FoxRockX

Put Event Binding to Work, Part 2

Use BindEvent() to make your applications easier to use.

Tamar E. Granor, Ph.D.

In my last article, I gave an overview of the BindEvent() function and showed some of the ways I use it to improve my applications. In this article, we'll see some additional examples, including one that binds to a Windows event.

Let's start with a quick review. BindEvent(), which was added in VFP 8, lets you specify a method that should run whenever a particular event fires. The syntax is shown in Listing 1. You can read it as saying "Whenever oEventSource.cMethod fires, run oEventHandler.cDelegateMethod." The nFlags parameter determines which method runs first, among other things. By default, it's the delegate method.

Listing 1. The BindEvent() function lets you bind one method to another.

```
BINDEVENT( oEventSource, cMethod, ;
           oEventHandler, cDelegateMethod, ;
           nFlags)
```

As in the previous article, the examples here (with one exception) are drawn from a small application (originally written to demonstrate user interface practices) that manages a lending library. The Library application is included in this month's downloads.

May 2012
Number 26

Resizing toolbars

The addition of anchors in VFP 9 made it much easier to properly resize controls when a form is resized. But there are situations where anchoring doesn't do the trick. In one application, I use controls inside a toolbar to provide dockable forms inside the application's main form (which is a top-level form). Toolbars don't have an Anchor property; their size is controlled by the size of their contents. When the user resizes the application, I want to ensure that the toolbar resizes; in order to do so, I have to resize the objects in the toolbar. BindEvent() makes this possible. However, if not done carefully, it can also crash the application.

I first built this functionality to provide a status bar for a top-level form that is the main form of the application. I put the ActiveX StatusBar control inside a toolbar. I bound the form's Resize method to a custom method of the toolbar, called ResizeStatus.

What makes this operation difficult is that the form's Resize method fires repeatedly as long as you're resizing the form, rather than once when you're finished. Resetting the status bar's size each time the form's Resize method fires causes the toolbar to resize itself repeatedly, which affects the size of the form, which then calls the ResizeStatus method, and so on and so forth. That sequence eventually crashes VFP 9.

So I needed a way to control the resizing and do it only once, each time the form is resized. I subclassed the OLEControl class and added the status bar control to it. That class, sbrMSStatus, has a custom ResizeStatus method that contains the code in Listing 2.

Listing 2. The status bar control's custom ResizeStatus method sets its width to a specified value.

```
LPARAMETERS nNewWidth

IF VARTYPE(m.nNewWidth) = "N"
   This.Width = m.nNewWidth
ELSE
   This.Width = ThisForm.Width
ENDIF
```

The toolbar class contains an instance of sbrMSStatus, and has two custom properties. oForm contains an object reference to the containing form, while lResizingNow is a flag to indicate whether we're in the middle of resizing. The toolbar has a custom ResizeStatus method, containing the code in Listing 3.

Listing 3. The toolbar's ResizeStatus method ensures that we resize the status bar just once.

```
LOCAL aFired[1]
IF NOT This.lResizingNow
   This.lResizingNow = .T.
   * Need width of calling form to pass in
   AEVENTS(aFired, 0)
   This.oStatusBar.ResizeStatus( ;
     aFired[1].Width)
   This.lResizingNow = .F.
ENDIF
```

The toolbar's Init method, shown in Listing 4, populates the oForm property, and the BeforeDock method, shown in Listing 5, ensures that the statusbar fills the full width of the form when you dock it. (The code should probably make sure that you're only docking it top or bottom before adjusting the width.)

Listing 4. The toolbar holds a pointer to the containing form, stored in the Init method.

```
DODEFAULT()

This.oForm = _VFP.ActiveForm
```

Listing 5. The toolbar's BeforeDock method makes the statusbar's width match the form's.

```
LPARAMETERS nLocation

* Make sure status bar fills form width
This.oStatusBar.Width = This.oForm.Width
```

The form does its portion of the work in the Activate method. If the toolbar doesn't already exist, it's instantiated and the form's Resize method is bound to the toolbar's ResizeStatus method. If the toolbar isn't already docked at the bottom, we then dock it. Listing 6 shows the Activate code.

