FoxRockX

Put Event Binding to Work, Part 1

The BindEvent() function lets you make your applications smarter and easier to use.

Tamar E. Granor, Ph.D.

March 2012
Number 25

At first glance, the BindEvent() function may seem unnecessary. After all, why bind to an event when you can just write code in the event's method?

When the BindEvent() function was added in VFP 8, I had a hard time understanding why I should care. Unlike the EventHandler() function that let my code respond to events for other servers, BindEvent() worked only for events fired in VFP code. Why would I need to bind to such events? Why couldn't I just put the necessary code into the corresponding event methods?

Fairly quickly, I realized that BindEvent() would be quite useful when dealing with black-box code that you couldn't modify or subclass (such as third-party tools). But it took a lot longer before I really saw the potential of BindEvent() and started using it extensively.

In the article and the next, I'll show how event binding works, and explore some of the ways it has improved applications I developed.

The downloads for these articles include a (simple) application that demonstrates many of the techniques discussed. The application is designed for a lending library, to handle books being checked out and checked in, tracking members, and maintaining the catalog of books. This application was originally created as a demonstration of a variety of user interface practices; as a result, some of its UI is a little different than standard Windows applications. In particular, it was designed to use bar codes to specify a member or a book, even when no forms are open. To simulate scanned barcodes, type the desired value with an asterisk ("*") on each side. (Some barcode standards, such as Code 39, use an asterisk on each size as a start/stop code.) For example, to specify the member whose barcode is "7160769048", you'd type "*7160769048*" (without the quotes). However, in a barcode field, you can omit the asterisks.

Throughout this series, I'll use the terms "top-level" and "base" interchangeably to refer to the first-level subclasses of the VFP base classes.

BindEvent() background

Before we dig into examples, let's start with a little theory and terminology. The BindEvent() function creates a connection between methods of two objects. Specifically, an event of one object, the *event source*, is handled by a method, the *delegate*, of another object, the *event handler*. The syntax for **BindEvent()** is shown in Listing 1. It says that whenever method cMethod of oEventSource fires, method cDelegateMethod of oEventHandler will also run.

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

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

By default, the delegate method runs first, but you can change that with the nFlags parameter. That is, code in both methods runs, but you can determine the order.

nFlags also controls whether the binding only applies when cMethod fires as an event, or also when cMethod is called programmatically.

It's worth pointing out that VFP has a built-in example of event binding (though it doesn't use BindEvent()). When you set the KeyPreview property of a form to .T., every time a key is pressed in any control on the form, the form's KeyPress method fires, followed by the KeyPress method of the control itself. It's as if you'd issued BindEvent(oControl, "KeyPress", ThisForm, "KeyPress") for every control on the form.

The example Library application uses this ability to handle bar codes. All of the data entry forms are derived from a class (frmBarCodeEnabled) that has KeyPreview set to .T. and includes code in KeyPress to determine whether the data that was just entered is a barcode.

VFP has three additional functions that deal with event binding. **UnbindEvents()** lets you turn off event binding. You can turn off all bindings for a particular object or only a specific binding. Bindings are automatically turned off when either object goes out of scope, so you only need to use UnbindEvents() if you need to remove a binding while both objects are still available.

AEvents() lets you find out what events are currently bound. It's most useful in the delegate method to find out what object and event triggered the method.

The last event binding function is **RaiseEvent()**; it lets you fire an event programmatically. It differs from just calling the event method in that it ensures that delegate methods are called, no matter which parameters you specified in BindEvent().

Binding to Windows events

In VFP 9, you can also bind to Windows events, such as switching applications, creating a file, or changing the color scheme. The syntax in that case is a little different; among other things, it lets you specify the window for which you want to capture the specified Windows event. Listing 2 shows the structure of the call.

Listing 2. When binding to Windows events, the parameters for BindEvent() change.

```
BindEvent( hWnd | 0, nMessage, oEventHandler,
           cDelegateMethod [, nFlags])
```

The official Windows term for these events is "messages." There are dozens of messages you can respond to; each has a unique numeric code. Unfortunately, the list is not in the VFP documentation. As of this writing, I can't find the list in MSDN, but this site seems to have a thorough list (though it doesn't indicate what action fires each message:
http://wiki.winehq.org/List_Of_Windows_Messages.

