAME)
    End If
    BeenHere = True
  End If
  '
  ' Get device context for entire window
  '
  wDC = GetWindowDC(Frm.hWnd)
  '
  ' Determine space required by text
  '
  nRet = GetTextExtent(wDC, CapText, Len(CapText))
  xText = WordLo(nRet)
  yText = WordHi(nRet)
  '
  ' Calc rectangle to put text into
  '
  Call GetWindowRect(Frm.hWnd, wr)
  wr.right = wr.right - wr.left - _
    (xIcon * 2) - xBorder - 2
  wr.left = xBorder + xIcon + 4
  wr.top = yBorder + ((yIcon - yText) \ 2)
  wr.bottom = yBorder + yIcon
  '
  ' Retrieve and set colors to use for
  ' titlebar and text
  ' Set background drawing mode
  '
  If fActive Then
    nRet = SetBkColor(wDC, _
       GetSysColor(COLOR_ACTIVECAPTION))
    nRet = SetTextColor(wDC, _
       GetSysColor(COLOR_CAPTIONTEXT))
  Else
    nRet = SetBkColor(wDC, _
       GetSysColor(COLOR_INACTIVECAPTION))
    nRet = SetTextColor(wDC, _
       GetSysColor(COLOR_INACTIVECAPTIONTEXT))
  End If
  '
  ' Draw the caption text
  '
  nRet = ExtTextOut(wDC, wr.left, wr.top, _
    ETO_CLIPPED Or ETO_OPAQUE, wr, _
    CapText, Len(CapText), ByVal 0&)
  '
  ' Release window device context
  '
  nRet = ReleaseDC(Frm.hWnd, wDC)
End Sub
```

LISTING 4 ***Left-Justifying Caption Text.*** *The code used to modify the appearance of caption text could be altered to provide any number of special effects. Studying the RefreshCaption routine offers an appreciation for just how much Visual Basic and Windows do without calling attention to themselves.*

```
' *** Listing from GETMINMX.FRM ***
Sub Form_Load ()

   ' Setup MsgHook
  MsgHook.HwndHook = Me.hWnd
  MsgHook.Message(WM_GETMINMAXINFO) = True

End Sub

Sub MsgHook_Message (msg As Integer, wParam As _
  Integer, lParam As Long, result As Long)
   Dim MinMax As MINMAXINFO

   If msg = WM_GETMINMAXINFO Then

      ' Copy to our local MinMax variable
      hmemcpy MinMax, ByVal lParam, Len(MinMax)

      ' Set minimum/maximum tracking size
      MinMax.ptMinTrackSize.x = 150
      MinMax.ptMinTrackSize.y = 150
      MinMax.ptMaxTrackSize.x = 400
      MinMax.ptMaxTrackSize.y = 400

      ' Copy data back to Windows
      hmemcpy ByVal lParam, MinMax, Len(MinMax)
      result = 0
   End If

End Sub
```

LISTING 5 ***Restricting Window Sizing.*** *By intercepting the WM_GETMINMAXINFO message, it is possible to restrict the range that a user can size a window. One nice thing about this approach is that it actually restricts the size of the sizing rectangle, giving the user visual feedback about how the window can be sized as the window is being resized.*

```
' *** Listing from MDIPAINT.FRM ***

Sub MDIForm_Load ()
  Dim F As Form
  Dim i As Integer
  '
  ' Setup MsgHook control
  '
  MsgHook.HwndHook = (GetWindow(Me.hWnd, GW_CHILD))
  MsgHook.Message(WM_PAINT) = True
  MsgHook.Message(WM_ERASEBKGND) = True
  '
  ' Place a few children out in the client space
  '
  For i = 1 To 6
    Set F = New Form1
    F.Caption = F.Caption & i
  Next i
End Sub

Sub MsgHook_Message (Msg As Integer, wParam As _
  Integer, lParam As Long, Result As Long)
  Select Case Msg
    Case WM_PAINT
      '
      ' Paint a nice background, then allow default
      ' window procedure to run (which instructs
      ' icons to repaint and clears the update status).
      '
      mdiBitBltCentered Picture2, Me, _
         GetSysColor(COLOR_APPWORKSPACE)
      Result = InvokeWindowProc(MsgHook.HwndHook, _
        Msg, wParam, lParam)
    Case WM_ERASEBKGND
      '
      ' Return non-zero to indicate we do "erase".
      ' No need to invoke default window procedure.
      '
      Result = 1
  End Select
End Sub

