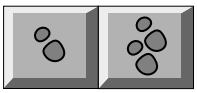

by Karl E. Peterson

PLAYING THE SHELL GAME

THE MICROSOFT KNOWLEDGE BASE PROVIDES HIDDEN RESOURCES ON TECHNICAL SUPPORT.

Lying somewhat buried within the Visual Basic Help file, unknown to many, is a page that contains a number of Microsoft Knowledge Base (KB) articles. To find it, open the Help file to the Contents page, select Technical Support, then select Knowledge Base Articles on VB. If you've never found this resource before, you will be rewarded with articles on a dozen of the most frequently asked questions regarding Visual Basic. This column will look at a couple of these questions, expand upon them, and see where that takes us. If you're intrigued by what you find in the Help file, the complete Knowledge Base consists of thousands of articles, and can be found on CompuServe (GO MSKB) or on the Microsoft Developer Network CDs.

Perhaps the question asked most often is, "How do I determine when a shelled process has terminated?" The solution is simple. Although poorly documented, the Shell function returns an instance handle to the process that the Shell function just started. The Windows API function GetModuleUsage, given an instance handle, returns the "reference count" for that module. To determine when the process has completed, enter an idle loop and check this value repeatedly until it's decremented to zero. To use this technique, place this declaration within your project:

```
Declare Function GetModuleUsage Lib "Kernel" (ByVal _
   hModule As Integer) As Integer
```

And, perform your Shell like this:

```
Dim hInst As Integer
hInst = Shell("Notepad.Exe", 1)
Do
   DoEvents
Loop While GetModuleUsage(hInst)
```

GetModuleUsage will continue to return a positive result as long as this instance of Notepad is running. Calling DoEvents within the loop reduces required processor time to a minimum, because control is immediately returned to Windows as soon as it's determined (using the results of the GetModuleUsage call) that Notepad is still running.

BUILD A BETTER SHELL

While Visual Basic's Shell function is versatile, it could be better. For example, now that you know when your shelled application has finished, it may not be necessary for it to remain visible as it executes. But among the WindowStyle (second parameter) values supported by Shell, the function lacks an option to invisibly start the process. Shell turns out to be a wrapper function for the WinExec API call—notice the correspondence between the ShowWindow constants (found in any API reference) and those used with Shell. But, one constant not supported by Shell is present, SW_HIDE, which has a value of zero.

The ShellAndWait function requires both a command line to execute and any one of the ShowWindow parameters as defined by the API (see Listing 1). To include this function in your project, place the necessary declarations and constant definitions in one of your modules. Then add the ShellAndWait function to your project. Call it just as you would Shell, using any of the ShowWindow constants as the desired WindowStyle. If you pass SW_HIDE as the second parameter, the spawned process will be invisible. Before experimenting with this, you should ensure that the process will end on its own without user intervention.

If the WinExec function fails, it returns a value less than 32. The potential error values can be found in nearly any API reference, and you may want to add code that will act to solve any errors you encounter. An applet that demonstrates the various ShowWindow parameter options, SHELLEX.MAK, is included with the archive of code from this article, and can be downloaded from the Magazine library of the *VBPJ* Forum on CompuServe.

USING A GETWINDOW LOOP

While it's certainly nice to know when a shelled process has terminated, sometimes you will need to interact with it while it's running. Generally, you must have a window handle (hWnd) to do this. Newcomers to the Windows API often confuse window handles required for other functions with instance handles (hInst) returned by Shell.

To convert an hInst to an hWnd, the most reliable method is to run through what's called a GetWindow loop. Such a loop

Karl E. Peterson is a GIS Analyst with a regional transportation planning agency and a member of the VBPJ *Technical Review Board. He's also an independent programming consultant and a writer based in Vancouver, Washington. The book he coauthored is scheduled for publication concurrent with the upcoming release of a major programming language <g>. 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.*

```
' Win16 API Declarations
Declare Function WinExec Lib "Kernel" (ByVal lpCmdLine _
  As String, ByVal nCmdShow As Integer) As Integer
Declare Function GetModuleUsage Lib "Kernel" (ByVal _
  hModule As Integer) As Integer

' ShowWindow() Constants
Global Const SW_HIDE = 0
Global Const SW_SHOWNORMAL = 1
Global Const SW_SHOWMINIMIZED = 2
Global Const SW_SHOWMAXIMIZED = 3
Global Const SW_SHOWNOACTIVATE = 4
Global Const SW_SHOW = 5
Global Const SW_MINIMIZE = 6
Global Const SW_SHOWMINNOACTIVE = 7
Global Const SW_SHOWNA = 8
Global Const SW_RESTORE = 9

Function ShellAndWait (ByVal CmdLine$, ByVal CmdShow%) _
  As Integer
  Dim hInstShell%
  '
  ' Trim any leading or trailing CmdLine spaces,
  ' and make sure CmdShow is valid.
  '
  CmdLine = Trim$(CmdLine)
  If CmdShow < SW_HIDE Or CmdShow > SW_RESTORE Then
     CmdShow = SW_SHOWNORMAL
  End If
  '
  ' Issue shell directive.
  '
  hInstShell = WinExec(CmdLine, CmdShow)
  If hInstShell >= 32 Then
     '
     ' Program executed normally!  Use Win16
     ' method of waiting for task to complete.
     '
     Do While GetModuleUsage(hInstShell)
        DoEvents
     Loop
     '
     ' Indicate success.
     '
     ShellAndWait = True
  End If
End Function
```

