Get a Grip With Subclassing

Click & Retrieve Source CODE!

by Karl E. Peterson

Q Add a Size Grip

It appears using the status-bar control is the only supported method of adding a size grip—those diagonal lines in the lower-right corner of many dialogs. However, bundling the comctl32.ocx package into my distribution set would be major overkill, because this is all I'd use it for. How can I add a size grip with straight code?

A You're right. There's no straightforward way to add this standard user interface element to VB forms. That's a real shame, given it's much more difficult to visually distinguish a sizable form from a nonsizable form under the new shell introduced in Win95. But it's certainly nothing a little creative coding can't cure.

You can take two approaches. The first approach is quick, simple, and effectively fakes what you're after. Put two label controls in the lower-right corner of the form. These labels use the Marlett font to display the grab handle symbols in system highlight and shadow colors. On MouseDown, VB normally calls SetCapture to claim all mouse input from that point forward. When the user presses the mouse button over the labels, call ReleaseCapture to reverse VB's native desires. Then call SendMessage to your form's window handle, passing WM_NCLBUTTONDOWN as the message and HTBOTTOMRIGHT in wParam (see Listing 1). This effectively tells the window—your form—to enter sizing mode, and Windows takes over from there.

The second approach is a little dirtier, but gets to the heart of the matter, taking full control. For this solution, pull out your favorite subclassing control or module, and hook WM_PAINT, WM_SIZE, and WM_NCHITEST.

Painting a size grip is trivial. When your form receives a WM_PAINT message, invoke the default window procedure so the form paints as it normally would. Following the default painting, calculate the rectangle in which the size grip should be drawn by offsetting a rectangle from the bottom-right corner of the form's client space. First call GetClientRect to fill a structure with the coordinates of the entire client area. Then reduce the Left and Top elements by subtracting the results of calls to GetSystemMetrics with SM_CXSIZE (for width) and SM_CYSIZE (for height). To paint the size grip, call DrawFrameControl with the calculated rectangle and the appropriate constants (download Listing A from DevX; see the Download Free Code box for details).

Because you're painting elements on the form, you're also responsible for erasing them as needed. In response to the WM_SIZE message, check whether any part of the rectangle calculated in the previous step is still within the client area of the form. Usually it is, but it might not be if the form is resized rapidly. If the rectangle is still visible, call InvalidateRect to force that area to be repainted at the next opportunity. If the rectangle is completely hidden, cover all bases by posting a WM_PAINT message back to your form. At that point, allow default processing to continue.

Now you have the size grip visually in place, but it still doesn't have the desired effect of expanding the area where the user can grab that corner of your form. Each window is responsible for telling Windows what part of its nonclient space the cursor is over, and Windows reacts accordingly. On receipt of nonclient hit-test (WM_NCHITEST) messages, tell Windows the cursor is over the bottom-right corner of the window border when it's over the area where you painted the size grip. Establish the coordinates rectangle using the same method as for WM_PAINT. Call the PtInRect API to confirm whether the cursor is within this rectangle. Return HTBOTTOMRIGHT to Windows if the cursor is within the desired area, and invoke the default window procedure if it isn't.

Which of the two approaches to use is your call. I'd likely use the subclassing method myself, because it requires you to monitor only a few extra messages in addition to the ones you're already hooking. It also is unlikely to break in the future. The shortcut method could break if, for example, Microsoft decided to use a font other than Marlett for standard symbols.

ABOUT THIS COLUMN

Ask the VB Pro provides you with free advice on programming obstacles, techniques, and ideas. Read more answers from our crack VB pros on the Web at www.inquiry.com/thevbpro. You can submit your questions, tips, or ideas on the site, or access a comprehensive database of previously answered questions.