Sub mdiBitBltCentered (Src As PictureBox, Dest _
  As MDIForm, FillColor As Long)
  Dim nRet As Integer
  Dim dDC As Integer, dWnd As Integer, cDC As Integer
  Dim sR As RECT, dR As RECT
  Dim hBmp As Integer, oldBmp As Integer
  Dim hBrush As Integer
  Dim dX As Integer, dY As Integer
  '
  ' Get DC to client space
  '
  dWnd = GetWindow(Dest.hWnd, GW_CHILD)
  dDC = GetDC(dWnd)
  '
  ' Get source and destination rectangles
  '
  Call GetClientRect(Src.hWnd, sR)
  Call GetClientRect(dWnd, dR)
  '
  ' Create a memory bitmap to build image in
  '
  cDC = CreateCompatibleDC(dDC)
  hBmp = CreateCompatibleBitmap(dDC, _
    dR.right, dR.bottom)
  oldBmp = SelectObject(cDC, hBmp)
  '
  ' Create new brush and paint background in memory
  '
  hBrush = CreateSolidBrush(FillColor)
  nRet = FillRect(cDC, dR, hBrush)
  '
  ' Calc upper-left position parameters to place image
  '
  dX = (dR.right - sR.right) \ 2
  If dR.bottom > sR.bottom Then
    dY = (dR.bottom - sR.bottom) \ 3
  Else
    dY = (dR.bottom - sR.bottom) \ 2
  End If
  '
  ' BitBlt first to memory DC,
  ' then from memory to screen
  '
  nRet = BitBlt(cDC, dX, dY, sR.right, sR.bottom, _
     Src.hDC, 0, 0, SRCCOPY)
  nRet = BitBlt(dDC, 0, 0, dR.right, dR.bottom, cDC, _
     0, 0, SRCCOPY)
  '
  ' and clean up
  '
  nRet = DeleteObject(hBrush)
  nRet = SelectObject(cDC, oldBmp)
  nRet = DeleteObject(hBmp)
  nRet = DeleteDC(cDC)
  nRet = ReleaseDC(dWnd, dDC)
End Sub
```

LISTING 6 ***Painting the Background of an MDIForm.*** *Drawing directly on the background of an MDIForm only requires knowing that it's actually the first child (in a non-MDI sense) window of the form. Hooking WM_PAINT and WM_ERASEBKGND messages for this window allows you to paint whatever pleases you; however, only GDI drawing methods are available.*

HOME

CONTINUED FROM PAGE 38.

painted on the left. Hooking the WM_SIZE message allows the Caption property to be restored whenever the form is minimized, preventing the need to paint the icon's caption as well.

Within the Message event (see Listing 4), InvokeWindowProc is called immediately to allow all normal nonclient painting, with its return value set aside in the Result parameter for later return to Windows. A generic RefreshCaption routine is used to handle all the custom painting details, which are surprisingly involved for what seems to be such a simple process. It would be routine to replace or modify other special effects within the title bar. It's critical with such an event handler that it be optimized for the quickest possible execution. Any delays in drawing nonclient elements will be highly apparent to your users.

5 RESTRICTING A WINDOW SIZE RANGE

Windows sends a WM_GETMINMAXINFO message when a window is being resized. By modifying the MINMAXINFO data structure, you can restrict the size to which the window can be sized.

While Visual Basic doesn't support pointers, the Windows API provides a function called hmemcpy that can be used to copy data from one location to another. Using hmemcpy, this example copies the data from the address specified by lParam to a Visual Basic variable of type MINMAXINFO. Note that the code declares two hmemcpy parameters As Any—this provides maximum flexibility, but it also means that Visual Basic is not able to verify that the correct data types are sent. Take care when incorporating hmemcpy in your own applications.

Setting the ptMinTrackSize and ptMaxTrackSize portions of the MINMAXINFO data structure indicates the minimum and maximum size, in pixels, to which the window can be sized (see Listing 5). Modifying the MINMAXINFO data not only allows you to restrict the size of the window, but it also restricts the sizing rectangle that appears while the user is sizing the window.

6 PAINTING THE BACKGROUND OF AN MDIFORM

Drawing directly on the background of an MDIForm requires but one piece of secret knowledge—that the client space is actually the first child window (in a non-MDI sense) of the form, a handle to which may be obtained using the GetWindow API. MsgHook is used to hook the WM_PAINT and WM_ERASEBKGND messages for this window. In this manner the Message event is transformed into the equivalent of an MDIForm_Paint event during which you may paint whatever pleases you (see Figure 3). However, only GDI drawing methods are available.

When the background window receives a WM_ERASEBKGND message you must prevent the original window procedure from gaining control by returning 1 to Windows as the Result parameter (see Listing 6). Otherwise, the background would indeed be erased, thus causing a distracting flash before it is repainted. When the WM_PAINT message is received, any sort of custom routine you desire

