Is SetForegroundWindow Broken?

Sometimes my application needs to alert the user and request intervention due to various critical errors. I’ve used the SetForegroundWindow API without fail to bring my app to the front, with one exception: In the Win98 environment, when I call SetForegroundWindow, I get only an irritating flash of the main window’s title bar and the application’s button in the taskbar. What can I do to restore normal behavior?

OK, first let’s get the formal admonishment out of the way. Bringing an app to the foreground unbidden is bad behavior in most apps. As a developer, make every effort to avoid the need for this behavior altogether. Few users appreciate one app plopping up to the foreground while they’re working in another. Given all that, no matter what anyone says, you have to do it at times.

Here’s why your strategy no longer works, and how you can patch it up. Microsoft has altered the behavior of SetForegroundWindow under both Windows 98 and Windows 2000 (“The OS Formerly Known as NT5”). If your application is currently in the foreground, this API still behaves as it always did. Then you can bring any window to the front—both those belonging to your app and those of other apps. However, if your app doesn’t hold the foreground, newer versions of Windows simply call FlashWindow to achieve the flashing effect you describe.

The supreme irony here is that Microsoft apps have always been among the worst offenders in their use of SetForegroundWindow. (Ever used Outlook Express?) You almost get the feeling the systems programmers were fed up with the applications developers’ antics and decided to slap their wrists with this change. The shame of it: Their powerful reach extends to us all.

Enough background; time to move on to a potential solution. I’ve written a routine called ForceForegroundWindow that works in nearly every case (see Listing 1). Call it by passing the hWnd property of the form you want to bring to the top.

Creating a Task List

I’d like to offer my users the ability to switch to any other running application. To do this, I need to build a list of all running apps, similar to that offered on the first tab of Task Manager under NT. What’s the best way to get such a list?

To provide an accurate list of all running tasks, do a brute-force cycle through all the top-level windows, checking whether each window meets a list of criteria. The easiest way to do this: Call the EnumWindows API, pointing at your filtering routine. I’ve written the FillTaskListBox routine to accept a ListBox control as its only parameter. It clears the list and calls EnumWindows, passing the AddressOf EnumWindowsProc and the handle to...
the listbox (see Listing 2).

The system calls EnumWindowsProc for each top-level window. The procedure checks each window to ensure it's both visible and has neither parent nor owner. If the window meets these criteria, GetWindowText obtains the window caption and adds it to the list with a quick SendMessage call. Finally, EnumWindowsProc adds the window's handle as ItemData for the NewItem with one last SendMessage call. The enumeration continues as long as EnumWindowsProc returns True, or until it encounters all windows.

If you'd like to list all running processes, rather than top-level applications, see the Microsoft Knowledge Base article Q192986 for a complete example. Also, article Q183009 provides still more examples of several window enumeration APIs.

Use Recursive Callbacks

Q I want to write a routine to clear the Immediate window from code, but I'm stumped when it comes to finding its window handle. I thought it would be a simple matter of calling EnumChildWindows until I'd drilled down deep enough. But both VB5 and VB6 toss me a "Compile error: Expected Sub, Function, or Property" message box, with the EnumChildWindows call's AddressOf parameter highlighted. What's going on?

A It sounds as if you're trying to pass the address of the currently executing procedure. If that's the case, then yes, you've uncovered an oddity in the way VB processes the AddressOf directive. To avoid this error, you must fully qualify the procedure name by prepending the module name.

Listing 1 Under newer Windows versions, Microsoft has disabled the SetForegroundWindow API call in all cases except when the calling application currently maintains the foreground. This routine forces the issue by attaching itself to the foreground thread, faking out the operating system. Slimy? You bet! But not as slimy as the reason for writing it in the first place.

