Bind Events for Better Applications

Tamar E. Granor
Tomorrow's Solutions, LLC
Web: www.tomorrowssolutionsllc.com
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

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

In this reprise of a popular session, we'll look at why BindEvents() and its cousins, the Access and Assign methods, are so valuable. Using examples drawn from real applications, we'll see how event binding lets you do things you couldn't otherwise do, and simplifies code for other tasks. Examples include tracking user changes on a form, monitoring user activity, dynamic

tooltips, and more. We'll also talk about the difficulties involved in debugging code that uses event binding.

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.

On the other hand, with the introduction of Access and Assign methods in VFP 6, I immediately saw that they gave us the opportunity to create our own custom events, and quickly started finding uses for them. As my comfort level with BindEvent() grew, I saw that the two capabilities were really related. Both Access and Assign methods and BindEvent give us more control over what happens as users work with our applications. In addition, they let us build more capabilities into our class libraries, so we can just use them in new forms and applications.

In this paper, I'll show how event binding and Access/Assign methods work, and explore some of the ways these abilities have improved applications I developed. In addition, I'll take a brief look at the challenges they offer for debugging.

The materials for this session 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 paper, 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* or *delegate method*, 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.

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(This, "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 code in KeyPress to determine whether the data that was just entered is a barcode.

There are 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; there are some examples later in this paper.

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 or changing the color scheme. The syntax in that case is a little different; among other things, it lets you specify for which window 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. The complete list doesn't seem to be in MSDN either, but this site seems to have a thorough list: 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. My experience is that usually, passing _VFP.hWnd gives me what I want. 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. The code you need is shown in "Updating colors when the color theme changes" later in this document.

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. In addition, the event binding can't be overridden by custom code in the control's RightClick method (although, of course, it can be by custom code in the Init method, but most developers are careful to issue DoDefault() in the Init method).

As for building the shortcut menus themselves, my approach is based on one that Doug Hennig published in FoxTalk a long time ago (the September, 1997 issue). 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 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 ShorcutMenu 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.

