# Stay in the Tray

Click & Retrieve Source CODE!

## *Add your application to the taskbar tray.*

by Karl E. Peterson

Visual Basic programmers can more freely develop applications that embed themselves in the taskbar notification area (commonly called "the tray") now that Windows NT 4.0 ships with the newshell interface. Such applications previously were restricted to running under Windows 95, and they required significant architectural changes if the author wanted them to run in both 32-bit environments. This month, I will present the basic functions required for a tray application.

Tray applications have two basic requirements. Foremost, of course, they must actually reside in the tray—modifying your application to make this happen requires only one API call. Most tray apps also require the ability to receive mouse messages from the system as the user clicks on the icon. Beyond this, tray apps vary little from any other type of application. They generally feature a form that displays itself when the user double-clicks on the icon, and they often provide a popup context menu when the user right-clicks on the icon. Tray apps don't need such features though; you can also design an application to just load at startup and remain running, undisturbed, throughout the Windows session.

It's wise to test for the operating system version at startup because your applications will undoubtedly still have to face workstations running NT 3.51 for a while. To do so, you can call the GetVersionEx API function and test the value of dwMajorVersion:

```
Dim os As OSVERSIONINFO
os.dwOSVersionInfoSize = Len(os)
Call GetVersionEx(os)
If os.dwMajorVersion < 4 Then
   MsgBox "NT4 or Win95 Required!", _
      vbCritical, "Program Ending"
   End
End If
```

See my last Programming Techniques column, "Unicode Strings You Along" [*VBPJ* September 1996], for a complete description of this API call, and a class module that provides easy access to all the information GetVersionEx provides.

This check is typically performed in Sub Main, so you can bail out before any forms load if you find yourself in an older operating system. If a suitable OS is found, you can then load your principal form after performing any other required initialization in Sub Main. This form requires three module-level declarations—two constants that will be used to identify your application and send messages to it, and the structure that provides the OS with required data about your tray app (see Listing 1 for the structure and API declarations):

```
Private Const cbNotify& = &H4000
Private Const uID& = 61860
Private nid As NOTIFYICONDATA
```

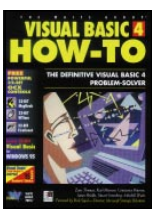
The constant cbNotify is a user-defined message number that the OS will send to your app to notify it of mouse events. You can set this constant to any value you choose, but as a good rule of thumb, ensure that it's higher than WM_USER so that it doesn't conflict with other system messages. Likewise, the uID is also a user-defined constant that you can set to any value. In cases where your application loads more than one icon into the tray, uID identifies which icon system messages are intended for. By declaring a module-level instance of the NOTIFYICONDATA structure, you can use it in multiple situations—at the least for adding the icon at initialization and removing it at termination.

Insert your application into the tray during the Form_Load event. To do so, you must fill out all members of the NOTIFYICONDATA structure and call the ShellNotifyIcon API. The cbSize element designates the structure's size and is required (as is typical with many API structures). The uFlags element instructs the OS which elements of this structure should be used and which to ignore. Because this is the initial call to ShellNotifyIcon, all available flags should be used. The hIcon element provides a handle to the icon used in the tray, and the szTip element contains a pointer to the tooltip text displayed when the user moves the mouse over your icon:

```
nid.cbSize = Len(nid)
nid.hWnd = Msghook.HwndHook
nid.uID = uID
nid.uFlags = NIF_MESSAGE Or NIF_TIP Or NIF_ICON
nid.uCallbackMessage = cbNotify
nid.hIcon = Me.Icon
nid.szTip = Me.Caption & Chr(0)
Call ShellNotifyIcon(NIM_ADD, nid)
```

The hWnd and uCallbackMessage elements instruct the OS which window to notify with system messages. If your application doesn't allow user interaction, you can safely ignore this step. But, if you want to be notified when the user clicks on your icon, you must use a subclassing control to intercept these messages. I've used MsgHook to intercept messages sent to the main form's window, specifically filtering for the cbNotify message. This hook,