Pass the window handle (hWnd) of the window whose messages you want to capture, or pass 0 to capture all Windows messages. In my experience, most of the time, passing _VFP.hWnd lets me capture events in VFP windows. The event handler and delegate method parameters are the same as when binding VFP events. Although you can pass it without error, the nFlags parameter is ignored when binding a Windows message.

In the delegate method, if you want the Windows event to occur as usual, you need to include code to pass it on. I'll show the code you need in my next article.

Putting BindEvent() to work

Now that we've covered the basics, let's move on to some examples that show how BindEvent() can improve your applications.

Managing Context Menus

The first place that the real utility of event binding became clear to me was for handling context menus (AKA right-click menus). Although you can manage these at the control level, I generally find that I want to do so at the form level. That is, I want to use a single form method to evaluate the current situation and populate and show a context menu. I do this by issuing the command in Listing 3 in the Init method of all my top-level control classes.

Listing 3. Putting this command in the Init of all my control classes gives me central handling of right-clicks.

```
BINDEVENT(This, "RightClick", ;
          ThisForm, "RightClick")
```

It's reasonable to ask why this is better than putting ThisForm.RightClick in the RightClick method of each top-level control class. The main reason is that with event binding, I can find out in the form's RightClick method which control was right-clicked; I don't have to pass the control as a parameter and remember to receive the parameter in the form's RightClick.

As for building the shortcut menus themselves, my approach is based on one that Doug Hennig published in the September, 1997 issue of FoxTalk. My base form class's RightClick method calls a custom ShowMenu method. That method determines the caller, creates a popup menu, calls a custom method (named ShortcutMenu, it's abstract in the base form class) that fills the pop-up based on who called it and other factors, and activates the popup. The code I use is shown in Listing 4.

Listing 4. The custom ShowMenu method of my base form class sets up and displays a context menu.

```
LOCAL aEventInfo[1], oObject

* Find out who called
IF AEVENTS(aEventInfo, 0) = 0
   * Called from form
   oObject = This
ELSE
   oObject = aEventInfo[1]
ENDIF

* Define menu
RELEASE POPUPS ShortCut
DEFINE POPUP ShortCut ;
   FROM MROW(), MCOL() SHORTCUT
ON SELECTION POPUP ShortCut ;
   WAIT WINDOW "Under construction.";
   NOWAIT

* Populate menu
This.ShortcutMenu(m.oObject)

* Activate menu
IF CNTBAR("ShortCut") > 0
   ACTIVATE POPUP Shortcut
ENDIF

RELEASE POPUPS ShortCut
RETURN
```

ShowMenu uses AEvents() to identify the control on which the user right-clicked; when you pass 0 as the second parameter to AEvents(), it fills the specified array (its first parameter) with information about the binding that led to the current routine; the first element of the array is the event source. If the function returns 0, it means that this code wasn't triggered by a bound event; in that case, we know the user right-clicked on the form itself.

For a particular form, all I have to do is put code in the ShortcutMenu method to create the appropriate menu bars, based on the object it receives as a parameter. Listing 5 shows the code in the ShortcutMenu method of the CheckOut form in the Library application. Figure 1 shows the context menu when there's a member displayed in the form and you click anywhere except over a book in the grid; Figure 2 shows the context menu when you right-click over a book in the grid.