Option Explicit

' Required Win32 API Declarations

Private Declare Function GetWindowThreadProcessId Lib "user32" (ByVal hWnd As Long, lpdwProcessId As Long) As Long
Private Declare Function AttachThreadInput Lib "user32" (ByVal idAttach As Long, ByVal idAttachTo As Long, ByVal fAttach As Long) As Long
Private Declare Function GetForegroundWindow Lib "user32" () As Long
Private Declare Function SetForegroundWindow Lib "user32" (ByVal hWnd As Long) As Long
Private Declare Function IsIconic Lib "user32" (ByVal hWnd As Long) As Long
Private Declare Function ShowWindow Lib "user32" (ByVal hWnd As Long, ByVal nCmdShow As Long) As Long

' Constants used with APIs

Private Const SW_SHOW = 5
Private Const SW_RESTORE = 9

Public Function ForceForegroundWindow(ByVal hWnd As Long) As Boolean
    Dim ThreadID1 As Long
    Dim ThreadID2 As Long
    Dim nRet As Long

    ' First need to get the thread responsible for
    ' the foreground window, then the thread running
    ' the passed window.
    ThreadID1 = GetWindowThreadProcessId(GetForegroundWindow, ByVal 0&)
    ThreadID2 = GetWindowThreadProcessId(hWnd, ByVal 0&)

    ' By sharing input state, threads share their
    ' concept of the active window.
    If ThreadID1 <> ThreadID2 Then
        Call AttachThreadInput(ThreadID1, ThreadID2, True)
        nRet = SetForegroundWindow(hWnd)
        Call AttachThreadInput(ThreadID1, ThreadID2, False)
    Else
        nRet = SetForegroundWindow(hWnd)
    End If

    ' Restore and repaint
    If IsIconic(hWnd) Then
        Call ShowWindow(hWnd, SW_RESTORE)
    Else
        Call ShowWindow(hWnd, SW_SHOW)
    End If

    ' SetForegroundWindow return accurately reflects
    ' success.
    ForceForegroundWindow = CBool(nRet)
End Function
I've written a few routines that successfully obtain the Immediate window's handle, using a recursive enumeration that drills down through all the child windows of the current IDE instance (see Listing 3).

Now for the bad news: Even after you have this handle, there's still no way to clear the window. It doesn't react to any standard messages. If, like me, you'd like to see a Debug.Clear method added to VB, write vbwish@microsoft.com and tell them I sent you.

Q Start at a Higher Level

I'd like to open up common file dialogs at the Network Neighborhood level, rather than lower down, on a local disk or path. Is this possible?

A Sure is. The common dialogs use Explorer windows, so they're "receptive" to some of the same parameters. You can find the command-line switches for Explorer in MSDN or on the Web at http://premium.microsoft.com/msdn/library/winresource/dnwin95/d1c/s732e.htm. Here's the basic syntax:

```
explorer [/n] [/e] [/root.object][/select].subobject
```

The main clue here is the object reference for the root parameter. Here you can substitute a class ID (CLSID) for a folder name. A little registry spelunking...