Listing 6. The form's Activate method sets up the status bar toolbar the first it runs.

```
IF ISNULL(This.oStatusBar)
   This.oStatusBar = ;
     NEWOBJECT("tbrStatusBar", "ebControls")
   BINDEVENT(This, "Resize", ;
     This.oStatusBar, "ResizeStatus", 1)
   This.oStatusBar.Show()
ENDIF

IF This.oStatusBar.DockPosition <> 3
   This.oStatusBar.Dock(3)
ENDIF
```

Note that, even with this approach, this toolbar is not compatible with the Library application's main form where buttons in toolbars may take on different sizes. That scenario crashes VFP.

This is the first example I've shown where the delegate method runs *after* the source object's method. We need to do things in that order so that the form's Width will have changed by the time we want to pass it to the status bar's ResizeStatus method.

There's an example of a form with a status bar toolbar in the downloads for this article, but it's not used in the Library application because that application docks and undocks other toolbars. To try it, run ebMain.PRG.

Monitoring user activity

In some applications, we want to pay attention to whether the user is doing anything. We might be running a background process that should stop when the user wants to do something else, or we might want to log the user out after a period of inactivity. In either case, we need to know any time the user moves the mouse or types. BindEvent() gives us a way to track user activity.

Tracking user activity is an application-level task, so the application class, cusApp, needs several custom properties. The first two manage activity tracking: lTrackUserActivity, which indicates whether we're tracking user activity; and oActivityTimer, a reference to a timer object used for tracking. Two additional properties let us indicate what timer to use for activity tracking: cTimerClass and cTimerClassLib point to the timer class to use.

The application class's Init method sets up the timer, if the lTrackUserActivity property is true, as shown in Listing 7.

Listing 7. The application class's Init method sets up a timer to track user activity.

```
IF This.lTrackUserActivity
   This.SetupActivityTimer()
ENDIF
```

The SetupActivityTimer method, shown in Listing 8, simply instantiates the timer and stores a reference in the oActivityTimer property. TRY-CATCH is used in case there's a problem.

Listing 8. Setting up the activity timer is as easy as instantiating the right object.

```
LOCAL lSuccess

TRY
   This.oActivityTimer = ;
     NEWOBJECT(This.cTimerClass, ;
       This.cTimerClassLib)
   lSuccess = .T.
CATCH
   lSuccess = .F.
ENDTRY

RETURN m.lSuccess
```

The base form class (frmBase) has a custom property, lBindUserActions, and a custom method, BindUserActions. In the form's Init method, we check the property and if it's true, call the method, as shown in Listing 9.

Listing 9. The lBindUserActions property determines whether to track user activity in the form.

```
IF This.lBindUserActions
   This.BindUserAction()
ENDIF
```

You might wonder why we need the lBindUserActions property at the form-level when we already track this at the application-level. There may be situations where a particular form should be excluded from activity tracking, or conversely, where only actions on certain forms should be considered user activity.

The form's BindUserActions method, shown in Listing 10, calls a method of the application object. A little extra code ensures that we can run forms stand-alone for testing. Note that we pass an object reference to the calling form to the application's BindUserActions method.

Listing 10. The BindUserActions method of the "base" form class asks the application's timer to watch this form.

```
* If we have an application-level object
* watching for user activity, bind things
* in this form to it.

IF TYPE("goApp.oActivityTimer") = "O"
   goApp.BindUserActions(THIS)
ENDIF
```

The application's BindUserActions method confirms that we're tracking and asks the timer to do the actual binding; it's shown in Listing 11. It passes along the reference to the form.

Listing 11. The application's BindUserActions method delegates the actual task of binding to the activity timer.

```
LPARAMETERS oObject
* Bind user actions in the specified object to
* the activity timer.

IF This.lTrackUserActivity
   * Only do this if we're tracking
   IF ISNULL(This.oActivityTimer)
      * In case we haven't already set it up,
      * do so now
      This.SetupActivityTimer()
   ENDIF

This.oActivityTimer.BindUserActionInObject( ;
  m.oObject)
ENDIF

RETURN
```

All the real work happens in the timer object; tmrUserActivity is derived from the base timer class (tmrBase) and has one custom property, lUserActed, which is .T. when there has been user activity within the specified time frame and .F. when there hasn't. The timer has two custom methods, BindUserActionInObject and UserActivity. BindUserActionInObject, shown in Listing 12 and called from the application object's BindUserActions method, binds the KeyPress and MouseMove events of every control in the specified object to the UserActivity method, and drills down recursively to ensure that no matter where the user types or moves the mouse, we catch it.