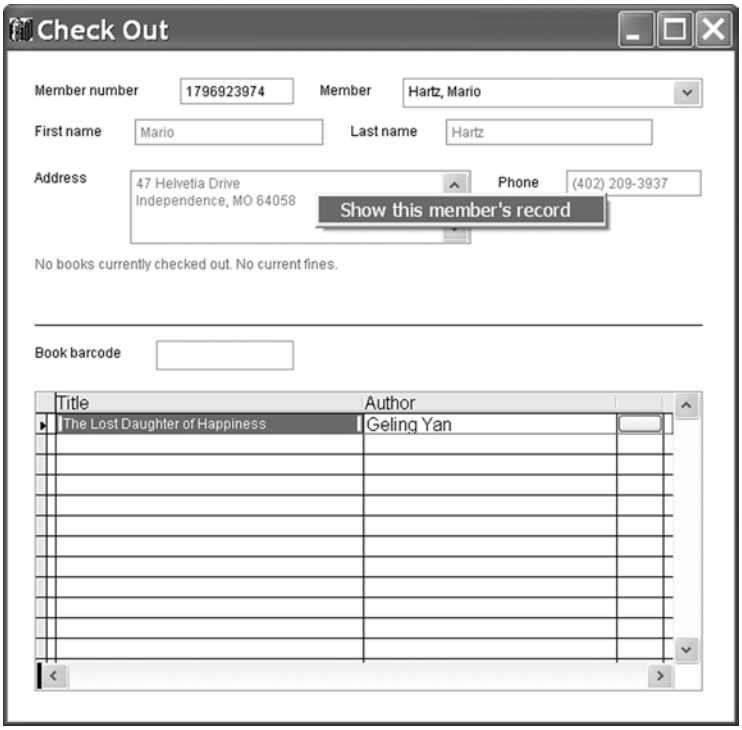

Figure 1. When there's a member showing in the Checkout form, right-clicking in most places offers a single choice: Show this member's record.

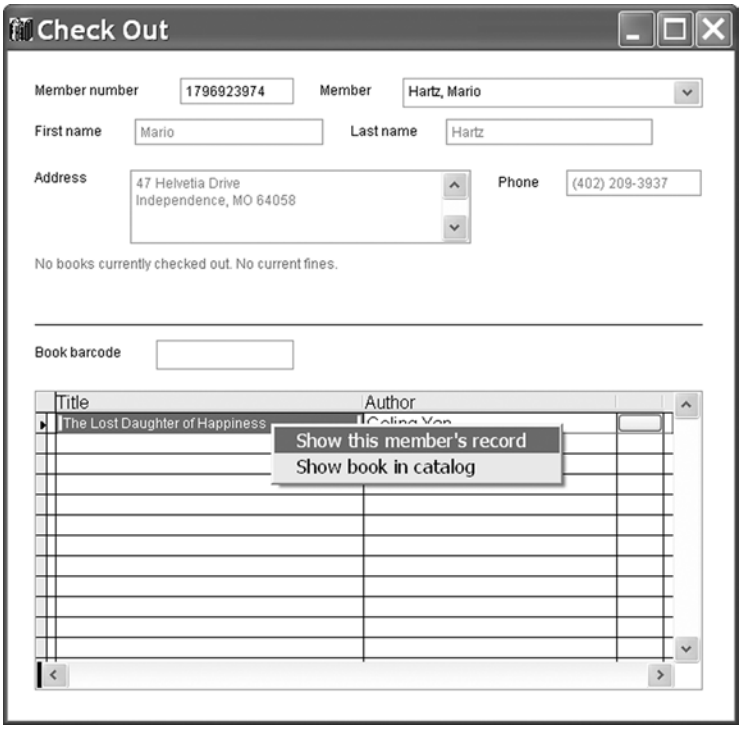

Figure 2. When you right-click over a book in the list to be checked out, you get the option of viewing that book in the catalog.

Listing 5. Binding all right-clicks to the form lets you centralize handling of shortcut menus.

```
LPARAMETERS oObject
* oObject = object actually right-clicked

* Build the shortcut menu for this form

LOCAL nNextBar
nNextBar = 1

* If we have a borrower, provide access to
* borrower form

IF NOT EMPTY(ThisForm.cCurrentMemberNum)
   DEFINE BAR m.nNextBar OF Shortcut ;
      PROMPT "Show this member's record"
   LOCAL cMemberNum
   cMemberNum = ThisForm.cCurrentMemberNum
   ON SELECTION BAR m.nNextBar OF Shortcut ;
      DO FORM Borrowers with "&cMemberNum"
   nNextBar = m.nNextBar + 1
ENDIF


* If we're over a book in the grid, offer to
* open the catalog pointing to it.
* If the control that got us here is the grid
* itself, we're not over a record.
LOCAL lInGrid, oCheckObj, cBookBarCode

lInGrid = .F.
IF NOT INLIST(UPPER(oObject.BaseClass), ;
              "GRID", "FORM")
   oCheckObj = m.oObject
   DO WHILE NOT m.lInGrid AND ;
      NOT ISNULL(oCheckObj.Parent)
      oCheckObj = oCheckObj.Parent
      DO CASE
      CASE UPPER(oCheckObj.BaseClass) = "GRID"
         lInGrid = .T.
      CASE UPPER(oCheckObj.BaseClass) = "FORM"
         * If we get to a form, we're not in a
         * grid. Get out of here.
         EXIT
      ENDCASE
   ENDDO
ENDIF

IF m.lInGrid
   DEFINE BAR m.nNextBar OF Shortcut ;
      PROMPT "Show book in catalog"
   cBookBarCode = CheckOutList.cBarCode
   ON SELECTION BAR m.nNextBar OF Shortcut ;
      DO FORM Catalog ;
         WITH "C", "&cBookBarCode"
ENDIF
RETURN
```