*Karl E. Peterson is a GIS analyst with a regional transportation planning agency and a member of the* VBPJ *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 in the MSBASIC Forum. Contact Karl in either of the CompuServe forums at 72302,3707 or on the Internet at karl@rtc.wa.gov.*



in effect, sets up a message-based callback to your form. All you need to do during Form_Load is ensure that your form has its Visible property set to False and instruct your subclassing control to intercept the proper message(s) before calling ShellNotifyIcon. With MsgHook, use these instructions:

```
Msghook.HwndHook = Me.hwnd
Msghook.Message(cbNotify) = True
```

As the user interacts with your icon, the system sends messages notifying you of what the user is doing. The message number is always cbNotify, and the wParam is uID; both are the values you specified in the NOTIFYICONDATA structure. The lParam value can be any of the various mouse messages. Use a Select Case block within your subclassing control's message notification event to determine how to react to different user actions (see Listing 2). Commonly, you want to show your main form when the user double-clicks on the icon, and provide a popup context menu when the user right-clicks on the icon.

You must observe just a few remaining details in order to assure a well-behaved app. For a nice touch, use the QueryUnload event to cancel an unload if the user presses the title bar's Close button on your form. Consider hiding your form instead, and offer a menu option to actually unload the application:

```
If UnloadMode = vbFormControlMenu Then
   ' Just hide form if user presses Close button
   Me.Visible = False
   Cancel = True
End If
```

Finally, you must remember to remove your icon from the tray while your application shuts down. This step requires one more call to ShellNotifyIcon in your Form_Unload event. Pass the same structure you used to set up the tray icon, but use the NIM_DELETE message to instruct the system to remove your icon:

```
Call ShellNotifyIcon(NIM_DELETE, nid)
```

Your application may want to modify the icon used in the tray or its tooltip text while it's running. To do so only requires additional calls to ShellNotifyIcon using the NIM_MODIFY message. The application framework available to members of the Premier Club of The Development Exchange includes the few extra steps required to provide this functionality. If I may offer one final word of advice, please carefully consider the appropriateness of placing your application in the taskbar tray. Users probably will be frustrated by unnecessary consumption of this precious real estate. For a complete application framework that will give you a great starting point in developing your own tray application, check out the files available to members of the DevX Premier Club (for details, see the Code Online box at the end of this column).

## RESOURCE CONSERVATION TIP

Often, you want extra icons for your app. Perhaps you want to "animate" your app's icon at run time or store a number of special mouse pointers. One resource-intensive way to do this would be to put multiple hidden picture boxes on your form and store an icon in each. Once an icon is in a picture box though, it essentially becomes a bitmap. The simple method I use consumes nearly the fewest possible resources while letting me maintain the unique characteristics (transparent and inverse pixels) of an icon: the intrinsic Label control offers the perfect container! I just place an invisible Label on my form and set the desired icon into its DragIcon property. I can then assign this property to anything that requires an icon handle. While using an RES file to hold the icon(s) would be slightly more resource-efficient, it would also be nowhere near as

VB4

```
Public Type NOTIFYICONDATA
  cbSize As Long
  hwnd As Long
  uID As Long
  uFlags As Long
  uCallbackMessage As Long
  hIcon As Long
  szTip As String * 64
End Type
Public Const NIM_ADD = &H0
Public Const NIM_MODIFY = &H1
Public Const NIM_DELETE = &H2
Public Const NIF_MESSAGE = &H1
Public Const NIF_ICON = &H2
Public Const NIF_TIP = &H4
Public Declare Function ShellNotifyIcon Lib _
  "shell32.dll" Alias "Shell_NotifyIconA" _
  (ByVal dwMessage As Long, _
  lpData As NOTIFYICONDATA) As Long
Public Const WM_MOUSEMOVE = &H200
Public Const WM_LBUTTONDOWN = &H201
Public Const WM_LBUTTONUP = &H202
Public Const WM_LBUTTONDBLCLK = &H203
Public Const WM_RBUTTONDOWN = &H204
Public Const WM_RBUTTONUP = &H205
Public Const WM_RBUTTONDBLCLK = &H206
Public Const WM_MBUTTONDOWN = &H207
Public Const WM_MBUTTONUP = &H208
Public Const WM_MBUTTONDBLCLK = &H209
```