```
Sub Form_Load ()
    Dim hMenu As Integer
    Dim i As Integer, j As Integer
    Dim nID As Integer

    ' Get handle to "Colors" menu
    hMenu = GetMenu(Me.hWnd)
    hMenu = GetSubMenu(hMenu, 1)

    ' Modify commands to be owner-draw and to contain
    ' color info
    For i = 0 To 7
        ' Get menu ID
        j = GetMenuItemID(hMenu, i)
        ' Modify menu item (command ID is maintained)
        j = ModifyMenu(hMenu, j, MF_BYCOMMAND Or _
          MF_OWNERDRAW, j, QBColor(8 + i))
    Next i

    ' Setup MsgHook
    MsgHook.HwndHook = Me.hWnd
    MsgHook.Message(WM_DRAWITEM) = True
    MsgHook.Message(WM_MEASUREITEM) = True

End Sub

Sub MsgHook_Message (msg As Integer, wParam As _
   Integer, lParam As Long, result As Long)

    Dim tmp As Integer, rc As RECT
    Dim hBrush As Integer, hOldBrush As Integer
    Dim DrawInfo As DRAWITEMSTRUCT
    Dim MeasureInfo As MEASUREITEMSTRUCT

    Select Case msg

        Case WM_DRAWITEM
           If wParam = 0 Then  'If sent by menu
              ' Copy DRAWINFOSTRUCT data to
              ' local variable
              Call hmemcpy(DrawInfo, ByVal lParam, _
                 Len(DrawInfo))
              ' Paint area around color bar
              If DrawInfo.itemState And ODS_SELECTED Then
                 hBrush = CreateSolidBrush _
                  (GetSysColor(COLOR_HIGHLIGHT))
              Else
                 hBrush = CreateSolidBrush(GetSysColor _
                  (COLOR_MENU))
              End If
              rc = DrawInfo.rcItem
              tmp = FillRect(DrawInfo.hDC, rc, hBrush)
              tmp = DeleteObject(hBrush)
              ' Paint color bar
              tmp = (rc.bottom - rc.top) / 5
              Call InflateRect(rc, -tmp, -tmp)
              hBrush = CreateSolidBrush _
                 (DrawInfo.itemData)
              hOldBrush = SelectObject(DrawInfo.hDC, _
                 hBrush)
              tmp = Rectangle(DrawInfo.hDC, rc.left, _
                 rc.top, rc.right, rc.bottom)
              tmp = SelectObject_
                  (DrawInfo.hDC, hOldBrush)
              tmp = DeleteObject(hBrush)
           End If

        Case WM_MEASUREITEM
           ' Copy MEASUREITEMSTRUCT to local variable
           Call hmemcpy(MeasureInfo, ByVal lParam, _
              Len(MeasureInfo))
           ' Tell Windows how big our
           ' owner-draw items are
           MeasureInfo.itemWidth = 70
           MeasureInfo.itemHeight = GetSystemMetrics _
              (SM_CYMENU)
           ' Copy MEASUREITEMSTRUCT data back to Windows
           Call hmemcpy(ByVal lParam, MeasureInfo, _
              Len(MeasureInfo))

