Windows Event Binding Made Easy

Doug Hennig

A feature available in other development environments but missing in VFP is the ability to capture Windows events. VFP 9 extends the BINDEVENT() function to allow our own code to be called when Windows passes certain messages to VFP windows. This has a wide range of uses, some of which Doug examines in this month's article.

Windows communicates events to applications by passing them messages. Although VFP exposes some of these messages through events in VFP objects, such as MouseDown and Click, many messages are not available to VFP developers. One common request is the ability to detect an application switch. For example, I created an application that hooks into GoldMine, a popular contact management system, displaying additional information about the current contact. If the user switches to GoldMine, moves to a different contact, and then switches back to my application, it would be nice to refresh the display so it shows information about the new contact. Unfortunately, there was no way to do this in earlier versions of VFP; I had to rely on a timer that constantly checked which contact was currently displayed in GoldMine.

VFP 9 extends the BINDEVENT() function added in VFP 8 to support Windows messages. The syntax for this use is:

```
bindevent(hWnd, nMessage, oEventHandler, cDelegate)
```

where hWnd is the Windows handle for the window that receives events, nMessage is the Windows message number, and oEventHandler and cDelegate are the object and method that's fired when the message is received by the window. Unlike with VFP events, only one handler can bind to a particular hWnd and nMessage combination. Specifying a second event handler object or delegate method causes the first binding to be replaced with the second. VFP doesn't check for valid hWnd or nMessage values; if either is invalid, nothing happens because the specified window can't receive the specified message.

For hWnd, you can specify _Screen.hWnd or _VFP.hWnd to trap messages sent to the application or a form's hWnd for those messages sent to the form. VFP controls don't have a Windows handle, but ActiveX controls do, so you also bind to them.

There are hundreds of Windows messages. Examples of such messages are: WM_POWERBROADCAST (0x0218), sent when a power event occurs such as low battery or switching to standby mode; WM_THEMECHANGED (0x031A), which indicates the Windows XP theme has changed; and WM_ACTIVATE (0x0006), raised when switching to or from an application. (Windows messages are usually referred to by a name starting with WM_.) Documentation for almost all Windows messages is available at http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/windowui.asp. The values for the WM_ constants are in the WinUser.H file that's part of the Platform SDK, which you can download from http://www.microsoft.com/msdownload/platformsdk/sdkupdate/.

The event handler method must accept four parameters: hWnd, the handle for the window that received the message, nMessage, the Windows message number, and two Integer parameters, the contents of which vary depending on the Windows message (the documentation for each message describes the values of these parameters). The method must return an Integer, which contains a result value. One of the possible return values is BROADCAST_QUERY_DENY (0x424D5144, which represents the string "BMQD") that prevents the event from occurring.

If you want the message to be processed in the normal manner, which is something most event handlers should do, you have to call the VFP Windows message handler in your event handler method; this is sort of like using DODEFAULT() in VFP method code. Your event handler method most likely returns the return value of the VFP Windows message handler. Here's an example of an event handler that does this (it does nothing else):

```
lparameters hWnd, ;
  Msg, ;
  wParam, ;
  lParam
local lnOldProc, ;
```

```
  lnResult
#define GWL_WNDPROC -4
declare integer GetWindowLong in Win32API ;
  integer hWnd, integer nIndex
declare integer CallWindowProc in Win32API ;
  integer lpPrevWndFunc, integer hWnd, integer Msg, ;
  integer wParam, integer lParam
lnOldProc = GetWindowLong(_screen.hWnd, GWL_WNDPROC)
lnResult  = CallWindowProc(lnOldProc, hWnd, Msg, ;
  wParam, lParam)
return lnResult
```

Of course, the event handler doesn't need to declare the Windows API functions or call GetWindowLong each time; you could put that code in the Init method of the class, storing the return value of GetWindowLong in a custom property, and then using that property in the call to CallWindowProc in the event handler. The rest of the examples show this.