**LISTING 1** ***Declarations Required for a Tray App.*** *Insert these API, constant, and structure declarations into your main startup module. You may not need to include all the mouse messages, but having them there makes it easy to modify the behavior of your app with the least hassle.*

VB4

```
Private Sub Msghook_Message(ByVal msg As Long, _
  ByVal wp As Long, ByVal lp As Long, result As Long)
  Dim param As String
  param = "msg: " & msg & "    wp: " _
     & wp & "    lp: " & lp
  If wp = uID Then
     Select Case lp
        Case WM_MOUSEMOVE
        Case WM_LBUTTONDOWN
        Case WM_LBUTTONUP
        Case WM_LBUTTONDBLCLK
           ' Show form
           Me.Visible = True
           AppActivate Me.Caption
        Case WM_RBUTTONDOWN
        Case WM_RBUTTONUP
           ' Display context menu
           ' Highlight default (Open)
           Me.PopupMenu mPopup, , , , mPop(0)
        Case WM_RBUTTONDBLCLK
        Case WM_MBUTTONDOWN
        Case WM_MBUTTONUP
        Case WM_MBUTTONDBLCLK
        Case Else
           Debug.Print "Unknown!" & param
     End Select
  End If
End Sub
```

**LISTING 2** ***Notification Message Template.*** *Here are some of the mouse messages a tray app can receive from the system when the user clicks on the icon. The two most common messages you need to watch for are WM_LBUTTONDBLCLK and WM_RBUTTONUP.*

easy to implement. This trick works in any version of Visual Basic.

## FIND THE TEMP DIRECTORY

Win32 provides a GetTempPath API that returns the path to where the system stores temporary files. Win16 doesn't have a direct counterpart, though it comes close. In Win16, the GetTempFileName API returns the temp directory as the path portion of the function return value, *provided* you supply the correct disk for the temp file. In order to do that, you must first call the Win16 GetTempDrive API. This call is a remnant from Windows 1.0, when it was not uncommon for users to have Windows on Drive A: and data/swap space on Drive B:.

The good news is that this mess has finally been cleaned up in Win32. (See Knowledge Base article Q137034 for related information.) The better news is that Win32 uses an extremely simple algorithm, and programmers can easily implement it in VB. Windows NT and 95 simply search for the environment variable "TMP." If that is not found, then "TEMP" is checked. If neither environment variable is present, the current directory is returned. You can envision the VB function now, right? For me, it's far easier to type this routine than to dig out the appropriate API declaration, set it up, and call it:

```
Public Function GetTempPath() As String
   Dim TempPath As String
   ' Follow same rules as API.
   TempPath = Environ("TMP")
   If Len(TempPath) = 0 Then
      TempPath = Environ("TEMP")
      If Len(TempPath) = 0 Then
         TempPath = CurDir
      End If
   End If
   GetTempPath = TempPath
End Function
```

Having the path to the temporary directory allows you to create your application's temp files where the user wishes. The simplest method to do so is to call the GetTempFileName API, passing the path to the temp directory as the desired location for the new file. ☒

## Code Online

***You can find all the code published in this issue of* VBPJ *on The Development Exchange (DevX) at http://www.windx.com. All the listings and associated files essential to the articles are available for free to Registered members of DevX, in one ZIP file. This ZIP file is also posted in the Magazine Library of the* VBPJ *Forum on CompuServe. DevX Premier Club members ($20 for six months) can get each article's listings in a separate file, as well as additional code and utilities for selected articles, plus archives of all code ever published in* VBPJ *and* Microsoft Interactive Developer *magazines.***

### *Stay in the Tray*

**Locator+ Codes**

***Listings ZIP file (free Registered Level): VBPJ1196***

✪ ***Listings for this article plus a taskbar tray application framework, a freeware version of MsgHook, a hyperlink for a taskbar tray implementation using Call32.DLL and VB3, and various other links (subscriber Premier Level): PT1196P***