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
```

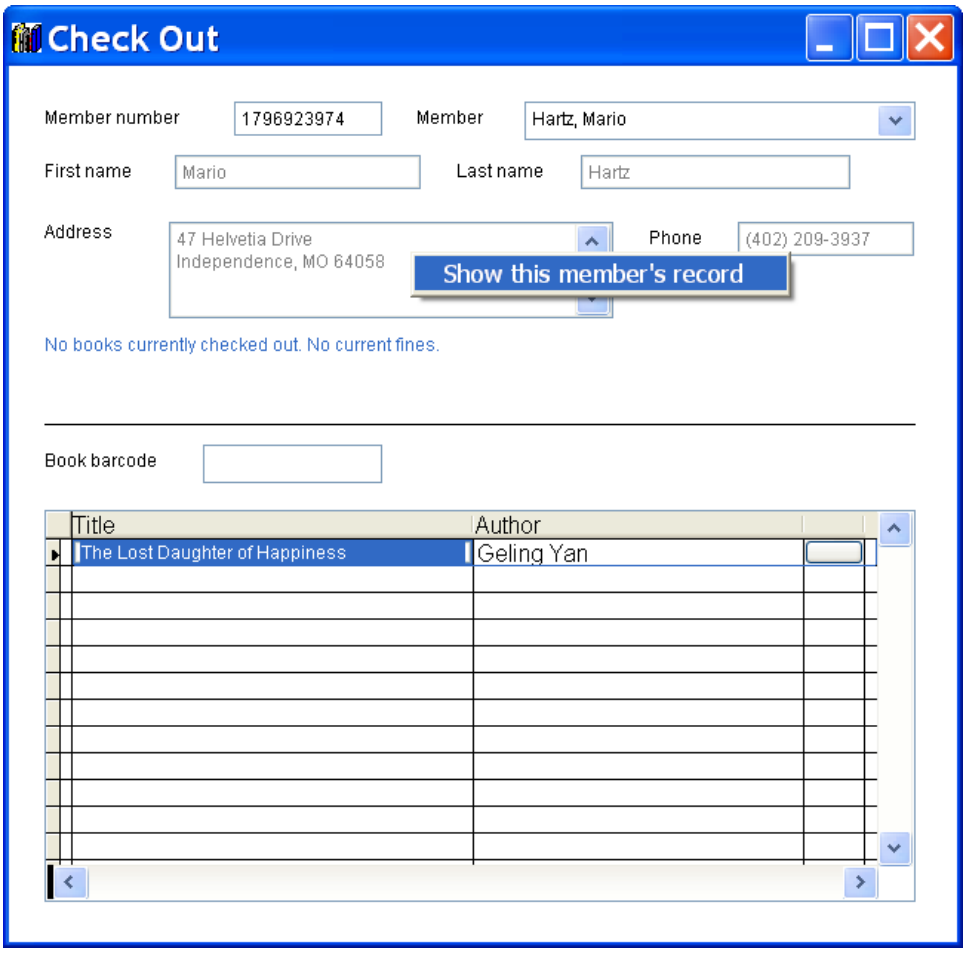

Figure 1. When there's a member showing in the Checkout form, right-clicking in most places offers a single choice: Show this member's record.

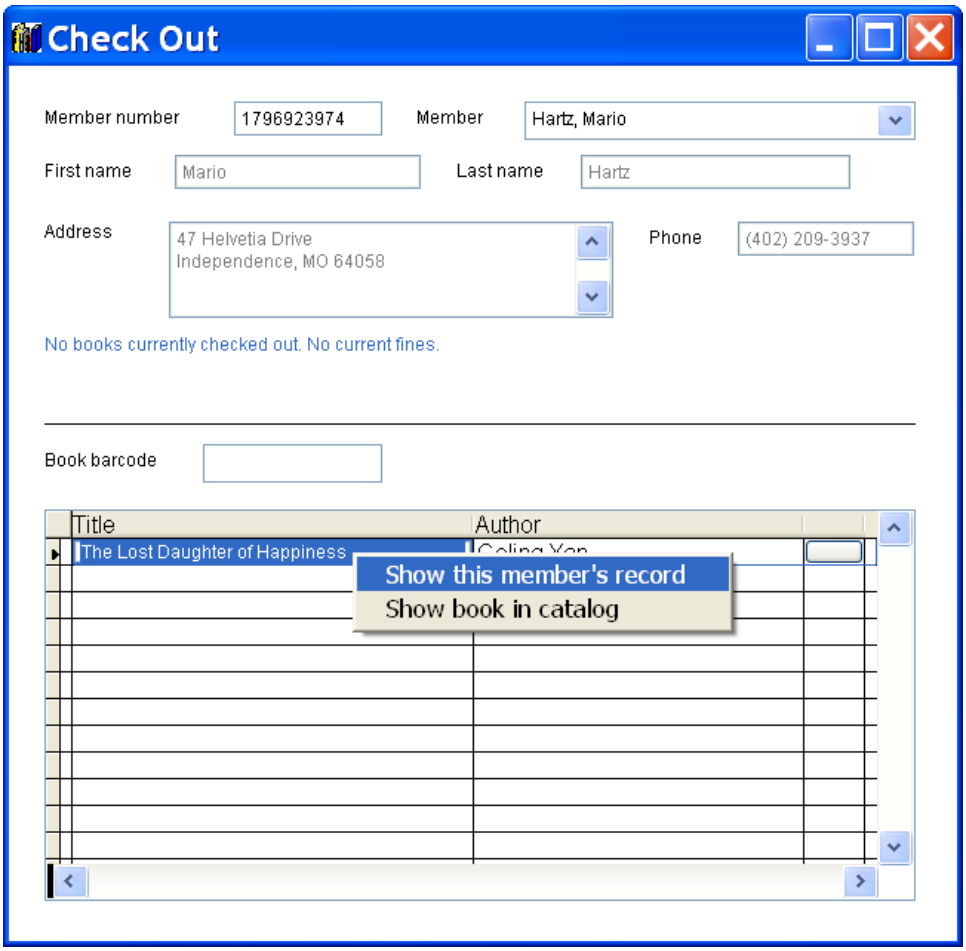

Figure 2. When you right-click over a book in the list to be checked out, you get the option of viewing that book in the catalog.

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 green 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 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 green boxes should open another form. Event binding makes this straightforward.

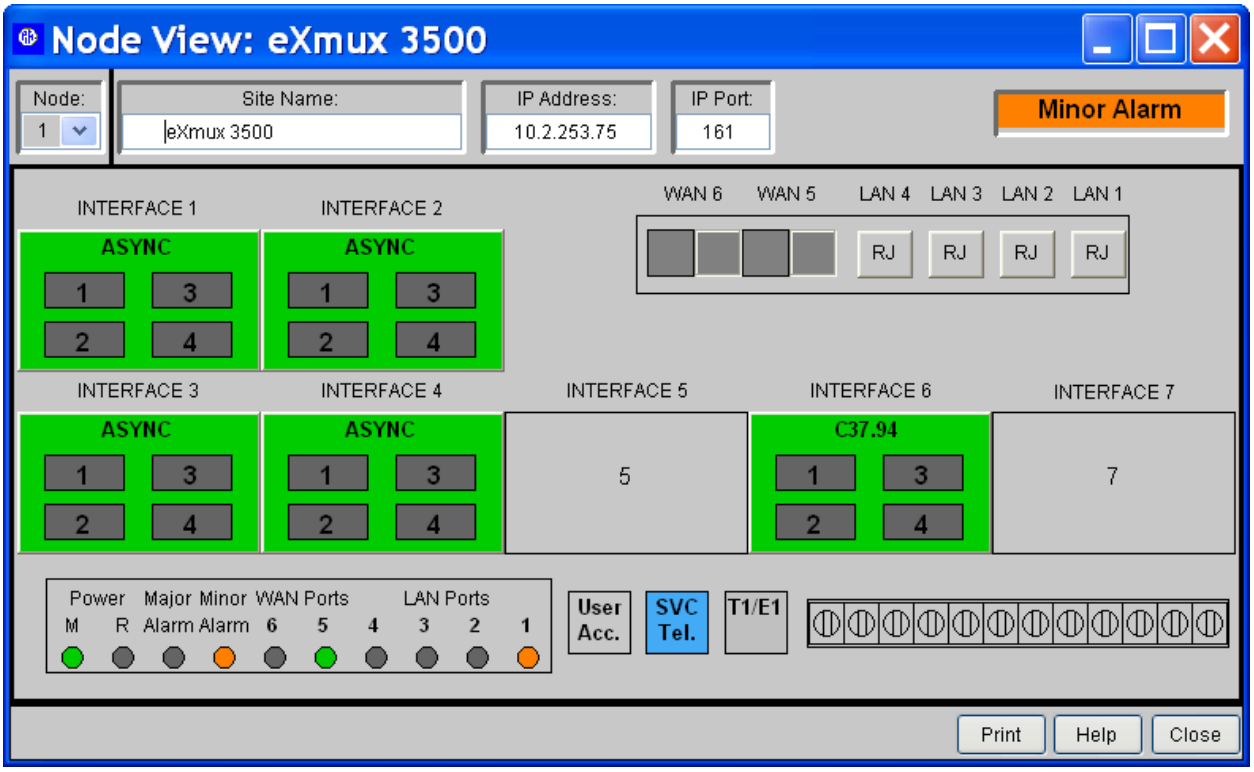

Figure 3. In this form, containers contain other containers, as well as labels, shapes, and other controls. Often, a click or doubleclick needs to be interpreted in the context of the container, not the control that receives the action.

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. (A useful improvement would be to create a single BindMethod method that accepts the method to be bound as a parameter, and replace the individual BindClick, etc., methods.)

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.

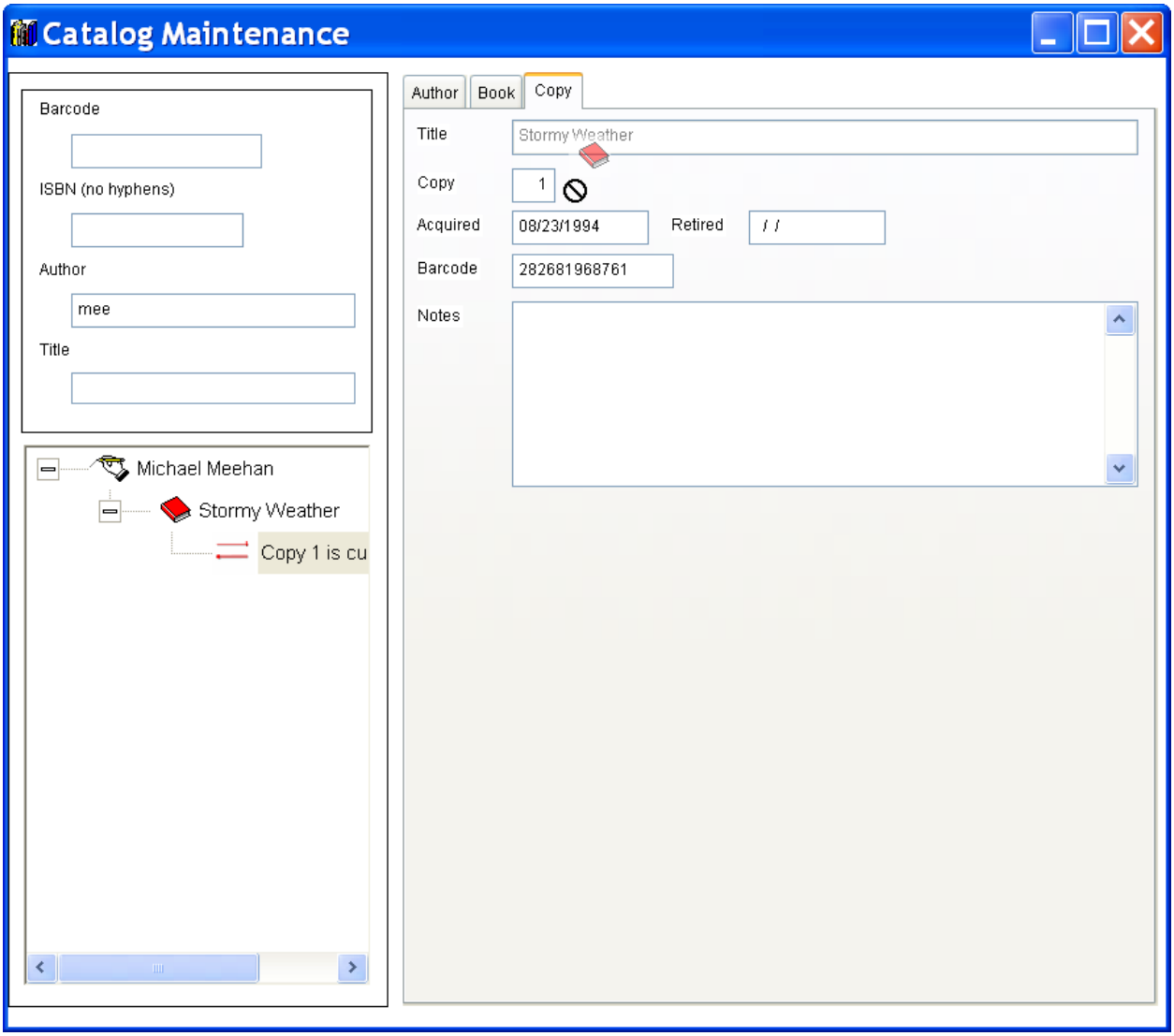

Figure 4. The Copy page of the catalog holds a container identifying this copy of the specified book. You can drag from the container whether you're over the container itself or any of its contained controls.

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 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 my 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**, just ensures that menu Skip For conditions get re-evaluated after a change. It 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. Note that the BindEvent call includes a value of 1 for the nFlags parameter, indicating that the delegate method (the toolbar's UpdateEnabled method) should be called after the source method (the form's or application object's UpdateEneabled method) runs. So the toolbar has its controls updated whenever the form and menu are updated, but afterward.

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. Again here, the delegate method (BindActiveForm or UnbindActiveForm) runs after the source method (SetActiveForm or ClearActiveForm). 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. All that's required when create forms is the custom form-level code in FormUpdateEnabled. For menus, setting the Skip For condition does the trick, while for toolbars, all you have to do is set the custom cSkipFor property of each control.

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. Note that this is, in a sense, the opposite of the bindings in some of the earlier examples. There, we pushed an event from a control up to its container or the containing form. Here, we're pushing it downward to have a particular control react when something happens to the form that contains it.

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, 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 18**.

Listing 18. 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 19**.

Listing 19. 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 20**, populates the oForm property, and the BeforeDock method, shown in **Listing 21**, 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 20. The toolbar holds a pointer to the containing form, stored in the Init method.