LISTING 1 ***An Enhanced Shell Function.*** *The ShellAndWait function uses the WinExec API to start an application, but expands on Shell's usefulness by supporting all possible ShowWindow styles. ShellAndWait also waits until the spawned application terminates before returning control to your application.*

```
' Required Win16 API declarations
Declare Function FindWindow Lib "User" (ByVal _
  lpClassName As Any, ByVal lpWindowName As Any) _
  As Integer
Declare Function GetParent Lib "User" (ByVal hWnd%) _
  As Integer
Declare Function GetWindowWord Lib "User" (ByVal _
  hWnd%, ByVal nIndex%) As Integer
Declare Function GetWindow Lib "User" (ByVal hWnd%, _
  ByVal wCmd%) As Integer

' Constant used by GetWindowWord to find next window
Global Const GW_HWNDNEXT = 2

Function GetHWndByInst (hInstFind%) As Integer
  Dim hWndTmp%
  '
  ' Find first window and loop through all subsequent
  ' windows in master window list.
  '
  hWndTmp = FindWindow(0&, 0&)
  Do Until hWndTmp = 0
     '
     ' Make sure this window has no parent.
     '
     If GetParent(hWndTmp) = 0 Then
        '
        ' Compare passed hInst against this window's
        ' hInst.
        ' If a match, then return hWnd for current
        ' window.
        '
        If hInstFind% = GetWindowWord(hWndTmp, _
           GWW_HINSTANCE) Then
           GetHWndByInst = hWndTmp
           Exit Do
        End If
     End If
     '
     ' Get next window in master window list and
     ' continue.
     '
     hWndTmp = GetWindow(hWndTmp, GW_HWNDNEXT)
  Loop
End Function
```

LISTING 2 ***Converting an Instance Handle to a Window Handle.*** *The GetHWndByInst function loops through the master task list searching for the first hInst that matches one returned by the Visual Basic Shell function or the WinExec API function.*

finds the first window in the master task list that Windows maintains, and checks various criteria to see if that window is the one of interest. If the desired window was found, the GetWindow loop is exited, and that window's hWnd is returned as the result. Otherwise, the looping process continues until all windows have been inspected.

The GetHWndByInst function uses an hInst to retrieve the corresponding hWnd for an application's main window (see Listing 2). To call this function, simply pass the hInst returned by either Shell or WinExec as its only parameter. GetHWndByInst begins its search by obtaining the hWnd for the first window in the master task list maintained by the system. Passing NULL pointers for both parameters of the FindWindow API function illustrates an apparently undocumented method to obtain this starting window's handle.

Because only top-level windows—those with no parents—are of interest, the first filter applied uses the GetParent API. If a parent exists, the loop continues with the next window. You can use the GetWindowWord API to retrieve the instance handle of any given window. If there is no parent window, the instance handle for the window you are currently testing is compared with the hInst passed into the function. If the instance handles match, the GetHWndByInst function is assigned the corresponding hWnd as its return value, and the loop is exited.

A similar loop is briefly discussed in the Knowledge Base article, "How to Get Windows Master List (Task List) Using Visual Basic," found in the Visual Basic Help file. Here, the same basic construct was used, but additional criteria were applied to filter the results. There are many other ways to take advantage of such a GetWindow loop. This example is simply the first of several methods I'll present in this column.

Have you ever wanted to transfer focus to an application you know is running, but you've been unsure what its current caption was? VB's AppActivate statement requires that you know the *exact* caption, or else AppActivate fails. You can use a GetWindow loop in these instances to walk through the master task list, checking the caption of each parentless window and looking for one that contains or starts with a given string.

