by Carl Franklin

SUBCLASS TO INSERT ITEMS INTO THE SYSTEM MENU

This is your forum for addressing the intricacies of the Visual Basic language. Send in your questions, clever tips, and techniques. Visual Basic Programmer's Journal *will pay $25 for any submission or question we print. If your submission includes code, please send a disk along with your hard copy. Mail submissions to Q&A Columnists, c/o Fawcette Technical Publications, 209 Hamilton Avenue, Palo Alto, CA, USA, 94301-2500. CompuServe: 74774,305.*

INSERTING ITEMS INTO SYSTEM MENU

Q For months I have been trying to insert items into a VB app's system menu. Finally I figured out how to add an item, but I don't know how to define what clicking on the item will do. Can you please shed some light on this subject?
—Jack Einhorn, New York, New York

A Your program needs to respond to system menu commands. You can accomplish this using MSGBLAST.VBX, which you can find on CompuServe in the *VBPJ* Forum (GO VBPJFO), or on the MSDN CD from Microsoft. MSGBLAST lets you hook one or more messages. By hook, I mean that when your application receives a message that MSGBLAST is listening for, MSGBLAST's message event is fired, and all the parameters of the message are passed. This lets your VB apps respond to messages that it could not otherwise process.

In your case, you want to listen for WM_SYSCOMMAND messages. This message is sent any time the user selects an option in the system menu or selects the system menu itself. The wParam argument holds the Menu's identifier. When you added your menu item, you specified its identifier. This value is passed in wParam. Also, the standard system commands have constant identifiers you can check for, letting you do much more than just capture your own menu options (see Listing 1).

Because there are so many other uses for the MSGBLAST control, I've assembled two programs to give you some ideas for your own apps. The first program (see Listing 2) handles power management issues on laptop or notebook computers that use advanced power management (APM). A WM_POWER message is sent to your application before the machine powers down, and again when it resumes power, allowing you to close critical files, or do whatever else may be necessary when in an idle state. You would not want to suspend a critical operation such as power suspension by putting up a message box. The message boxes are for the sake of demonstration.

Carl Franklin is an independent software developer who spends his time writing code on the cutting edge of innovation, and ripping blues leads on his Gibson Les Paul. Contact Carl for Q&A questions and private consulting by e-mail at carlf@win2.com, by CompuServe at 74561,3324 (he frequently visits the new VBPJ *Forum on CompuServe—GO VBPJFO), or by phone at 909-799-0302. You can also write Carl at 24879 Lawton Ave., Loma Linda, CA 92354.*

The second program (see Listing 3) returns the names of files that are dropped on your form from the File Manager. In order for this to work, you must call the DragAcceptFiles routine to tell Windows that your particular window will accept files being dropped on it. When files are dropped, the WM_DROPFILE message is sent, and a handle to the drop is passed. By calling the DragQueryFile routine and passing the hDrop, you can determine the number of files and their names.

As an aside, MSGBLAST.VBX works by continuously checking the message queue in a loop with the PeekMessage API function. When it finds a message you've told it to hook, it fires the Message event and passes you all of the message's parameters. You could do the same thing in your application, but you'd have to be in a continuous loop, checking the message queue.

USING API CALLS TO PRINT

Q I have written several VB applications that display graphical and text information on a map background. I recently added a Hewlett-Packard DeskJet 560C printer to my system. I can print using VB printer control commands (Printer.Line), but when I use Windows API functions (LineTo%) to print, I get no printer response. I have used the same API code with three other printers, with no problem. I have traced the problem to the API function CreateDC%, which creates a device context for the printer. CreateDC returns a zero (error) for my HP 560C, but not for the other printers.
—Ken Cranford, Colorado Springs, Colorado

A It's either the driver or your code. From the sound of it, it's the driver. Either it's not set up correctly, or there is a problem with it. Before you call for tech support, get the printer's device context from the printer object's hDC property, which always returns a handle to the current printer. If Printer.hDC is returning zero, call tech support. The key in pinpointing problems with drivers or with any other element of a high-level system is finding out which subsystems are working. To do that, you usually have to test each piece of the puzzle until you come to the piece that's broken.