```
DODEFAULT()

This.oForm = _VFP.ActiveForm
```

Listing 21. 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 22** shows the Activate code.

Listing 22. 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 another example 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 materials for this session, 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 anytime 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 23**.

Listing 23. 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 24**, simply instantiates the timer and stores a reference in the oActivityTimer property. TRY-CATCH is used in case there's a problem.

Listing 24. 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 25**.

Listing 25. The lBindUserActions property determines whether to track user activity in the form.

```
IF This.lBindUserActions
   This.BindUserActions()
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 26**, calls a method of the application object. The IF statement ensures that we can run forms stand-alone for testing (that is, when the application object hasn't been instantiated). Note that we pass an object reference to the calling form to the application's BindUserActions method.

Listing 26. The BindUserActions method of the "base" form class calls the application object's BindUserActions to do the binding.

```
* 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 27**. It passes along the reference to the form.

Listing 27. 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 28** 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 28. The activity timer's BindUserActionInObject method binds the KeyPress and MouseMove events of the specified object and (recursively) 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. 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 29**.

Listing 29. 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 source 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. (See the next major section of this paper for details on how assign methods work.) 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 30**, 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 30. 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 in an orderly way.

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 31**.

Listing 31. 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 32**. (The GetMember method retrieves the member data and puts it in a cursor.)

Listing 32. 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 33**. 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 33. 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 in 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. (See the note at the end of this section about Windows 10 and colors.)

If the user changes the color scheme while an application is running, I want my application to follow her lead. To do this, I bind to the Windows HandleThemeChanged message.

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 in this paper; it's fairly straightforward.

The BindColorChanges method, called from the Init method, sets up the binding; it's shown in **Listing 34**. The two API function declarations and the call that follows them store information about the Windows procedure to call when handling the event.

Listing 34. 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 35**, 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 35. 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. (You could also instantiate GetUserColors just once in the Init code for the application object and store the reference there.) It then binds the GetUserColors.ReadUserColors method to the form's GetUserColors method. **Listing 36** shows the part of frmBase.Init that sets things up.

Listing 36. 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 37**) is fairly simple.

Listing 37. 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 38**, and called from the label's Init. The method sets the ForeColor of the label to the form's stored nHighlightColor. The Init method 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 38. This code is called by the Init of the lblHighlight class to set the text color for the label and ensure that it gets updated each time the user changes system colors.

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

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

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.

Windows 10 appears to significantly reduce user color over colors. While the user can set a single accent color, the only way to actually change all the colors that used to be readily available is by choosing a new theme. Add to that that users cannot change the colors within themes unless they pick a High-Contrast theme. So while this code works, it’s far less useless in Windows 10.

Access and Assign methods

By now, you can see that event binding offers lots of opportunities to make applications more responsive. An older technique provides some additional ways to make your applications behave as users expect. Access and Assign methods were introduced in VFP 6, and essentially allow you to define your own events in an application.

Any property can have either an Access method, an Assign method or both. The Access method fires whenever the property is referenced, while the Assign method fires whenever the property changes. Each of them allows you to change the value of the property as well as take other actions.