Listing 12. The activity timer's BindUserActionInObject method binds the KeyPress and MouseMove events of the specified object and every control it contains to the timer's UserActivity method.

```
LPARAMETERS oObject

IF PEMSTATUS(oObject, "KeyPress", 5)
   BINDEVENT(oObject, "KeyPress", ;
             This, "UserActivity")
ENDIF

IF PEMSTATUS(oObject, "MouseMove", 5)
   BINDEVENT(oObject, "MouseMove", ;
             This, "UserActivity")
ENDIF

IF PEMSTATUS(oObject, "Objects", 5)
   FOR EACH oChild IN oObject.Objects
      This.BindUserActionInObject(m.oChild)
   ENDFOR
ENDIF

RETURN
```

Thus, any user action on the form fires the UserActivity method of the timer.

The tmrUserActivity class simply manages the lUserActed flag; it needs to be enhanced or subclassed to actually do something based on user activity or inactivity. Here, the UserActivity method sets the flag and resets the timer, so that it starts watching for inactivity again. The method is shown in Listing 13.

Listing 13. The UserActivity method fires every time the user types or moves the mouse.

```
LPARAMETERS uParm1, uParm2, uParm3, uParm4
* Parameters for bound events

This.lUserActed = .T.
* So start counting again.
This.Reset()

RETURN
```

The parameters to UserActivity are worth mentioning. When a method is a delegate for an event, it must accept the same parameters as the bound event method. Since MouseMove accepts four parameters, we need four parameters, though we're not doing anything with them. (KeyPress accepts two parameters, but accepting four here won't cause any problems.)

The timer's Timer method clears the lUserActed flag because when it fires, it indicates that the specified time has passed without any user activity.

To make it easier to write code to respond to user activity or inactivity, the custom lUserActed property has an Assign method. (I'll write about assign methods in a future article.) That method fires each time we change lUserActed, whether from UserActivity or Timer. In a subclass, we can put code in that method to take the appropriate action based on the new value.

In the application for which I originally created the activity timer, the goal was to begin a background activity (polling for changes on a piece of dedicated hardware) after the user was inactive for 10 seconds, and stop it as soon as the user became active again.

A simpler example, included in the Library application, is to ask the user whether to shut down the application after a period of inactivity. To create this timer, I subclassed tmrUserActivity to create tmrShutDown. The Interval property is set to 60000, which is 1 minute (60,000 milliseconds). In a real application, you'd probably set the Interval much higher; one minute of inactivity is awfully short for shutting down an application. The lUserActed_Assign method, shown in Listing 14, checks whether the new value for the property is .T. or .F. If it's .F., the user is prompted to indicate whether to shut down the app. Since a period of inactivity may mean that the user is no longer sitting in front of the computer, the InputBox() used to prompt the user has a timeout of one minute and a timeout default of "Y" to shut down the app.

Listing 14. This code, in the lUserActed_Assign method of the timer subclass, asks the user whether to shut down the app. If the user says yes, or doesn't answer within a minute, the app is shut down.

```
LPARAMETERS tuNewValue

LOCAL cResponse

* Shut down the application, if inactivity and
* user agrees
IF NOT m.tuNewValue
   cResponse =   INPUTBOX( ;
     "You don't seem to be doing anything. " ;
     + "Do you want to shut down this " + ;
     + "application (Y/N)?", ;
     "No activity", "Y", 60000, "Y", "N")
   IF UPPER(m.cResponse) = "Y"
      IF TYPE("goApp") = "O" AND ;
         NOT ISNULL(m.goApp)
         goApp.ShutDownApp()
      ENDIF
   ENDIF
ENDIF

This.lUserActed = m.tuNewValue
```

The ShutDownApp method of the application object, as its name suggests, shuts down the application.

The Library application uses this set-up and lets the user decide (via the Preferences form) whether to shut down after a period of inactivity and how long that period should be.