To determine which messages are bound, use AEVENTS(ArrayName, 1). It fills the specified array with one row per binding and four columns, containing the values of the parameters passed to BINDEVENT().

You can unbind events using UNBINDEVENT(hWnd [, nMessage]). Omitting the second parameter unbinds all messages for the specified window. Pass only 0 to unbind all messages for all windows. Events are also automatically unbound the next time the message occurs after the event handling object is destroyed.

What would a new version of VFP be without some new SYS() functions? The VFP team added three SYS() functions related to Windows events in VFP 9. SYS(2325, wHandle) returns the wHandle (an internal VFP wrapper for hWnd) for the client window of the window whose wHandle is passed as a parameter. (A client window is a window inside a window; for example, _Screen is a client window of _VFP.) SYS(2326, nWnd) returns the wHandle for the window specified with hWnd. SYS(2327, wHandle) returns the hWnd for the window specified with wHandle. The documentation for these functions indicates they're for BINDEVENT() scenarios using the VFP API Library Construction Kit. However, you can also use them to get the hWnd for the client window of a VFP IDE window, as you'll see in the next example.

Binding to VFP IDE window events

TestWinEventsForIDE.PRG, included in this month's Subscriber Downloads, demonstrates event binding to VFP IDE windows. Set lcCaption to the caption of the IDE window you want to bind events to (the code shown below uses the Command window), then run the program. Activate and deactivate the window, move it, resize it, and so forth; you should see Windows events echoed to the screen. When you finish, type RESUME and press Enter in the Command window to clean up. To test this with the client window of an IDE window, uncomment the indicated code. You can also bind to other events by adding BINDEVENT() statements to this code; use the constants in WinEvents.H for the values for the desired events. Note that TestWinEventsForIDE.PRG only works with non-dockable IDE windows, so before you run this program, right-click in the title bar of the window you want to test and ensure Dockable is turned off.

Here's the code for this PRG:

```
#include WinEvents.H

lcCaption      = 'Command'
loEventHandler = createobject('IDEWindowsEvents')
lnhWnd         = loEventHandler.FindIDEWindow(lcCaption)
* Uncomment this code to receive events for the window's
* client window instead
*lnhWnd         = loEventHandler.FindIDEClientWindow(lcCaption)
if lnhWnd > 0
  bindevent(lnhWnd, WM_SETFOCUS,      loEventHandler, ;
    'EventHandler')
  bindevent(lnhWnd, WM_KILLFOCUS,     loEventHandler, ;
    'EventHandler')
  bindevent(lnhWnd, WM_MOVE,          loEventHandler, ;
    'EventHandler')
  bindevent(lnhWnd, WM_SIZE,          loEventHandler, ;
    'EventHandler')
```

```
  bindevent(lnhWnd, WM_MOUSEACTIVATE, loEventHandler, ;
    'EventHandler')
  bindevent(lnhWnd, WM_KEYDOWN,       loEventHandler, ;
    'EventHandler')
  bindevent(lnhWnd, WM_KEYUP,         loEventHandler, ;
    'EventHandler')
  bindevent(lnhWnd, WM_CHAR,          loEventHandler, ;
    'EventHandler')
  bindevent(lnhWnd, WM_DEADCHAR,      loEventHandler, ;
    'EventHandler')
  bindevent(lnhWnd, WM_KEYLAST,       loEventHandler, ;
    'EventHandler')
  clear
  suspend
  unbindevents(0)
  clear
else
  messagebox('The ' + lcCaption + ;
    ' window was not found.')
endif lnhWnd > 0

define class IDEWindowsEvents as Custom
  cCaption = ''
  nOldProc = 0

  function Init
    declare integer GetWindowLong in Win32API ;
      integer hWnd, integer nIndex
    declare integer CallWindowProc in Win32API ;
      integer lpPrevWndFunc, integer hWnd, integer Msg, ;
      integer wParam, integer lParam
    declare integer FindWindowEx in Win32API;
      integer, integer, string, string
    declare integer GetWindowText in Win32API ;
      integer, string @, integer
    This.nOldProc = GetWindowLong(_screen.hWnd, ;
      GWL_WNDPROC)
  endfunc

  function FindIDEWindow(tcCaption)
    local lnhWnd, ;
      lnhChild, ;
      lcCaption
    This.cCaption = tcCaption
    lnhWnd        = _screen.hWnd
    lnhChild      = 0
    do while .T.
      lnhChild = FindWindowEx(lnhWnd, lnhChild, 0, 0)
      if lnhChild = 0
        exit
      endif lnhChild = 0
      lcCaption = space(80)
      GetWindowText(lnhChild, @lcCaption, len(lcCaption))
      lcCaption = upper(left(lcCaption, ;
        at(chr(0), lcCaption) - 1))
      if lcCaption = upper(tcCaption)
        exit
      endif lcCaption = upper(tcCaption)
    enddo while .T.
    return lnhChild
  endfunc

  function FindIDEClientWindow(tcCaption)
    local lnhWnd, ;
      lnwHandle, ;
      lnwChild
    lnhWnd = This.FindIDEWindow(tcCaption)
    if lnhWnd > 0
      lnwHandle = sys(2326, lnhWnd)
      lnwChild  = sys(2325, lnwHandle)
      lnhWnd    = sys(2327, lnwChild)
```

```
    endif lnhWnd > 0
    return lnhWnd
  endfunc

  function EventHandler(hWnd, Msg, wParam, lParam)
    ? 'The ' + This.cCaption + ;
      ' window received event #' + transform(Msg)
    return CallWindowProc(This.nOldProc, hWnd, Msg, ;
      wParam, lParam)
  endfunc
enddefine
```

