June, 2006

Advisor Discovery

Use BindEvent() to keep things in synch

By Tamar E. Granor, technical editor

I've been experimenting with BindEvent() since it was added in VFP 8; I've even written about it (see the July, 2005 issue). But I recently found a use for the function that helped me see how it could be an integral part of an application and not just a way to add functionality after the fact.

I wanted buttons on a form to be properly enabled and disabled as soon as a user acts. It's easy to reset the Enable property when the user moves off a control, but I wanted a way to change the property with the first keystroke, so that for example, the Save button is enabled only when the user has actually changed the record, not just because he asked for a new record.

My first attempt set the KeyPreview property of my data entry form class to .T. and then checked in the form's KeyPress for keystrokes that indicated a change of data. It required a complete list of navigation keystrokes, so that pressing those wouldn't change the button status. I got this version working, but it didn't handle using the mouse for data entry or situations like using an ellipsis button to choose a file. The more I looked at getting it totally right, the uglier the problem got.

At this point, I discussed the problem with Doug Hennig, who pointed me at the solution. Before I show the details, let's take a look at event binding in VFP.

BindEvent() Refresher

BindEvent() lets you hook a method to an event. You can specify that when a certain event fires, another method besides the event method should be called. The example most people use to demonstrate the technique looks something like this:

```
PUBLIC oHandler

oHandler = NEWOBJECT("Handler")
BINDEVENT(_SCREEN, "Resize", oHandler, "HandleResize")

DEFINE CLASS Handler AS Custom
```

```
PROCEDURE HandleResize

WAIT WINDOW "Resizing main window" NOWAIT
RETURN
ENDPROC

ENDDEFINE
```

In this code, the Resize event of _SCREEN is bound to the HandleResize event of the oHandler object, which is based on the custom Handler class. After you run this code, when you resize the main VFP window, the WAIT WINDOW appears. (Issuing CLEAR ALL or RELEASE oHandler to turn this behavior off.) This example is included on this month's Professional Resource CD as BindResize.PRG.

Like many simple examples, this code isn't terribly useful, but it demoes well. With a little more work, you can use the same idea to keep an image centered on the main VFP window (or a form) as it's resized.

BindEvent() has four required parameters and one optional parameter. Here's the syntax:

```
BINDEVENT( oEventSource, cEvent,
           oEventHandler, cHandlerMethod [, nFlags])
```

The first two parameters specify the bound event; you supply a reference to the object and the name of the event. The next two parameters specify the event handler (or "delegate"); again, you provide a reference to the object and the name of the method. You can read the function call as a sentence: When the cEvent method of oEventSource fires, call the cHandlerMethod of oEventHandler as well.

The nFlags parameter deals with a couple of variations. It's an additive value that lets you specify several things with a single parameter.

The first issue is whether the bound event or the handler runs first. By default, the handler code runs first. Add 1 to nFlags to run the bound event's method first and then the event handler's method.

The second issue is a little trickier to understand. VFP lets you call an event method directly with code like ThisForm.cmdSave.Click() (though such calls are bad form). You can determine whether code bound to such an event fires on a programmatic call or not. By default, such a call does also run the event handler method; add 2 to nFlags to prevent such calls from executing the event handler method.

The reason for the distinction is that direct calls to an event method like Click don't actually fire the event. They run the code contained in the event method, but not the built-in VFP code associated with the event. For example, when you call a button's Click method, the code there runs, but the button doesn't go down and back up visually.

One final point about BindEvent(), though it's not relevant to this particular problem. You can actually bind to properties as well as events. That is, the cEvent parameter can actually be the name of a property. When you bind to a property, the event handler method fires every time that property's value changes. This is similar to having an Assign method for the property, but without the need to subclass.

In addition to providing another way to respond to VFP's events, the event binding mechanism also provides a way to create and fire custom events. The RaiseEvent() function lets you indicate that an event is firing, even if it's a custom method of a custom object. The syntax is:

```
RAISEEVENT( oEventSource, cEvent [, uParameters ])
```

Events fired by RaiseEvent() call any handler code regardless of the nFlags setting used in BindEvents(). You can also use RaiseEvent() on native events; as with custom methods, handler code is always called. However, even RaiseEvent() doesn't fire built-in VFP code for the native event.