        Case Else

    End Select

End Sub
```

LISTING 7 ***Creating Owner-Draw Menu Commands.*** *This code re-creates several menu commands to give them the owner-draw style. As a result, Windows sends a WM_DRAWITEM message whenever these commands need to be painted. Steps are also taken to make sure the menu commands have their original menu ID so that Visual Basic is still able to call the menu command event handlers.*

may be called to place graphics or text within the client space. Follow painting with a call to InvokeWindowProc, which cleans up a little, instructing minimized MDI children to repaint their icons and captions as well as clearing the flag that indicates this window is in need of update. The return value of InvokeWindowProc is passed back in the Result parameter.

We've written a routine called mdiBitBltCentered, which copies a bitmap stored in a hidden picture control to the center of the MDI client space when called. It does this by first constructing a memory device context in which to build up an image for the entire window. The background is filled with the system color the user has chosen for MDI backgrounds (called the Application Workspace in the Control Panel), and the hidden picture control's contents are BitBlted to the center of this memory bitmap. Finally, the entire memory bitmap is BitBlted to the client space of the MDI form.

7 DRAWING CUSTOM MENU COMMANDS

Windows allows applications to create owner-draw controls. Using owner-draw controls, you can have Windows handle all the logic it normally handles for a control, except that it doesn't draw the control. Instead, Windows sends a WM_DRAWITEM message to your application when the control needs painting. This allows you to make the control appear in any manner you prefer, without requiring that you create the control from scratch.

Unfortunately, Visual Basic doesn't provide direct support for owner-draw controls. Using a subclassing control, you can intercept the WM_DRAWITEM message; however, things are a little more complicated than that. The trouble is that in order to have Windows send the WM_DRAWITEM message, the control must be created with the owner-draw attribute—this presents a problem, because Visual Basic automatically creates controls.

DLLs such as Desaware's SpyWorks-VB, and VBASM.DLL (see Programming Techniques, *VBPJ* August 1995) provide tools for re-creating a control with different attributes. Unfortunately, it is not clear that this technique will work under future versions of Visual Basic.

Fortunately, owner-draw menus can be created quite easily (see Listing 7). Listing 7 demonstrates processing the WM_DRAWITEM message by creating some owner-draw menu commands. The code uses the ModifyMenu API function to change several menu commands to be owner drawn. You'll see that the code first calls GetMenuItemID to get the original menu's ID. The ID is then reassigned in the call to ModifyMenu. By making sure the same menu IDs are maintained, Visual Basic's regular processing of menu commands will not be affected.

Because Windows doesn't know how much room you'll need to draw the item, it first sends a WM_MEASUREITEM message to get the item's size in pixels. The code presented here uses the GetSystemMetrics API function to determine the normal height of menu commands, and arbitrarily uses the

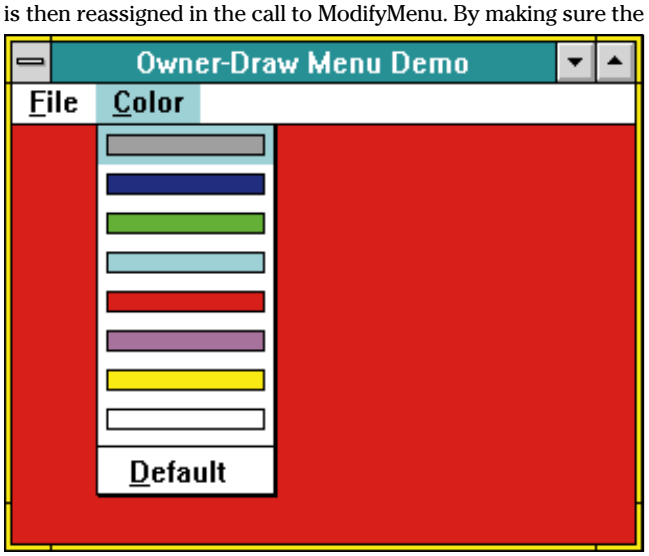


FIGURE 4 ***Owner-Draw Menus.*** *A subclassing control to intercept the WM_DRAWITEM message, sent when owner-draw menu items need painting, allows you make commands appear in the manner of your choice.*