The program starts by instantiating the IDEWindowsEvents class. It calls the FindIDEWindow method to get a handle to the window whose caption is stored in the lcCaption variable. It then uses BINDEVENT() to bind certain events from the desired window to the EventHandler method of the class. These events include activating, deactivating, resizing, and moving the window, and keypresses within the window.

The Init method of the IDEWindowsEvents class declares the Windows API functions used by the class. It also determines the value used to call the VFP Windows message handler and stores it in the nOldProc property; this value is used by the EventHandler method to ensure normal event handling occurs. The FindIDEWindow method uses a couple of Windows API functions to find the specified VFP IDE window. It does this by looking at each child window of _VFP to see if its caption matches the caption passed as a parameter. FindIDEClientWindow does something similar, but uses the new SYS() functions to get the handle for the client window of the specified window.

When you run TestWinEventsForIDE.PRG, you will find not all events occur for all IDE or client windows. For example, you won't see keypress events for the Properties window. This is likely due to the way VFP implements windows, which is somewhat different from other Windows applications.

Note that you won't use code like this in a typical application; it's intended for developers who want to add behavior to the VFP IDE. The next example is something you might use in an end-user application.

Binding to application window and disk events

WindowsMessagesDemo.SCX (see **Figure 1**) demonstrates hooking into activate and deactivate events as well as certain Windows shell events, such as inserting or removing a CD or USB drive. The latter shows an interesting use of Windows events: the code registers _VFP to receive a subset of Windows shell events as a custom Windows event.