The Assign method receives the new value (the one you're assigning) as a parameter. In the code, you can assign any value you want to the property, not just the one it receives as a parameter. So, among other things, you can use the Assign method to make a property read-only or read-only under some circumstances. Most often, though, I use Assign methods to ensure that certain things happen when the value of the property changes. In other words, an Assign method becomes an event upon which I can take action.

In the Access method, you can control what value the triggering code sees as the value of the property. The value you return from the Access method is used by the triggering code, even if the actual value stored in the property is different. For example, you could translate the value into a different language based on an application or system setting, or convert it from a convenient internal storage mechanism to a friendlier user format (say, numeric to character). Most often, I use Access methods to ensure that a property is up-to-date based on current settings. That is, an Access method lets me avoid having to ensure that a property gets updated every time the things it depends on change. I do the update when I actually need the value.

Adding Access and Assign methods to a property is somewhat different from adding custom methods to an object. In the Property Sheet, right-click on the property in question, and choose Edit Property/Method, shown in **Figure 5**.

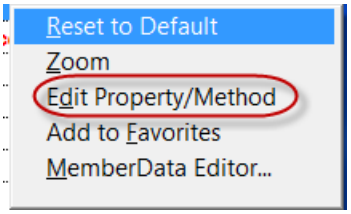

Figure 5. Use the Edit Property/Method dialog to add Access and Assign methods.

In the Edit Property/Method dialog, there are checkboxes for Access Method and Assign Method; check one or both. The updated Add Property dialog that comes with VFP 9 (and has an improved version in VFPX), as well as the improved Edit Property/Method dialog (shown in **Figure 6**) and Thor's PEM Editor all make it easy to add Access and Assign methods when adding a new property or editing it.

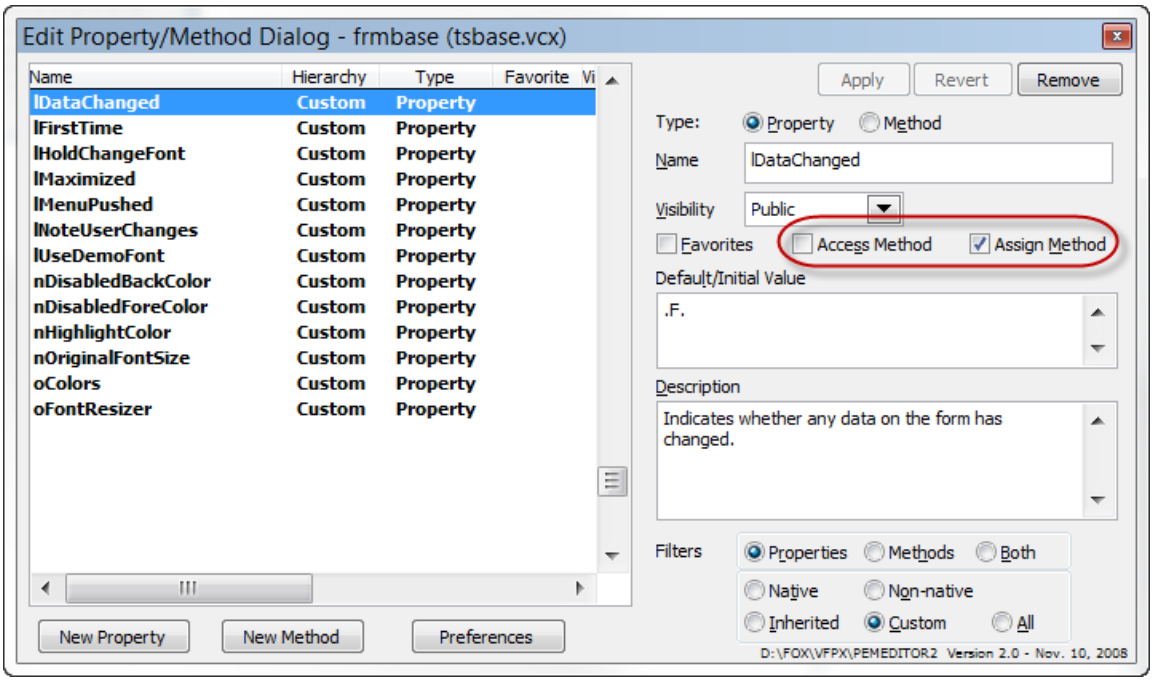

Figure 6. The enhanced Edit Property/Method dialog, available from VFPX, makes it easy to add Access and Assign methods.

Access methods automatically contain a single line of code that returns the property's value. Assign methods automatically receive the new value as a parameter and contain a single line assigning that value to the property. Once they exist, you can edit the code as you desire.

Using Access methods

As indicated above, the principal use I find for Access methods is to update a property when it's needed. However, they also offer a workaround for a VFP bug.

Just-in-time updating

When the value of a property is based on other items that can change as the application runs, an Access method lets us look up or compute the value when it's needed. Doing so serves several purposes. First, we don't have to insert calls (or use Assign methods) to update the property every time one of its components changes. Second, it's a kind of encapsulation. Only the property has to know how to find its own value. If the rules for generating the value change, the only thing we have to change is the Access method (or a custom method it calls).

For example, in **Figure 3** (much earlier in this paper), the orange block that says "Minor status" indicates the overall status of the displayed node (a node here is a utility substation); the determination of a node's status involves checking a number of conditions. The application monitors the corresponding hardware and when there are changes, updates the form. The node's status is stored in a property called cStatus; cStatus's Access method calls another method that computes the current status and returns the computed

value. The form then displays that value and colors the background around it appropriately (red for "Major Alarm", orange for "Minor Alarm", green for "Normal").

In the same application, a number of forms indicate the last time complete data was read from the hardware; **Figure 7** shows an example. There's a lot of data to read, so some data is only read either by user request or when the form that displays the data item is open. Rather than updating the form-level timestamp each time an item is read, an Access method loops through all the relevant items and updates the form-level timestamp when the form is first displayed and each time it's refreshed.

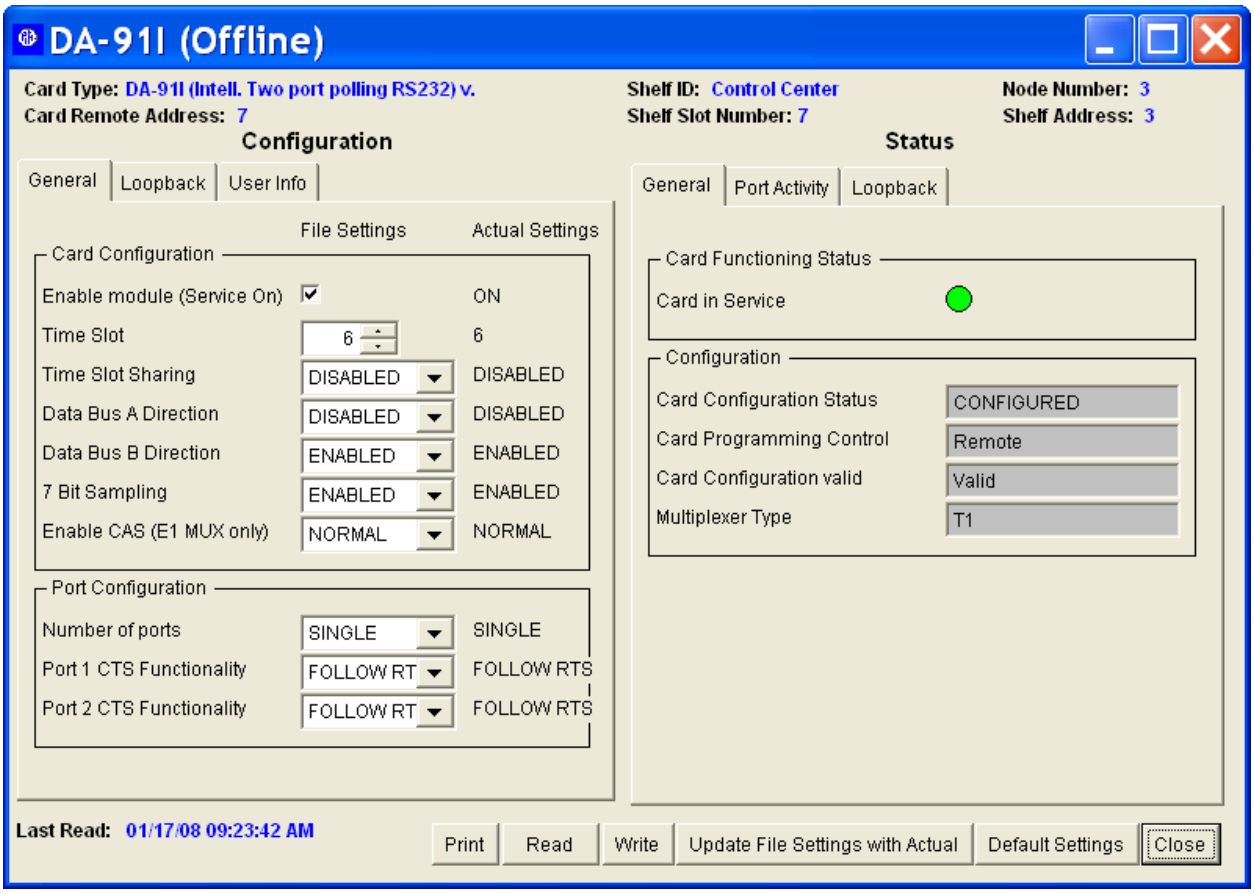

Figure 7. In this form, the last read value at the bottom indicates the oldest timestamp for any of the settings shown on the form. Rather than update each time a setting changes, an Access method finds the right value when it's needed.

The data-handling form class in the Library application, frmBizObjAware, uses this strategy to determine the first control on the form in tab order. The class has a custom property, cFirstControl; if the developer of a particular form sets the property at design-time, the specified control is treated as the first in tab order. However, that's an easy thing for a developer to forget, so an access method for the property cycles through the controls on the form and sets the property based on the TabOrder of the various controls. **Listing 39** shows the code in cFirstControl_Access.

Listing 39. This code ensures that a first control is specified on the form, even if the user forgets to indicate which control comes first.