Handling events inside a container

It's not unusual to want all the controls inside a container to delegate behavior to the container. This is especially true when using a container to represent graphical objects as in Figure 3 (which comes from a client application), where many layers of containers are used to represent a physical object. Actions need to take place at the level of the physical object, not the controls from which it's built. For example, each of the boxes under the "Interface n" labels is a container object, containing a label and four additional containers. (In fact, though the figure doesn't show it, the number of

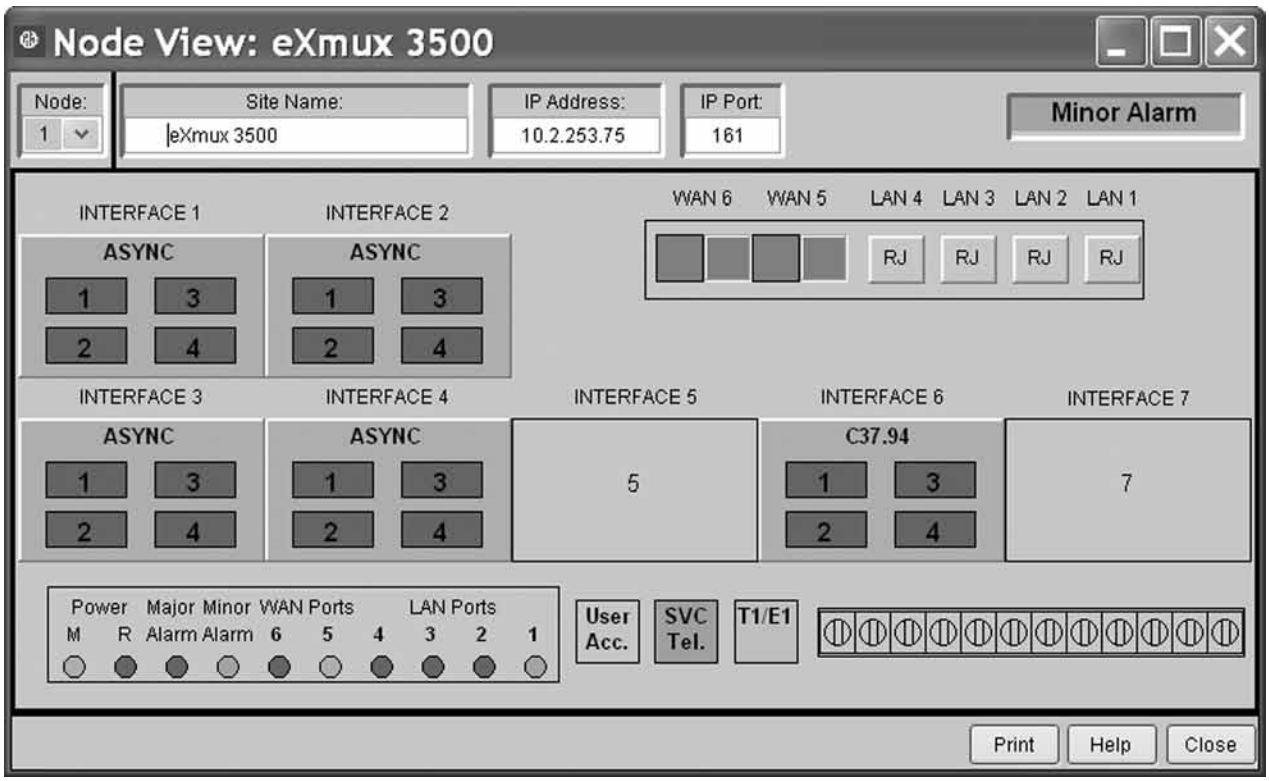

Figure 3. In this form, containers contain other containers, as well as labels, shapes, and other controls. Often, a click or doubleclick needs to be interpreted in the context of the container, not the control that receives the action.

containers inside can be 2, 4 or 8.) Each of those additional containers contains a label. In this application, a double-click anywhere inside one of the large boxes should open another form. Event binding makes this straightforward.