To construct this useful function, add the required API declarations and constant definitions (see Listing 3). Then, add the AppActivatePartial function that calls the FindWindowPartial function (both functions are also shown in Listing 3). This

```
' Required Win16 API declarations
Declare Function FindWindow Lib "User" (ByVal _
  lpClassName As Any, ByVal lpWindowName As Any) As Integer
Declare Function SetActiveWindow Lib "User" (ByVal hWnd _
  As Integer) As Integer
Declare Function GetWindow Lib "User" (ByVal hWnd As _
  Integer, ByVal wCmd As Integer) As Integer
Declare Function GetWindowText Lib "User" (ByVal hWnd As _
  Integer, ByVal lpString As String, ByVal aint As _
  Integer) As Integer
Declare Function GetParent Lib "User" (ByVal hWnd As _
  Integer) As Integer

' Constant used by GetWindowWord to find next window
Global Const GW_HWNDNEXT = 2

' Constants used by FindWindowPartial
Global Const FWP_STARTSWITH = 0
Global Const FWP_CONTAINS = 1

Sub AppActivatePartial (TitleContains$, Method%)
  Dim hWndApp As Integer
  Dim nRet As Integer
  '
  ' Retrieve window handle for first top-level window
  ' that starts with or contains the passed string.
  '
  hWndApp = FindWindowPartial(TitleContains, Method)
  If hWndApp Then
     '
     ' Switch to it.
     '
     nRet = SetActiveWindow(hWndApp)
  Else
     '
     ' Alert user that request failed.
     '
     MsgBox "No matching applications found."
  End If
End Sub

Function FindWindowPartial (ByVal TitlePart$, Method%) _
  As Integer
  Dim hWndTmp As Integer
  Dim nRet As Integer
  Dim TitleTmp As String
  '
  ' Alter partial title for case-insensitive compares.
  '
  TitlePart = UCase(TitlePart)
  '
  ' Find first window and loop through all subsequent
  ' windows in master window list.
  '
  hWndTmp = FindWindow(0&, 0&)
  Do Until hWndTmp = 0
     '
     ' Make sure this window has no parent.
     '
     If GetParent(hWndTmp) = 0 Then
        '
        ' Retrieve caption text from current window.
        '
        TitleTmp = Space(256)
        nRet = GetWindowText(hWndTmp, TitleTmp, _
           Len(TitleTmp))
        If nRet Then
           '
           ' Clean up return string, preparing for
           ' case-insensitive comparison.
           '
           TitleTmp = UCase(Left(TitleTmp, nRet))
           '
           ' Use appropriate method to determine if
           ' current window's caption either starts
           ' with or contains passed string.
           '
           Select Case Method
              Case FWP_STARTSWITH
                 If InStr(TitleTmp, TitlePart) = 1 Then
                    FindWindowPartial = hWndTmp
                    Exit Do
                 End If
              Case FWP_CONTAINS
                 If InStr(TitleTmp, TitlePart) Then
                    FindWindowPartial = hWndTmp
                    Exit Do
                 End If
           End Select
        End If
     End If
     '
     ' Get next window in master window list and continue.
     '
     hWndTmp = GetWindow(hWndTmp, GW_HWNDNEXT)
  Loop
End Function
```

LISTING 3 ***Finding a Window With Only a Partial Caption.*** *The FindWindowPartial function uses a GetWindow loop to find the first top-level window whose caption matches that caption passed in. You have the option of matching captions that either start with or contain the search text and are case-insensitive. The AppActivatePartial subroutine uses FindWindowPartial to convert an hInst (returned by Shell) to an hWnd, then activates the found application.*

method has been separated into two distinct functions for when you simply want to obtain the hWnd for the first window that matches a given caption.

Call AppActivatePartial with the string you're looking for and a constant indicating whether the caption must start with, or simply contain, the partial title string. For example, Microsoft Word's caption changes to reflect the current document the program is processing. To locate a running instance of Word, you could call AppActivatePartial in one of two ways. While the first choice would have a high probability of success if Word were running, the second option could just as easily turn up WordPad if your program were running under Win95. It's always best to be as specific as possible:

```
AppActivatePartial "Microsoft Word", FWP_STARTSWITH
AppActivatePartial "Word", FWP_CONTAINS
```

AppActivatePartial works by calling FindWindowPartial with the same method constant as it received. If FindWindowPartial returns a valid hWnd, AppActivatePartial uses the SetActiveWindow API to transfer focus to that application. Because this routine is just a demonstration of the technique, I used a MsgBox to alert the user if the routine did not find the application. In a production application, you'd want to replace the MsgBox with more suitable error-handling code. In addition to SetActiveWindow you may also want to use ShowWindow to change the target application's WindowState, or MoveWindow to move or resize it.

The FindWindowPartial function is very similar to the GetHWndByInst function. Before entering the GetWindow loop, the passed partial title string is uppercased so that comparisons won't take case into consideration. Again, you use FindWindow to obtain the first window, and GetParent to filter out child windows. For all the top-level windows found, use the GetWindowText API to obtain their captions. Uppercase these, then search with Visual Basic's Instr function to see if they either start with or contain the partial title string. A demonstration applet, FINDPART.MAK, is included with the archive of code from this article, and can be downloaded from the *VBPJ* Forum on CompuServe. ■