```
* Handle the possibility that no first control has been set
LOCAL oControl, oLowControl, nLowTab

IF EMPTY(THIS.cFirstControl)
   * Figure it out
   nLowTab = 1000000
   FOR EACH oControl IN THISFORM.OBJECTS
      IF PEMSTATUS(oControl, "TabIndex", 5) AND PEMSTATUS(oControl, "SetFocus", 5)
         IF oControl.TABINDEX < nLowTab
            oLowControl = oControl
            nLowTab = oControl.TABINDEX
         ENDIF
      ENDIF
   ENDFOR

   THIS.cFirstControl = oLowControl.NAME
ENDIF

RETURN THIS.cFirstControl
```

This code is used, for example, in the New button. After adding a new record and appropriately enabling and disabling controls, focus is set to the first control on the form. **Listing 40** shows the code in the form class's New method:

Listing 40. This code in frmBizObjAware.New sets focus to the form's designated first control. The access method in **Listing 39** is called implicitly to make sure that cFirstControl has a non-empty value.

```
* Add a record in this form
LOCAL lReturn

IF MethodExists(This.oBizObj, "New")

   lReturn = This.BeforeNew()
   IF lReturn
      lReturn = This.oBizObj.New()
      IF lReturn
         This.lNewRecord = .T.
         This.lDataChanged = .F.
         lReturn = This.AfterNew()

         oFirstControl = EVALUATE("This." + This.cFirstControl)
         oFirstControl.SetFocus()
      ENDIF
   ENDIF

ENDIF

RETURN lReturn
```

Dynamic tooltips

In the form shown in **Figure 3**, we need to show tooltips for each of the numbered boxes, which represent ports; the content of the tooltip is determined by the current status of the port. An Access method provides an easy way to do what's needed; just build and return the appropriate string in the ToolTipText_Access method.

In the Catalog form of the Library application, it would be useful to be able to show all the details about a book's current status in a tooltip on the Copy page. (In fact, we might actually want to put the information on that page, but for the purpose of demonstrating this technique, we'll use a tooltip.) The information on that page is displayed in a container whose class is cntCopyInformation; **Listing 41** shows the ToolTipText_Access method for that container.

Listing 41. The code in the ToolTipText_Access method calls a form method to build the tooltip.

```
* Check whether the book is out and build a tooltip with that info
RETURN ThisForm.GetBookDetail(ThisForm.cBarCode)
```

The form's GetBookDetail method looks up the book currently displayed and builds the appropriate string to display. **Figure 8** shows an example.

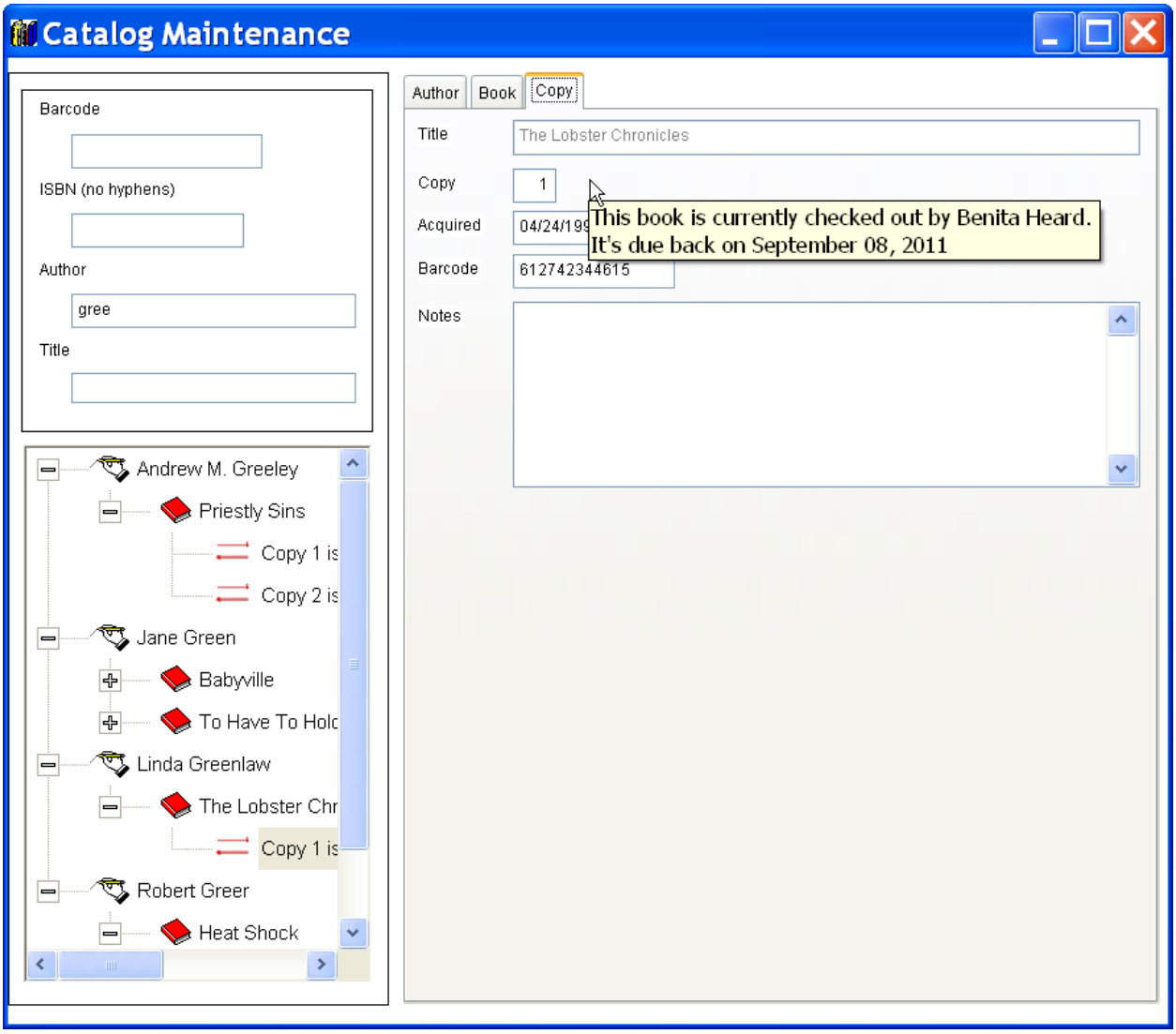

Figure 8. The tooltip for the container on the Copy page is built on the fly using the Access method of ToolTipText.

Delegating tooltips for contained objects

The example in **Figure 8** also demonstrates another use for Access methods. In a container like the one shown, we may want the same tooltip for all controls. One easy way to do that is to have the Access method for a control's ToolTipText return the parent's ToolTipText, along the lines of **Listing 42**.

Listing 42. To give a container and its contents the same tooltip, put code like this in the contained controls' ToolTipText_Access method.