Updating when a form closes

Sometimes in an application, you need to update data in one form when another closes. When the form that's closing is modal and was called from the other form, this is easy because you're still in the method that called the other form in the first place. But when both forms are modeless, you need a way to connect them; BindEvent() offers one way.

For simplicity, we'll assume that the form that needs to be updated called the form that's closing. All you need in that case is to bind the Destroy event of the called form to a method of the calling form, as in Listing 15.

Listing 15. The code runs a form, and binds its Destroy method to the Refresh method of the calling form.

```
LPARAMETERS cChildForm

LOCAL oChild

TRY
   DO FORM (m.cChildForm) NAME m.oChild

   IF NOT ISNULL(m.oChild)
      BINDEVENT(m.oChild, "Destroy", ;
                This, "Refresh")
   ENDIF
CATCH
   * Nothing to do here, maybe tell user.
ENDTRY

RETURN
```

In the Library application, the context menu of the check-out form lets you open the Members form, looking at the current member. If the user changes some of the Member's data, we want to update the check-out form. Because updating that data is a little more complicated than just calling the form's Refresh, we bind the Members form's Destroy to a custom method of the check-out form, RefreshMember, shown in Listing 16. (The GetMember method retrieves the member data and puts it in a cursor.)

Listing 16. This custom method of the check-out form refreshes the display of the current member. It's called when the borrowers form closes.

```
* Update the display for the current member
This.GetMember(This.cCurrentMemberNum)

RETURN
```

The shortcut menu item "Show this member's record" calls another custom method named ShowBorrower, to set things up; that method is shown in Listing 17. We pass 1 for the flags parameter of BindEvent here to be sure that we finish closing the form (and thus, data is saved) before we do the refresh.

Listing 17. This method of the check-out form, called ShowBorrower, is called when the user asks to see data for the current borrower.

```
LPARAMETERS cBorrowerNum

LOCAL oBorrowerForm

DO FORM Borrowers WITH m.cBorrowerNum ;
   NAME oBorrowerForm

IF VARTYPE(m.oBorrowerForm) = "O" AND ;
   NOT ISNULL(m.oBorrowerForm)
   * Make sure we refresh this form when the
   * borrower form closes
   BINDEVENT(oBorrowerForm, "Destroy", ;
             This, "RefreshMember", 1)
ENDIF
```

Of course, in this case, we might actually want to do the update not just when the borrower form closes, but when we save the data in that form. We can bind to the borrower form's Save method to get that behavior.

Updating colors when the color theme changes

I'm a big believer is sticking to the user's chosen color scheme for most applications. Thus, I rarely set colors for VFP controls or forms. However, there are times when I want to highlight some feature. Rather than choosing a color I like, I prefer to pick a color out of the user's selected color scheme. The Windows API function GetSysColor pulls the user's colors from the registry, so I can use them.

If the user changes the color scheme while an application is running, I want my application to follow his lead. To do this, I bind to the Windows WM_ThemeChanged and WM_SysColorChange messages.

I've wrapped all this functionality up into a class called GetUserColors, based on the Custom class. The class includes methods to retrieve the colors from the user's current scheme, and to return a particular color from that scheme. (There's a set of names for the various colors that roughly matches what you see in the Advanced Appearance dialog of the Display Properties dialog.) I won't go through all the code for extracting the user's colors; it's fairly straightforward.

The BindColorChanges method, called from the Init method, sets up the binding; it's shown in Listing 18. The two API function declarations and the call that follows them store information about the Windows procedure to call when handling the event.

Listing 18. These method binds to two Windows messages, so the application respond when the user changes colors.

```
PROCEDURE BindColorChanges
* Bind the info here to changes in the user's
* theme/scheme

* Prepare for binding
DECLARE integer CallWindowProc IN WIN32API ;
   integer lpPrevWndFunc, ;
   integer hWnd,integer Msg,;
   integer wParam,;
   integer lParam

DECLARE integer GetWindowLong IN WIN32API ;
   integer hWnd, ;
   integer nIndex

THIS.nOldProc=GetWindowLong(_SCREEN.HWnd, -4)
&& GWL_WNDPROC

BINDEVENT(_VFP.hWnd, 0x031A, THIS, ;
  "HandleThemeChange") && WM_THEMECHANGED
BINDEVENT(_VFP.hWnd, 0x0015, THIS, ;
  "HandleThemeChange") &&WM_SYSCOLORCHANGE

RETURN
```

The HandleThemeChange method, shown in Listing 19, is called when the user changes the theme or an individual color. First, it ensures that the appropriate Windows code executes; this is the equivalent of issuing DODEFAULT() in a method of a VFP subclass. Then, it simply rereads the user's colors into the object, so that it always holds the current colors.