---

**Listing 2** Windows' Task Manager roughly follows the criteria shown here when offering a list of all running tasks. The FillTaskListBox routine starts a top-level window enumeration, passing the handle of a listbox, which gets an addition each time an enumerated window passes through all the filters.

```vbnet
Option Explicit

' Required Win32 API Declarations

Private Declare Function EnumWindows Lib "user32" (ByVal lpEnumFunc As Long, ByVal lParam As Long) As Long
Private Declare Function IsWindowVisible Lib "user32" (ByVal hWnd As Long) As Long
Private Declare Function GetParent Lib "user32" (ByVal hWnd As Long) As Long
Private Declare Function GetWindowLong Lib "user32" Alias "GetWindowLongA" (ByVal hWnd As Long, ByVal nIndex As Long) As Long
Private Declare Function GetWindowText Lib "user32" Alias "GetWindowTextA" (ByVal hWnd As Long, ByVal lpString As String, ByVal cch As Long) 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

' Constant used to determine window owner.
Private Const GWL_HWNDPARENT = (-8)

' Listbox messages
Private Const LB_ADDSTRING = &H180
Private Const LB_SETITEMDATA = &H19A

Public Function FillTaskListBox(lst As ListBox) As Long
    ' Clear list, then refill it. Return final count.
    lst.Clear
    Call EnumWindows(AddressOf EnumWindowsProc, lst.hWnd)
    FillTaskListBox = lst.ListCount
End Function

Private Function EnumWindowsProc(ByVal hWnd As Long, ByVal lParam As Long) As Long
    Static WindowText As String
    Static nRet As Long
    ' Make sure we meet visibility requirements.
    If IsWindowVisible(hWnd) Then
        ' It shouldn't have any parent window, either.
        If GetParent(hWnd) = 0 Then
            ' And, finally, it shouldn't have an owner.
            If GetWindowLong(hWnd, GWL_HWNDPARENT) = 0 Then
                ' Retrieve window text (caption)
                WindowText = Space$(256)
                nRet = GetWindowText(hWnd, WindowText, Len(WindowText))
                ' Clean up window text and add to list.
                WindowText = Left$(WindowText, nRet)
                nRet = SendMessage(lParam, LB_ADDSTRING, 0, ByVal WindowText)
                Call SendMessage(lParam, LB_SETITEMDATA, nRet, ByVal hWnd)
                ' Return True to continue enumeration.
                EnumWindowsProc = True
            End If
        End If
    End If
End Function

Public Function FillTaskListProc(hWnd As Long) As Long
    ' Simple one-liner, since this is all that's being passed.
    nRet = hWnd
End Function
```

---
turns up the required CLSIDs for Network Neighborhood, as well as a few other interesting ones (see Table 1). To use these CLSIDs with the common dialogs, prepend two colons and pass the CLSIDs to the InitDir property—
or assign them to the lpstrInitialDir element of the OPENFILENAME structure if you’re calling the OpenFile API directly:

```vbnet
table 1 fun with clsids.

<table>
<thead>
<tr>
<th>Name</th>
<th>CLSID</th>
<th>Works With Dialogs</th>
</tr>
</thead>
<tbody>
<tr>
<td>My Computer</td>
<td>(20D04FE0-3AEA-1069-A2D8-08002B30309D)</td>
<td>Yes</td>
</tr>
<tr>
<td>Network Neighborhood</td>
<td>(20B2C60-3AEA-1069-A2D7-08002B30309D)</td>
<td>Yes</td>
</tr>
<tr>
<td>Recycle Bin</td>
<td>(645FF040-5081-101B-9F08-00AA002F954E)</td>
<td>Yes</td>
</tr>
<tr>
<td>Desktop</td>
<td>(00021400-0000-0000-C000-000000000000)</td>
<td>No</td>
</tr>
<tr>
<td>My Briefcase</td>
<td>(85BBD920-42A0-1069-A2E4-08002B30309D)</td>
<td>No</td>
</tr>
<tr>
<td>Control Panel</td>
<td>(21EC2020-3AEA-1069-A2DD-08002B30309D)</td>
<td>No</td>
</tr>
<tr>
<td>Printers</td>
<td>(2227A280-3AEA-1069-A2DE-08002B30309D)</td>
<td>No</td>
</tr>
<tr>
<td>Dial-Up Networking</td>
<td>(992CFA0-F557-101A-88EC-00DD010000C48)</td>
<td>No</td>
</tr>
</tbody>
</table>
```

To get the free code for this issue, click on Locator+, the right-most option on the menu bar at the top of the VBPJ home page, and type VBPJ0299 into the box. (You first need to register, for free, on DevX.) The free code for this article includes all code listings, plus demo applets that exercise each listing.

To get the code for this article only, available to DevX Premier Club members, type VBPJ0299AP into the Locator+ field.

**Listing 3** These routines find and return the handle to VB’s Immediate (Debug) window by recursively calling EnumChildWindows on each child of the top-level window. At each level, FindWindowEx looks for children that meet the desired criteria.