The Init method of this form handles the necessary setup. As with TestWinEventsForIDE.PRG, it declares several Windows API functions and stores the value for the VFP Windows event handler in the nOldProc property. The call to SHChangeNotifyRegister tells Windows to register _VFP to receive disk events, media insertion and removal events, and drive addition and removal events using the custom message WM_USER_SHNOTIFY. (Items in upper-case in this example are constants defined in WinEvents.H or ShellFileEvents.H.) This code then binds activate events for the form and device change events and the custom message it just defined for _VFP to the HandleEvents method of the form. Note: The call to SHChangeNotifyRegister requires Windows XP or later. If you're using an earlier operating system, comment out the assignment statement for This.nSHNotify.

```
local lcSEntry

* Declare the Windows API functions we'll use.

declare integer GetWindowLong in Win32API ;
  integer hWnd, integer nIndex
declare integer CallWindowProc in Win32API ;
  integer lpPrevWndFunc, integer hWnd, integer Msg, ;
  integer wParam, integer lParam
declare integer SHGetPathFromIDList in shell32 ;
  integer nItemList, string @szPath
declare integer SHChangeNotifyRegister in shell32 ;
  integer hWnd, integer fSources, integer fEvents, ;
  integer wMsg, integer cEntries, string @SEntry
declare integer SHChangeNotifyDeregister in shell32 ;
  integer
```

```
* Get a handle for the VFP Windows event handler.

This.nOldProc = GetWindowLong(_screen.hWnd, GWL_WNDPROC)

* Register us to receive certain shell events as a custom
* Windows event.

lcSEntry = replicate(chr(0), 8)
This.nShNotify = SHChangeNotifyRegister(_vfp.hWnd, ;
  SHCNE_DISKEVENTS, SHCNE_MEDIAINSERTED + ;
  SHCNE_MEDIAREMOVED + SHCNE_DRIVEADD + ;
  SHCNE_DRIVEREMOVED, WM_USER_SHNOTIFY, 1, @lcSEntry)

* Bind to the Windows events we're interested in.

bindevent(This.hWnd, WM_ACTIVATE,      This, ;
  'HandleEvents')
bindevent(_vfp.hWnd, WM_DEVICECHANGE,  This, ;
  'HandleEvents')
bindevent(_vfp.hWnd, WM_USER_SHNOTIFY, This, ;
  'HandleEvents')

* Hide the VFP main window so it's easier to see what's
* going on.

_screen.Visible = .F.
```

The HandleEvents method handles the registered events. It uses a CASE statement to determine which event occurred and updates the Caption of the status label on the form appropriately. Certain event types have "subevents" as indicated by the wParam parameter; this is used to determine the exact event that occurred. For example, when a WM_ACTIVATE event occurs, wParam specifies whether the window was activated or deactivated, and whether activation occurred by task switching (such as Alt-Tab) or by clicking on the window.

Handling the custom shell event is a little more complicated than the other events. In this case, lParam specifies the specific event and wParam contains the address where the path for the drive that was inserted or removed is stored. So, SYS(2600) is used to copy the value from the address, the custom BinToInt method (not shown here) converts the value to an integer, and the SHGetPathFromIDList Windows API function provides the actual path from the integer. Finally, this method calls the HandleWindowsMessage method, which simply calls CallWindowProc to get the normal event handling behavior. Here's the code for HandleEvents:

```
lparameters hWnd, ;
  Msg, ;
  wParam, ;
  lParam
local lcCaption, ;
  lnParm, ;
  lcPath
do case

* Handle an activate or deactivate event.

  case Msg = WM_ACTIVATE
    do case

* Handle a deactivate event.

      case wParam = WA_INACTIVE
        This.lblStatus.Caption = 'Window deactivated'

* Handle an activate event (task switch or clicking on
* the title bar).

      case wParam = WA_ACTIVE
        This.lblStatus.Caption = 'Window activated ' + ;
          '(task switch)'
```

```
* Handle an activate event (clicking in the client area
* of the window).

      case wParam = WA_CLICKACTIVE
        This.lblStatus.Caption = 'Window activated ' + ;
          '(click)'
    endcase

* Handle a device change event.

  case Msg = WM_DEVICECHANGE
    do case
      case wParam = DBT_DEVNODES_CHANGED
        This.lblStatus.Caption = 'DevNodes changed'
      case wParam = DBT_DEVICEARRIVAL
        This.lblStatus.Caption = 'Device arrival'
      case wParam = DBT_DEVICEREMOVECOMPLETE
        This.lblStatus.Caption = 'Device removal ' + ;
          'complete'
    endcase

* Handle a custom shell notify event.

  case Msg = WM_USER_SHNOTIFY
    do case
      case lParam = SHCNE_DRIVEADD
        lcCaption = 'Drive added'
      case lParam = SHCNE_DRIVEREMOVED
        lcCaption = 'Drive removed'
      case lParam = SHCNE_MEDIAINSERTED
        lcCaption = 'Media inserted'
      case lParam = SHCNE_MEDIAREMOVED
        lcCaption = 'Media removed'
    endcase
    lnParm = This.BinToInt(sys(2600, wParam, 4))
    lcPath = space(270)
    SHGetPathFromIDList(lnParm, @lcPath)
    lcPath = left(lcPath, at(chr(0), lcPath) - 1)
    This.lblStatus.Caption = lcCaption + ': ' + lcPath
endcase
return This.HandleWindowsMessage(hWnd, Msg, wParam, ;
  lParam)
```