VB4/32, VB5, VB6 | **Add a Size Grip to Forms Without Subclassing**

```
Option Explicit

Private Declare Function ReleaseCapture Lib "user32" () _
  As Long
Private Declare Function SendMessage Lib "user32" Alias _
  "SendMessageA" (ByVal hwnd As Long, ByVal wMsg As _
  Long, ByVal wParam As Long, lParam As Any) As Long

Private Const WM_NCLBUTTONDOWN = &HA1
Private Const HTBOTTOMRIGHT = 17

Private Sub Form_Load()
  ' Define font for size grip labels. (Done here for
  ' demo, but probably easier at design time.)
  Dim fnt As New StdFont
  With fnt
    .Name = "Marlett"
    .Bold = False
    .Size = 12
  End With

  ' Assign all relevant properties to size grip labels.
  With lblGrip(0)
    Set Font = fnt
    .AutoSize = True
    .Caption = "o"
    .ForeColor = vb3DHighlight
    .MousePointer = vbSizeNWSE
    .ZOrder
  End With
  With lblGrip(1)
    Set Font = fnt
    .AutoSize = True
    .Caption = "p"
    .ForeColor = vb3DShadow
    .MousePointer = vbSizeNWSE
    .ZOrder
  End With
End Sub

Private Sub Form_Resize()
  ' Position size grip labels at lower-right.
  lblGrip(0).Move Me.ScaleWidth - lblGrip(0).Width, _
    Me.ScaleHeight - lblGrip(0).Height
  lblGrip(1).Move Me.ScaleWidth - lblGrip(1).Width, _
    Me.ScaleHeight - lblGrip(1).Height
End Sub

Private Sub lblGrip_MouseDown(Index As Integer, Button _
  As Integer, Shift As Integer, x As Single, y As _
  Single)
  ' Negate VB's call to SetCapture, and tell Windows
  ' that the user is trying to resize the form.
  ReleaseCapture
  SendMessage hwnd, WM_NCLBUTTONDOWN, HTBOTTOMRIGHT, _
    ByVal 0&
End Sub
```

Listing 1 The two labels display the two Marlett characters that together make up the common size grip "character." Both labels are required because of the dual tone. Initiate "sizing mode" on a window by calling SendMessage with WM_NCLBUTTONDOWN and HTBOTTOMRIGHT.

need a surefire way to prevent people from using my DLL in a distributed application unless they have registered it.

Subclassing also allows you to extend the feature to non-VB windows owned by your application, such as a browse or file dialog. In either case, if you don't like doing this much work for a solution that should be simple, you could write to vbwish@microsoft.com and let them know that SizeGrip should be a standard property on forms in VB7.

VB4/32, VB5, VB6 | Don't Paint Until You Need To

```
Option Explicit

Private Const WM_ENTERSIZEMOVE = &H231
Private Const WM_EXITSIZEMOVE = &H232

Private m_Sizing As Boolean

Private Sub Form_Load()
  ' Set up subclassing
  Msghook1.HwndHook = Me.hWnd
  Msghook1.Message(WM_ENTERSIZEMOVE) = True
  Msghook1.Message(WM_EXITSIZEMOVE) = True
End Sub

Private Sub Form_Paint()
  If Not m_Sizing Then
    ' Paint something complicated.
  End If
End Sub

Private Sub Msghook1_Message(ByVal msg As Long, ByVal _
  wp As Long, ByVal lp As Long, result As Long)
  ' Set flag appropriately, then call default
  ' window procedure.
  Select Case msg
    Case WM_ENTERSIZEMOVE
      m_Sizing = True
    Case WM_EXITSIZEMOVE
      m_Sizing = False
      Me.Refresh
  End Select
  result = Msghook1.InvokeWindowProc(msg, wp, lp)
End Sub
```

Listing 2 If your form requires extensive reaction when it's resized, you might want to defer the continuous firing of Paint and Resize events until the user finishes sizing the form. Toggle a flag in reaction to WM_ENTERSIZEMOVE and WM_EXITSIZEMOVE, and don't repaint or resize controls unless this flag is set to False.

Q Don't Paint Until Finished Resizing

Is there a way of telling when a user has finished dragging the form's edges to resize? I'd like to set a flag to tell my repaint routine not to draw if the user hasn't let go of the mouse button during a resize.