DETERMINING WHETHER SCROLLBARS ARE VISIBLE

Q While it's true that you can read the Grid1.Scrollbars property at run time, there is no way to determine if either scrollbar is actually visible. This makes placement of a text box over a cell near the grid edges a bit hard to calculate.
—Zsolt Halmos, Los Gatos, California

A VB does not provide a way to determine if a scrollbar is actually visible, but there is a workaround. When a scrollbar is visible, the size of the client area is diminished by the space taken up by the scrollbar. Compare the pixel-

CONTINUED ON PAGE 126.

```
DefInt A-Z

Const WM_SYSCOMMAND = &H112
Const MF_STRING = &H0
Const SC_TEST = 1
Const SC_CLOSE = &HF060
Const SC_MAXIMIZE = &HF030
Const SC_MINIMIZE = &HF020
Const SC_MOVE = &HF010
Const SC_RESTORE = &HF120
Const SC_SIZE = &HF000
Const SC_TASKLIST = &HF130

Declare Function GetSystemMenu% Lib "User" _
   (ByVal hWnd%, ByVal bRevert%)
Declare Function AppendMenu% Lib "User" _
   (ByVal hMenu%, ByVal wFlags%, ByVal wIDNewItem%, _
   ByVal lpNewItem As Any)

Sub Form_Load ()

   '-- Get a handle to the system menu
   hw = Me.hWnd
   hMenu = GetSystemMenu(hw, False)

   '-- Add a menu option with the caption, "Test"
   If AppendMenu(hMenu, MF_STRING, _
      SC_TEST, "&Test") Then
      '-- This means success.

      '-- Tell MSGBLAST to hook this form and process
      ' WM_SYSCOMMAND Messages.
      MsgBlaster1.hWndTarget = hw
      MsgBlaster1.MsgList(0) = WM_SYSCOMMAND
   End If

End Sub
Sub MsgBlaster1_Message (MsgVal As Integer, _
   wParam As Integer, lParam As Long, _
   ReturnVal As Long)

   Select Case MsgVal
      Case WM_SYSCOMMAND
           '-- An item was selected from the system menu
           Select Case wParam
                '-- Create a message for each menu item
                Case SC_TEST
                     Msg$ = "You selected Test"
                Case SC_CLOSE
                     Msg$ = "You selected Close"
                Case SC_MAXIMIZE
                     Msg$ = "You selected Maximize"
                Case SC_MINIMIZE
                     Msg$ = "You selected Minimize"
                Case SC_MOVE
                     Msg$ = "You selected Move"
                Case SC_SIZE
                     Msg$ = "You selected Size"
                Case SC_TASKLIST
                     Msg$ = "You selected Switch To"
                Case SC_RESTORE
                     Msg$ = "You selected Restore"
           End Select

           If Len(Msg$) Then
                '-- Display the message
                MsgBox Msg$
           End If
   End Select

End Sub
```

LISTING 1 ***System Menu Hook.*** *This code adds an item to your app's system menu and uses Message Blaster to process the menu selection. Note that each item in the system menu shares the same message (WM_SYSCOMMAND) and the menu option is specified in the two-byte wParam parameter of the message. To set up this application, place a MSGBLAST.VBX control on a form.*

```
DefInt A-Z

Const WM_POWER = &H48

Const PWR_CRITICALRESUME = 3
Const PWR_SUSPENDREQUEST = 1
Const PWR_SUSPENDRESUME = 2

Const PWR_FAIL = -1
Const PWR_OK = 1

Sub Form_Load ()

   '-- Get a handle to the system menu
   hw = Me.hWnd

   '-- Tell MSGBLAST to hook this form and process
   ' WM_POWER Messages.
   MsgBlaster1.hWndTarget = hw
   MsgBlaster1.MsgList(0) = WM_POWER

End Sub

Sub MsgBlaster1_Message (MsgVal As Integer, _
   wParam As Integer, lParam As Long, _
   ReturnVal As Long)

   Select Case MsgVal
      Case WM_POWER
           '-- We're getting a power management message
           Select Case wParam
                Case PWR_SUSPENDREQUEST
                     '-- System is suspending power
                     Beep
                     '-- Ask the user if he/she wants to
                     '-- close files
                     OK = MsgBox("System Powering Down. _
                          Close All Open Files?", 36)
                     If OK Then
                          '-- Return this value if
                          '-- everything is cool.
                          ReturnVal = PWR_OK
                     Else
                          '-- Return PWR_FAIL if you could
                          '-- not close files.
                          '   This will abort the suspend.
                          ReturnVal = PWR_FAIL
                     End If
                Case PWR_SUSPENDRESUME
                     '-- System is about to power up
                     MsgBox "System Coming Online. _
                          Re-Open Files here"
                Case PWR_CRITICALRESUME
                     '-- System is about to power up, but we
                     '-- didn't recieve a PWR_SUSPENDREQUEST
                     '-- message first.
                     MsgBox "System Coming Online after _
                          unexpected powerdown. Re-Open _
                          Files here"
           End Select
   End Select

End Sub
```

LISTING 2 ***Power Management.*** *Windows sends messages before and after suspending power on laptops that use advanced power management. You can hook these messages so you can safely close any open files in case the machine is left turned off. When the machine's power is restored, another message notifies your program. To set up this application, place a MSGBLAST.VBX control on a form.*

CONTINUED FROM PAGE 122.

based width of the control with the width of its client area (which you can get by a call to GetClientRect), and if they are not equal then the vertical scrollbar is visible.