```
RETURN This.Parent.ToolTipText
```

A little code in the base classes lets us set this up across the board, so we can turn it on for a given container by setting a single property. First, cntBase (the base container class) has a custom property, lBindToolTip. (Using the word "bind" here is a little misleading since it

implies BindEvents, but in fact, we are really binding the container control's ToolTipText to the parent.) Then, the ToolTipText_Access method for each control class that has a ToolTipText method contains the code in **Listing 43**. Then, all you have to do to ensure that everything in a container shows the same tooltip is set the container's lBindToolTip property to .T.

Listing 43. Put this code in the ToolTipText_Access method of each base control class.

```
* Check whether we're supposed to be passing tooltips up
LOCAL cTip

IF PEMSTATUS(This.Parent, "lBindToolTip", 5) AND This.Parent.lBindToolTip
   cTip = This.Parent.ToolTipText
ELSE
   cTip = This.ToolTipText
ENDIF

RETURN m.cTip
```

Grid component tooltips

A similar approach allows you to work around a bug in VFP 9 SP2; the contained objects of a grid don't show their own tooltips, but the tooltip of the grid. An Access method for the grid's ToolTipText property lets you drill down into the contained objects and use their ToolTipText instead.

In the library application, the top-level grid class, grdBase, has the code in **Listing 44** in its ToolTipText_Access method. This code actually combines the technique in the previous section for propagating tooltips down from containers with the ability to give controls inside a grid their own tips. It checks first whether the parent of the grid has the lBindToolTip property set to .T. If so, it uses the parent's tip; if not, it drills into the grid and allows the controls inside to provide tips.

Listing 44. Use ToolTipText_Access to work around the VFP 9 SP2 bug regarding tooltips in grids.

```
* Check whether we're supposed to be passing tooltips up
LOCAL cTip

IF PEMSTATUS(This.Parent, "lBindToolTip", 5) AND This.Parent.lBindToolTip
   cTip = This.Parent.ToolTipText
ELSE
   * Let components have their own tooltips.
   * Look up the tooltip for the object currently under the mouse.
   LOCAL aMousePos[1], oColumn, oControl

   cTip = ""

   IF AMOUSEOBJ(aMousePos) > 0
      oColumn = aMousePos[1]
      IF NOT ISNULL(m.oColumn) AND UPPER(oColumn.BaseClass) = "COLUMN"
         * First, grab column-level tip in case we don't find something below
         cToolTip = oColumn.ToolTipText
```

```
         * Now, look for the right control.
          oControl = EVALUATE("oColumn." + oColumn.CurrentControl)
          IF NOT EMPTY(oControl.ToolTipText)
             cTip = oControl.ToolTipText
          ENDIF
       ENDIF
   ENDIF
ENDIF

RETURN m.cTip
```

In the application, the CheckOut form has a tooltip for the delete button in the third column; it's shown in **Figure 9**.

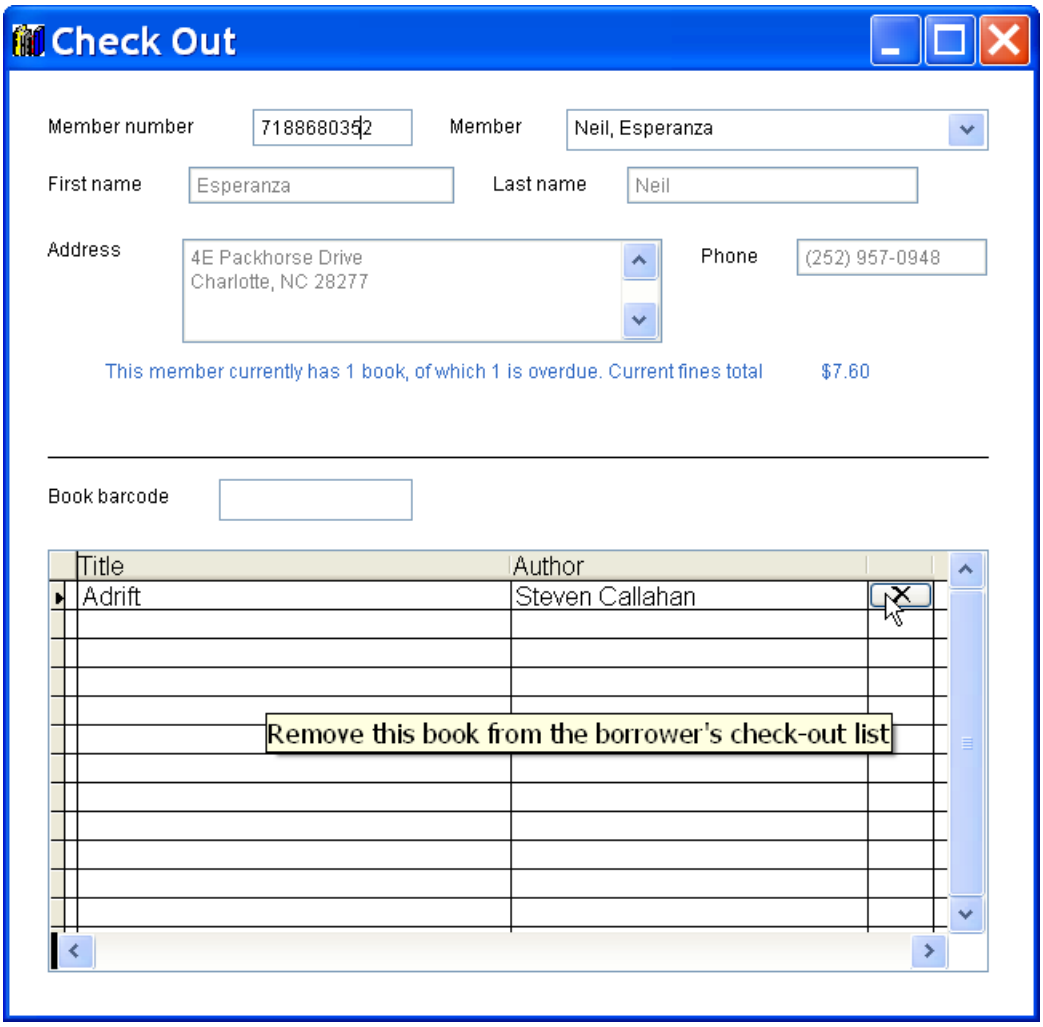

Figure 9. The tooltip shown comes from the button, not the grid. The ToolTipText_Access method finds the right tip to display.

Using Assign methods

I've found more ways to use Assign methods than Access methods. Although an Assign method lets you change the value assigned, I rarely use that capability. More often, my Assign method code lets me ensure that additional things happen when the value is saved. Often, the goal is encapsulation; by using an Assign method, I keep the code that changes a property from having to know what a change in that property affects.

Updating a timestamp

In the application from which **Figure 7** is drawn, we track the current value (the "Actual Value") of each of the items shown in the form. These values are read from actual hardware and we want to know not just the value, but when it was read (among other things, to show the last read date on the form, as in **Figure 7**). A business object contains the information for a single item; there's a collection of such items. The business object has a cActual property for the current value and a tLastRead property for the timestamp. An Assign method for cActual updates tLastRead, as in **Listing 45**.

Listing 45. The Assign method for a property that tracks values read from hardware updates the timestamp for the value.