My top-level container class, cntBase, has custom properties, lBindClick, lBindDblClick and lBindMouseDown, as well as custom methods BindClick, BindDblClick and BindMouseDown. The three methods are all quite similar. Listing 6 shows the BindClick method.

Listing 6. The BindClick method of the top-level container class drills down through all the objects in the container and binds their Click methods to the container's Click method. It uses recursion to drill down.

```
LPARAMETERS oContainer

* Bind all contents to start drag
FOR EACH oObject IN oContainer.Objects ;
   FOXOBJECT
   IF PEMSTATUS(oObject, "Click", 5)
      BINDEVENT(oObject, "Click", ;
                This, "Click")
   ENDIF

   IF PEMSTATUS(oObject, "Objects", 5)
      This.BindClick(m.oObject)
   ENDIF
ENDFOR
```

The Init method includes the code in Listing 7, which uses the properties to determine whether to call the methods and set up binding for each of the three events (Click, DblClick, MouseDown).

Listing 7. This code in the top-level container class's Init method binds the Click, DblClick and MouseDown methods of controls inside the container to the container, if the relevant flags are set.

```
IF This.lBindDblClick
   This.BindDblClick(This)
ENDIF

IF This.lBindMouseDown
   This.BindMouseDown(This)
ENDIF

IF This.lBindClick
   This.BindClick(This)
ENDIF
```

With this structure in place, for any given container, all you have to do is set the lBindClick, lBindDblClick and lBindMouseDown properties to determine which actions should propagate from the contained controls to the container itself.

MouseDown is included here because it's the easiest event for triggering drag-and-drop operations. If drag-and-drop is implemented, binding MouseDown allows the user to click on any object within a container to drag the entire container.

In the Library application, the Copy tab of the right pane of the Catalog form has all of its controls in a single container. When a copy of a book has been selected, you can drag from that container to either the CheckIn or CheckOut form to add the book to the check-in or check-out list respectively. I don't want the user to have to worry about where he is on the Copy page, so the container, cntCopyInformation, has lBindMouseDown set to .T., thus MouseDown on any object in the container fires the container's MouseDown method, which contains the code in Listing 8 to start dragging. Figure 4 shows a drag in progress.

Listing 8. This code in the MouseDown event of cntCopyInformation fires when the user clicks anywhere on the container because the MouseDown event of all the contained objects is bound to the container's MouseDown.

```
LPARAMETERS nButton, nShift, nXCoord, nYCoord

IF NOT EMPTY(This.txtBarCode.Value)
   This.OLEDrag(.T.)
ENDIF
```

There's code in both the container and the forms on which you can drop to handle the drag-and-drop operation, but it's not terribly relevant to the binding.