A Once again, subclassing comes to the rescue. Windows sends your form a WM_ENTERSIZEMOVE message when the user begins resizing your form, and follows with a WM_EXITSIZEMOVE message when the user finishes the resize operation. If you watch for these messages, and toggle your flag variable in reaction to them, you can defer your painting until the user stops dragging the form's border (see Listing 2). By the way, you can use the same strategy to defer resizing controls in your form's Resize event, too.

Q Produce a Demo DLL

I've written a shareware DLL I'd like to offer for folks to try before they buy. The problem is that I need a surefire way to prevent people from using my DLL in a distributed application unless they have in fact registered it. How can I restrict the usage of the free download to within the VB Integrated Development Environment (IDE)?

A The GetModuleHandle API provides a quick and dirty way to cover most bases in this case. This API accepts a file name and

VB4/32, VB5, VB6 | Restrict DLL Usage to the IDE

```
Private Declare Function GetModuleHandle Lib "kernel32" _
  Alias "GetModuleHandleA" (ByVal lpModuleName As _
  String) As Long

Private Function TestEnvs() As Boolean
  Dim buffer As String
  Dim Envs As Variant
  Dim nRet As Long
  Dim i As Long

  ' Fill array with the names of all environments in
  ' which you want users to be able to use your DLL.
  Envs = Array("vb.exe", "vb32.exe", "vb5.exe", _
    "vb6.exe")

  ' Test each array element with GetModuleHandle to see
  ' if that environment is mapped into the current
  ' process.
  For i = LBound(Envs) To UBound(Envs)
    buffer = Envs(i)
    nRet = GetModuleHandle(buffer)
    If nRet <> 0 Then
      m_EnvFileName = buffer
      TestEnvs = True
      Exit For
    End If
  Next i
End Function
```

Listing 3 Within your ActiveX DLL, you can use GetModuleHandle to determine whether specific files are mapped into the current process's address space. If any of these files are so mapped, you can reasonably assume your DLL is being called from within a development environment, not from a distributed EXE.

returns a handle to that module if the module is mapped into the address space of the current process. Your assignment is to come up with the file names of all the development environments from which you want to allow usage of your DLL. Once you gather these file names, place them in an array to facilitate iteration and make it easy to add or remove them as time goes on.

Pass each file name in turn to GetModuleHandle (see Listing 3). If GetModuleHandle returns a value other than 0, it means the file is presently mapped into the current address space. From this information, you can reasonably assume your user is operating in development mode, and not from a distributed EXE. Call the routine housing this logic from the Initialize event of each object exposed by your DLL, and alter that object's behavior based on the return.

Determined folks find a way to break any protection scheme. With this scheme, they could name their EXE after one of the targeted IDEs. Most aren't that foolish, and you can probably rest assured that users who are probably won't have a successful product anyway.

This question makes it three for three this month—three features that really should be built into VB but aren't. Have you written to vbwish@microsoft.com today? VBPJ

About the Author

Karl E. Peterson is a GIS analyst with a regional transportation planning agency and serves as a member of the **Visual Basic Programmer's Journal** Technical Review and Editorial Advisory Boards. Online, he's a Microsoft MVP and a section leader on several **VBPJ** forums. Find more of Karl's VB samples at www.mvps.org/vb.

DOWNLOAD FREE CODE

Download the code for this issue of **VBPJ free** from www.vbpj.com.

To get the free code for this entire issue, click on Locator+, the right-most option on the menu bar at the top of the VBPJ home page, and type **VBPJ0699** into the box. (You first need to register, for free, on DevX.) The free code for this article includes all code listings, plus the Grabber sample, which adds a size grip using MsgHook; a demo that defers Paint and Resize events; and a class that provides IDE mapping detection. You can download the MsgHook control from the author's Web site at www.mvps.org/vb.

✪ To get the bonus code for this article, available to DevX Premier Club members, type **VBPJ0699AP** into the Locator+ field. The bonus code includes all the free code described above, plus an enhanced Grabber sample that implements native subclassing to add a class-wrapped size grip and a ready-to-use size grip UserControl that employs the Marlett method.