Listing 19. This method is the delegate for two Windows events that occur when the user changes Windows colors. It re-reads the components of the current color scheme.

```
PROCEDURE HandleThemeChange
* Respond to user's change of theme/scheme

LPARAMETERS hWnd as Integer, Msg as Integer, ;
  wParam as Integer, lParam as Integer
```

```
LOCAL lResult
lResult=0

* Note: for WM_THEMECHANGED, MSDN indicates
* the wParam and lParam
* are reserved so can't use them.

lResult=CallWindowProc(this.nOldProc, hWnd, ;
   msg, wParam, lParam)

This.ReadUserColors()

RETURN lResult
```

To take advantage of this, the base form class instantiates the GetUserColors object in Init, then loads the needed colors into the form. It then binds the GetUserColors.ReadUserColors method to the form's GetUserColors method. Listing 20 shows the part of frmBase.Init that sets things up.

Listing 20. The base form class sets things up so that the user's colors are available to the form, and get updated when the user changes the Windows colors.

```
* Load color object
This.oColors = NEWOBJECT("GetUserColors", ;
                         "GetUserColors.PRG")
This.GetUserColors()

* Bind to color changes
BINDEVENT(This.oColors, "ReadUserColors", ;
          This, "GetUserColors", 1)
```

At present, I'm only using three of the color scheme colors directly, so the form class's GetUserColors method (shown in Listing 21) is fairly simple.

Listing 21. The GetUserColors form method fires whenever colors are re-read from the Registry.

```
* Load colors from user's current theme/scheme
This.nDisabledForeColor = ;
  This.oColors.GetAColor("APPWORKSPACE")
This.nDisabledBackColor = ;
  This.oColors.GetAColor("WINDOW")
This.nHighlightColor = ;
  This.oColors.GetAColor("HIGHLIGHT")
```

The final piece of this scheme is a way to update the actual colors used by various controls when the user changes colors. That's handled by subclasses of the base control classes. For example, the lblHighlight class is used to show a label colored with the user's specified highlight color, rather than the default text color. It has a custom method, GetHighlightColor, shown in Listing 22, and called from the label's Init. The method sets the ForeColor of the label to the form's stored nHighlightColor. The Init method (shown in Listing 23) also binds changes to that property to the same method. That is, whenever the form's nHighlightColor is changed, GetHighlightColor fires and updates the label's ForeColor.

Listing 22. This code is called by the Init of the lblHighlight class to set the text color for the label and each time the user changes system colors.

```
IF PEMSTATUS(ThisForm, "nHighlightColor", 5)
   This.ForeColor = ThisForm.nHighlightColor
ENDIF
```

Listing 23. This code in the label's Init method ensures that the label's color changes whenever the user changes system colors.

```
DODEFAULT()

IF PEMSTATUS(ThisForm, "nHighlightColor", 5)
   This.SetHighlightColor()
   BINDEVENT(ThisForm, "nHighlightColor", ;
             This, "SetHighlightColor", 1)
ENDIF
```

The Library classes also include edtEnhancedDisabled and txtEnhancedDisabled that work similarly.

To demonstrate, run the Borrower form and choose a borrower. Note the blue message about the borrower's status. Leave the form open and change the color for Selected Items. Come back to the Borrower form and note that the status message now uses your new color.

One warning here. In my testing, sometimes, Windows hadn't completely updated the colors by the time ReadUserColors ran. That is, the colors returned were not always the new colors. In general, I got better results if I waited longer between choosing a new color scheme or theme and clicking the Apply button in the Display Properties dialog.

Give it a try

When I sat down to start writing about BindEvent(), I had no idea how many different ways I was already using it. As I reviewed code I'd written, I found lots of ways that BindEvent() let me meet user expectations.

I hope my examples give you ideas about how to improve your own applications.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of nearly a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL. Her latest collaboration is Making Sense of Sedna and SP2. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.