Back to the problem

To solve the problem of properly enabling and disabling buttons, I made a few changes. First, I eliminated the KeyPress-related code entirely.

I added a custom AnyChange method to every control class with a ControlSource property and put this code in Both InteractiveChange and ProgrammaticChange:

```
RaiseEvent(This, "AnyChange")
```

Since there might be controls that aren't involved in the record itself and thus shouldn't determine the status of the buttons, I also added a custom property, lNoteChange, to each of the control classes with AnyChange methods.

Next, I added an AnyChange method to the base form class. Then I added a method called BindControlEvents and put the following code in it:

```
LOCAL oControl

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

So the AnyChange method of every control on the form that has lNoteChange set to .T. fires the form's AnyChange method. The Init method calls BindControlEvents.

With this code in place, any time there's a change to a control on the form, the form's AnyChange method fires. This is essentially a generalization of the mechanism that KeyPreview provides for KeyPress.

In my situation, I put a method call in AnyChange to the form's custom UpdateEnabled method that evaluates the current situation and enables and disables controls.

This month's Professional Resource CD contains a class library (Base.VCX) containing the necessary code for this technique, as well as an example form (Customers.SCX) that demonstrates it. The form is a simple editing form, but the buttons enable and disable based on the current state, as soon as you start typing. Figure 1 shows the form after clicking the New button; Figure 2 shows it after something has been typed.

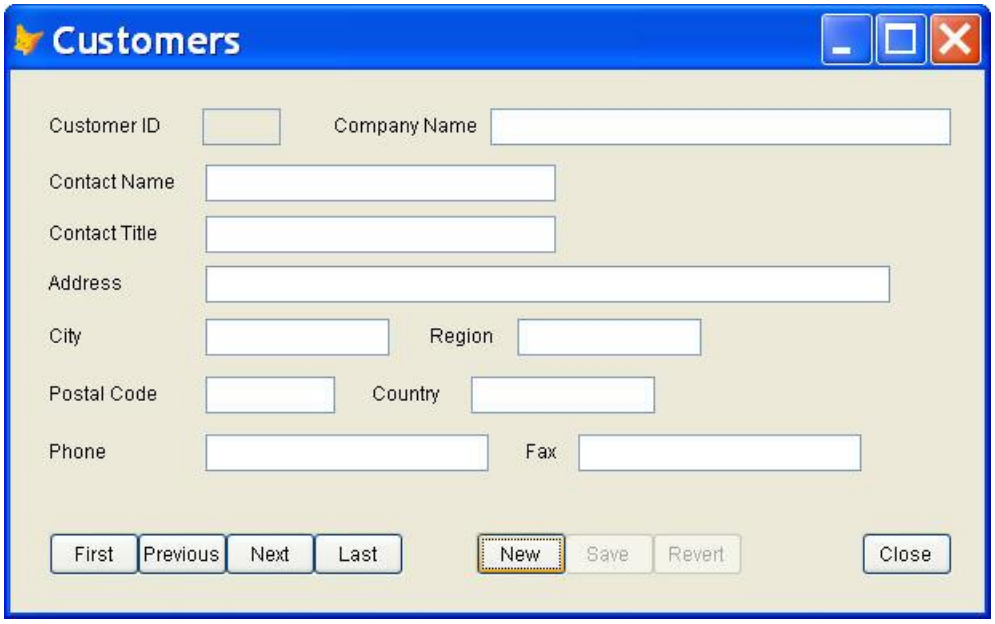

Figure 1. New record—After clicking New, the new record is ready to edit, but there's nothing worth saving, so Save and Revert are disabled.

Figure 2. Changed record—Once there's some data to save, the Save and Revert buttons come alive. In your applications, you might chose to be more picky about what's enough to enable the Save button.

Lots of ideas

Since this experience, I'm seeing more uses for BindEvent(). For example, in a project I'm currently working on, I'm binding the Activate and Deactivate methods of the base form class to a couple of

application object methods, so I can maintain an application-level property pointing to the active form. Then I can bind to changes in this property to handle application-level tasks when the user moves from one form to another. Among the things I'm currently working out is keeping toolbar buttons properly enabled and disabled, based on the active form and its status.