Do the same calculation for the horizontal scrollbar, comparing the pixel-based height with the client area's height. I have some functions (see Listing 4), VScrollVisible and HScrollVisible, that use this method to determine whether or not the scrollbars are visible for any given control.

```
DefInt A-Z

Const WM_DROPFILES = &H233

Declare Sub DragAcceptFiles Lib "SHELL.DLL" _
   (ByVal hWnd, ByVal Accept)
Declare Function DragQueryFile Lib "SHELL.DLL" _
   (ByVal hDrop, ByVal Index, ByVal FileName$, _
   ByVal Size)
Declare Sub DragFinish Lib "SHELL.DLL" _
   (ByVal hDrop)

Sub Form_Load ()

   '-- Get a handle to the system menu
   hw = Me.hWnd

   '-- Tell windows that our form can be a dropping
   ' place for files.
   Call DragAcceptFiles(hw, True)

   '-- Tell MSGBLAST to hook this form and process
   ' WM_DROPFILES Messages.
   MsgBlaster1.hWndTarget = hw
   MsgBlaster1.MsgList(0) = WM_DROPFILES

End Sub

Sub MsgBlaster1_Message (MsgVal As Integer, _
   wParam As Integer, lParam As Long, ReturnVal _
      As Long)

   Select Case MsgVal
      Case WM_DROPFILES
         '-- wParam is a handle to the drop
         hDrop = wParam

         '-- Get the number of files by calling
         '-- DragQueryFile
         NumFiles = DragQueryFile(hDrop, -1, _
               FileName$, 127)

         If NumFiles Then
               '-- Clear the form
               Me.Cls
               '-- Get each filename from _
      DragQueryFile
               For i = 0 To NumFiles - 1
                     FileName$ = Space$(127)
                     L = DragQueryFile(hDrop, i, _
                           FileName$, 127)
                     FileName$ = Left$(FileName$, L)
                     '-- Print the file name on the form
                     ' for demonstration purposes.
                     Me.Print FileName$
               Next
         End If

         Call DragFinish(hDrop)
   End Select

End Sub
```

LISTING 3 ***Drag and Drop File Names from File Manager.*** *This code returns the names of files as they are dropped on the form from the File Manager. When you drop one or more files on the form, the WM_DROPFILES message is received. From the information passed you can determine the number of files that were dropped and their names.*

The only stipulation is that the control passed must have an hWnd property. The RECT structure (or Type) holds a pixel-based map of any rectangle. The four elements of RECT (Top, Bottom, Left, Right) hold pixel positions of the bounds of the rectangle. The GetClientRect routine returns the bounds of a Window's client area, or the area available to the user as a working area.

The client area does not include scrollbars or title bars. It is for this reason that comparing the size of a control's client area to its overall size works for determining if scrollbars are visible. ■

```
DefInt A-Z

Type Rect
   Left As Integer
   Top As Integer
   Right As Integer
   Bottom As Integer
End Type

Declare Sub GetClientRect Lib "User" _
   (ByVal hWnd As Integer, lpRect As Rect)

Function VScrollVisible (Ctrl As Control) _
   As Integer

   '-- Create a RECT structure
   Dim Area As Rect

   '-- Get the control's client area coordinates
   ' (pixel based)
   hw = Ctrl.hWnd
   Call GetClientRect(hw, Area)

   '-- Determine the width as the Right
   ' minus the Left
   ClientWidth = (Area.Right - Area.Left) + 2

   '-- Get the actual width in pixels
   WindowWidth = Ctrl.Width \ Screen.TwipsPerPixelX

   '-- If they are not the same then the
   ' vertical scrollbar is visible.
   If ClientWidth <> WindowWidth Then
      VScrollVisible = True
   End If

End Function

Function HScrollVisible (Ctrl As Control) _
   As Integer

   '-- Create a RECT structure
   Dim Area As Rect

   '-- Get the control's client area coordinates
   ' (pixel based)
   hw = Ctrl.hWnd
   Call GetClientRect(hw, Area)

   '-- Determine the height as being the
   ' bottom minus the top
   ClientHeight = (Area.Bottom - Area.Top) + 2

   '-- Get the actual height in pixels
   WindowHeight = Ctrl.Height \ Screen.TwipsPerPixelY

   '-- If they are not the same then the
   ' horizontal scrollbar is visible.
   If ClientHeight <> WindowHeight Then
      HScrollVisible = True
   End If

End Function
```

LISTING 4 ***VScrollVisible and HScrollVisible.*** *These routines use the GetClientRect API routine to determine if a given control's scrollbar is visible.*