Run the form and, as indicated in the instructions, click on other windows or the desktop and back on the form to show activate and deactivate events. Insert and remove a removable device of some type, such as USB drive or a digital camera, to see the events that occur.

There are several practical uses for the type of code shown in this form. For example, the GoldMine add-on I mentioned at the beginning of the article can now refresh itself when it receives an activate event. When I attach my digital camera to my computer with a USB cable, the software that comes with the camera pops up and prompts me to download the pictures. A VFP real estate or medical imaging application could do something similar with pictures of houses or injuries.

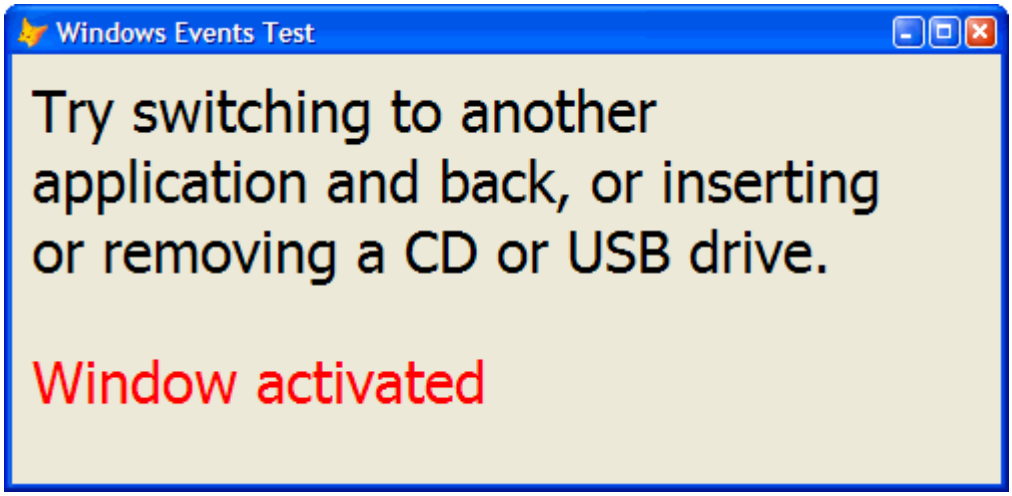

Figure 1. WindowsMessagesDemo.SCX demonstrates several Windows events that your applications may be interested in.

Other uses

There are a lot of other uses for Windows events. For example, you may wish to prevent Windows from shutting down under certain conditions, such as an import process not being complete. In that case, you'd bind to the WM_POWERBROADCAST message and return BROADCAST_QUERY_DENY if the shutdown should be halted.

I use Microsoft Money for doing my home finances and have always liked the fact that when I download a statement from my bank, Money immediately knows about it and displays the appropriate dialog. That type of behavior is now possible in VFP applications; rather than constantly polling a directory to see if a file has been added (or removed or renamed or whatever), your application can now be notified as soon as that occurs and take the appropriate action.

Summary

Support for Windows event binding is an incredible addition to VFP; it allows you to hook into just about anything that goes on in Windows. I expect to see many cool uses of this as the VFP community starts to learn about its capabilities.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com