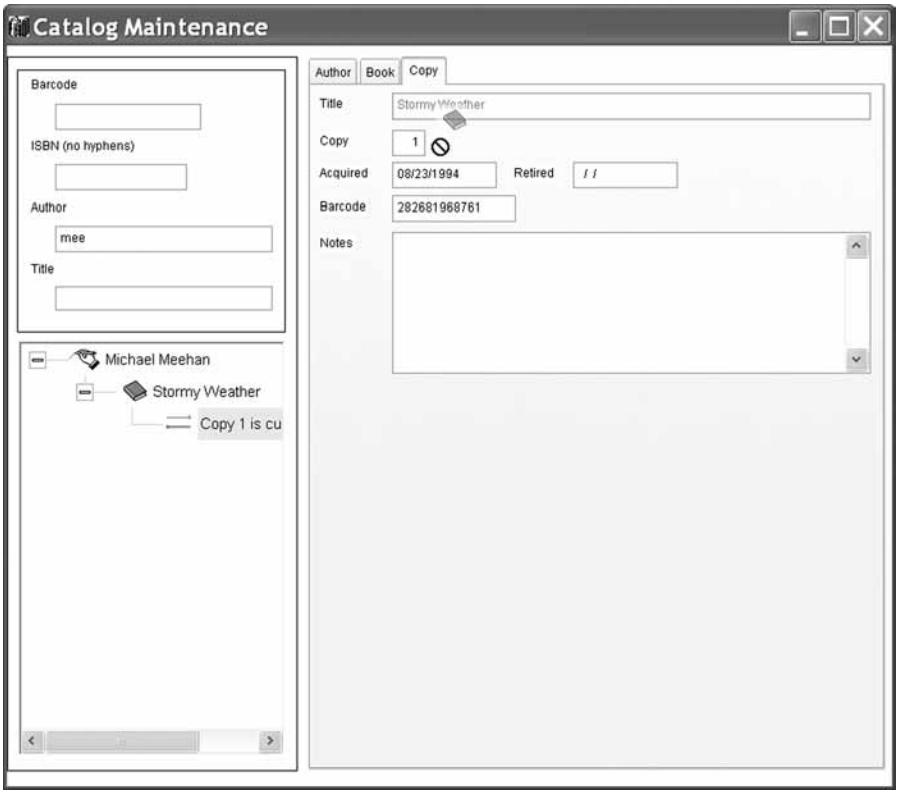

Figure 4. The Copy page of the catalog holds a container identifying this copy of the specified book. You can drag from the container whether you're over the container itself or any of its contained controls.

Tracking user changes

Really well-behaved applications enable and disable controls dynamically, taking user activity into account. One very common feature is to keep the Save button disabled until the user actually changes a record. To do that, of course, you need a way of knowing that the user has changed data.

In the Library application, the top-level class for each control that allows data to change (such as editbox and textbox) has a custom property, lNoteChange. When the property is .T., it indicates that a change to this particular control's value should be noted by the form, and appropriate action taken. In addition, each control has a custom AnyChange method and code in both InteractiveChange and ProgrammaticChange that raises the AnyChange event; that code is shown in Listing 9.

Listing 9. To be able to track user changes closely, add a custom AnyChange method to each control class, and put this code in InteractiveChange and ProgrammaticChange.

```
RAISEEVENT(This,"AnyChange")
```

The top-level form class also has a custom AnyChange method, as well as one called BindControlEvents. BindControlEvents recursively binds the control-level AnyChange method to the form-level AnyChange method, as in Listing 10.

Listing 10. This code binds changes in controls to a form method, so the form can react.

```
* Bind events of controls to events of the
* form as appropriate
LPARAMETERS toControl

LOCAL oControl

FOR EACH oControl IN toControl.Objects
   IF PEMSTATUS(oControl, "lNoteChange", 5) ;
      AND oControl.lNoteChange
      BINDEVENT(oControl, "AnyChange", ;
                This, "AnyChange")
   ENDIF

   IF PEMSTATUS(oControl, "Objects", 5)
      This.BindControlEvents(oControl)
   ENDIF

ENDFOR
```

The base form class has a custom property, lNoteUserChanges, that determines whether BindControlEvents is called in the form's Init method. It's set to .F. in the base class, but to .T. in the frmBizObjAware class that's used for data-aware forms.

The base form class's AnyChange method sets a flag and calls a custom method, UpdateEnabled, to enable and disable controls and menu items appropriately; the code in AnyChange is shown in Listing 11.

Listing 11. This code in the form's AnyChange method responds to user changes.

```
This.lDataChanged = .T.
This.UpdateEnabled(.T.)
```

Finally, in the base form class, UpdateEnabled, shown in Listing 12, ensures that menu Skip For conditions get re-evaluated after a change. It also calls an abstract method you can use in individual forms to deal with specific controls that need to be enabled or disabled.

Listing 12. To make sure menus and toolbars get enabled and disabled appropriately, the form's UpdateEnabled class reactivates menus.

```
LPARAMETERS lForce

* Ensure that skip for conditions get re-
* evaluated.
ACTIVATE MENU MainMenu NOWAIT

IF NOT EMPTY(This.cMenuName)
   ACTIVATE MENU (This.cMenuName) NOWAIT
ENDIF

This.FormUpdateEnabled(m.lForce)
```

The toolbar controls use event binding to piggyback onto the menu to determine whether they should be enabled or disabled. The Init method of tbrBase (the base toolbar class) calls the custom BindToActiveForm method, shown in Listing 13, which binds the custom UpdateEnabled method of the toolbar to the calling form's UpdateEnabled method. So the toolbar has its controls updated whenever the form and menu are updated.