```
Sub Form_Load ()
  '
  ' Install app in viewer chain
  '
  hWndNext = SetClipboardViewer(hWnd)
  '
  ' Setup MsgHook control
  '
  MsgHook.HwndHook = Me.hWnd
  MsgHook.Message(WM_CHANGECBCHAIN) = True
  MsgHook.Message(WM_DRAWCLIPBOARD) = True
  '
  ' Paint whatever's currently in the clipboard
  ' using a custom routine written for this demo.
  '
  UpdateClipView
End Sub

Sub MsgHook_Message (Msg As Integer, wParam As Integer, _
   lParam As Long, Result As Long)
  Dim nRet As Long
  '
  ' Take appropriate action based on incoming message.
  '
  Select Case Msg
    Case WM_CHANGECBCHAIN
      '
      ' If the window being removed is the
      ' next window in the chain, the window
      ' specified by the hwndNext parameter
      ' becomes the next window and clipboard
      ' messages are passed on to it.
      '
      If wParam = hWndNext Then
        hWndNext = WordLo(lParam)
      End If

    Case WM_DRAWCLIPBOARD
      '
      ' Contents of clipboard have changed.
      ' Call routine to read.
      '
      UpdateClipView

  End Select
  '
  ' Each window that receives either of these
  ' messages should call the SendMessage
  ' function to pass the message on to the
  ' next window in the clipboard-viewer chain.
  '
  nRet = SendMessage(hWndNext, _
    WM_CHANGECBCHAIN, wParam, lParam)
  Result = 0
End Sub

Sub Form_Unload (Cancel As Integer)
  Dim nRet As Integer
  '
  ' Remove Me from the clipboard viewer chain.
  ' DO NOT stop execution from VB menu or toolbar!
  '
  nRet = ChangeClipboardChain(Me.hWnd, hWndNext)
End Sub
```

LISTING 8 ***Hooking Into the Clipboard Viewer Chain.*** *Here, MsgHook is used to receive notification whenever the contents of the clipboard change. There are a number of responsibilities that go along with becoming a clipboard viewer app, including notifying the next app in the chain of all changes, and properly removing yourself upon termination.*

value of 70 for the width (see Figure 4).

8 HOOKING INTO THE CLIPBOARD VIEWER CHAIN

Changes to the Windows clipboard are passed along a chain, or list, of applications that install themselves as clipboard viewers. Simply calling the SetClipboardViewer API inserts your form at the head of the list, and returns the handle of the next window in the chain. It is your responsibility to notify this window whenever the clipboard contents change. Due to this arrangement, when your application is about to end you must remove yourself from the chain to prevent wild pointers to your form's old handle.

After becoming part of the chain, use MsgHook to receive the WM_DRAWCLIPBOARD message (see Listing 8), which is sent to the first window in the viewer chain whenever the clipboard contents change. Each application in the chain can use this message as a signal to retrieve the new contents of the clipboard.

Occasionally, another application will either install or remove itself from the viewer chain, and you'll receive the WM_CHANGECBCHAIN message. If this window is the one you were to notify of changes, the handle for the next one following it in the chain is in the low word of lParam parameter. When either of these messages is received, you need to pass them on along the chain after processing.

Normally, a clipboard viewer app would respond to the WM_DESTROY message by removing itself from the chain. However, due to the nature of Visual Basic subclassing controls, this message is not typically interceptable. Luckily, the Form_Unload event is available as an alternative. Due to this slight kludge, it's important that while testing your application, you do not stop execution from Visual Basic's toolbar. Instead, do it in a manner that ensures the Form_Unload event will occur. ■

User Tip

ROUND NUMBERS AUTOMATICALLY

This simple function automates the task of rounding numbers:

```
Function Round (aNumber As Variant, DecimalPlaces%) _
  As Variant
'Use DecimalPlaces=0 to round to an integer
Dim Temp As Double, DecShift As Long
  Temp = CDbl(aNumber)
  DecShxift = 10 ^ DecimalPlaces
  Round = (Fix((Temp + .5 / DecShift) * DecShift)) /
DecShift
End Function
```

—Gary Baren, received by CompuServe

SEND YOUR TIP

If it's cool and we publish it, we'll pay you $25. If it includes code, limit code length to 10 lines if possible. Be sure to include a clear explanation of what it does and why it is useful. Send to 74774.305@compuserve.com or Fawcette Technical Publications, 209 Hamilton Ave, Palo Alto, CA, USA, 94301-2500.