Listing 13. This method, BindToActiveForm, ensures that toolbar controls get updated when form controls and the menu do.

```
LPARAMETERS oForm

* Bind updating of controls to updating on
* active form

DO CASE
CASE VARTYPE(m.oForm) = "O" AND ;
     NOT ISNULL(m.oForm)
   BINDEVENT(m.oForm, "UpdateEnabled", ;
             This, "UpdateEnabled", 1)

CASE VARTYPE(goApp) = "O" AND ;
     MethodExists(goApp.oActiveForm, ;
                  "UpdateEnabled")
   BINDEVENT(goApp.oActiveForm, ;
             "UpdateEnabled", ;
             This, "UpdateEnabled", 1)

ENDCASE
```

The toolbar's UpdateEnabled method relies on the controls themselves to know the condition for enabling or disabling. It just loops through all the controls and calls whatever code the control points to. The method is shown in Listing 14.

Listing 14. The toolbar's UpdateEnabled method tells each control to evaluate its own Skip For condition.

```
* Update the buttons on the toolbar to reflect
* the current state of affairs. Each button
* should have a cSkipFor expression to use for
* this. If not, leave it as is.
* Use TRY-CATCH to avoid problems in special
* cases (such as the app is closing)

FOR EACH oControl IN THIS.Controls
   IF TYPE("oControl.cSkipFor")="C"
      TRY
         oControl.Enabled = ;
            NOT EVALUATE(oControl.cSkipFor)
      CATCH
         * Nothing to do
      ENDTRY
   ENDIF
ENDFOR
```

Because several forms may share the same toolbar, it's not enough to bind the toolbar to the form in the Init. We need to change the binding as the user activates different forms. The application object tracks the active form in the application through another use of BindEvent(); a pair of application methods are called when the active form changes. The Activate method of each form is bound to the application object's SetActiveForm method, while the Deactivate method of each form is bound to the application object's ClearActiveForm method. The form's Init method calls the application object's BindForm method to set this up. BindForm is shown in Listing 15. SetActiveForm and ClearActiveForm simply change an application property to always point to the active form.

Listing 15. The application object's BindForm method lets the application keep track of what form is currently active.

```
PROCEDURE BindForm(oForm)
* Bind a form's events as needed

BINDEVENT(oForm, "Activate", ;
          This, "SetActiveForm", 1)
BINDEVENT(oForm, "Deactivate", ;
          This, "ClearActiveForm")
IF PEMSTATUS(oForm, "GotBarCode",5)
   BINDEVENT(oForm, "GotBarCode", ;
             This, "ProcessBarCode")
ENDIF

RETURN
```

The Init method of tbrBase ties into this model by binding SetActiveForm and ClearActiveForm to the toolbar's BindToActiveForm and UnbindActiveForm methods. The toolbar's Init method is shown in Listing 16, while the UnbindActiveForm method is in Listing 17.

Listing 16. The toolbar class's Init method uses BindEvent() to ensure that the toolbar controls get bound and unbound appropriately.

```
LPARAMETERS oCallingForm
* Reset enable/disable of controls when active
* form changes
IF MethodExists(goApp, "ClearActiveForm")
   * Disconnect from old form
   BINDEVENT(goApp, "ClearActiveForm", ;
             This,"UnbindActiveForm",1)
ENDIF
IF MethodExists(goApp, "SetActiveForm")
   * Connect to new form
   BINDEVENT(goApp, "SetActiveForm", ;
             This,"BindToActiveForm",1)
ENDIF

* Bind to current form
IF VARTYPE(m.oCallingForm) = "O" AND ;
   NOT ISNULL(m.oCallingForm)
   This.BindToActiveForm(m.oCallingForm)
ENDIF
```

Listing 17. This toolbar method, UnbindActiveForm, is called when the form to which toolbar controls are currently bound is deactivated. It releases the bindings, so that the toolbar can, if appropriate, be bound to another form.

```
* Unbind from active form.
IF VARTYPE("goApp") = "O" AND ;
   VARTYPE("goApp.oActiveForm") = "O" AND ;
   PEMSTATUS(goApp.oActiveForm, ;
             "UpdateEnabled", 5)
   UNBINDEVENTS(goApp.oActiveForm, ;
                "UpdateEnabled", ;
                This, "UpdateEnabled")
ENDIF
```

This whole sequence probably seems quite convoluted, but in fact, it provides a virtually invisible mechanism for keeping menu items and toolbar controls properly enabled and disabled.

More to come

In my next article, we'll look at some additional uses for BindEvent, including one that binds to a Windows event.

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.