```
LPARAMETERS tuNewValue

IF NOT (ALLTRIM(THIS.cActual) == ALLTRIM(m.tuNewValue))
   THIS.cActual = m.tuNewValue

   * Set last read time.
   THIS.tLastRead = DATETIME()
ENDIF
```

Set a "dirty" flag

The code in **Listing 45** is actually only part of what we do when we read a new hardware value. We need a way to know whether the data has changed since it was last stored, that is, a "dirty" flag. Assign methods provide that, as well. We have an application property, lIsDirty. Whenever we save data or open a new data file, we clear that property. The Assign method (from **Listing 45**) includes the additional line of code shown in **Listing 46** inside the IF; we use the same code in the Assign methods of all properties where a change indicates that the data is now different than it was at the last save.

Listing 46. One line of code in an Assign method (plus a little application-level code) lets you keep track of whether data has changed since it was last saved.

```
goApp.lIsDirty = .T.
```

In the Library application, we already have a lot of the framework in place for keeping an application-wide "dirty" flag. Each form has the lDataChanged property. We can use those properties to determine whether there's any unsaved data in the application. To do so, we add an Assign method to lDataChanged in frmBase. That method calls an application object method to update the application-wide "dirty" flag.

Just to demonstrate another possibility, lDataChanged_Assign also updates the form caption so that whenever there's unsaved data, it includes an asterisk, just as VFP does. The method is shown in **Listing 47**.

Listing 47. This code in the Assign method of the form's lDataChanged property aids in keeping an application-wide "dirty" flag, and has each form's Caption indicate whether it currently has unsaved changes.

```
LPARAMETERS tuNewValue
This.lDataChanged = tuNewValue

* Set app-level dirty flag
IF VARTYPE("goApp") = "O" AND NOT ISNULL(goApp) AND ;
   PEMSTATUS(goApp, "SetDirtyFlag", 5)
   goApp.SetDirtyFlag(m.tuNewValue)
ENDIF

* Update form caption
IF m.tuNewValue
   IF RIGHT(This.Caption, 1) <> "*"
      This.Caption = This.Caption + " *"
   ENDIF
ELSE
   IF RIGHT(This.Caption, 1) = "*"
      This.Caption = LEFT(This.Caption, LEN(This.Caption)-2)
   ENDIF
ENDIF
```

The application object's SetDirtyFlag method, along with an application object property, lUnsavedData, does the rest of the job. SetDirtyFlag, shown in **Listing 48**, checks the value passed to it; if it's .T., some form has unsaved data, so the flag is set to .T. If the parameter is .F., we know that at least one form has just either saved data or restored the previous data, but we don't know the state of the other forms, so we loop through them until we find one with unsaved data.

Listing 48. This method manages an application-wide "dirty" flag, so that we can know just by checking a single property whether there's any unsaved data.

```
PROCEDURE SetDirtyFlag(lNewValue)
* Set the application dirty flag. If the parameter
* is true, then some form has changed data, and we can
* just set this flag. If the parameter is false, some
* form has just saved or reverted its changed data
* and we need to look at all open forms.

IF m.lNewValue
   This.lUnsavedData = .T.
ELSE
   This.lUnsavedData = .F.
   FOR EACH oForm IN _VFP.Forms FOXOBJECT
      IF PEMSTATUS(oForm, "lDataChanged", 5)
         IF oForm.lDataChanged
            This.lUnsavedData = .T.
            * No need to find more than one
```

```
            EXIT
         ENDIF
      ENDIF
   ENDFOR
ENDIF

RETURN
```

In most data-entry applications, the next step would be to check the "dirty" flag on exit and prompt the user to save changes if there's unsaved data. The Library application was designed so that when you close a form, the data is automatically saved, so there's no need to prompt.

Delegate handling of a new value

For one application, I needed the ability to manage a complex set of user preferences. The preference items correlated more or less directly to either application object properties or properties of objects managed by the application object. For example, one preference addresses how often garbage collection takes place; this property needs to become the Interval for a timer. By creating an application object property for each preference and giving them Assign methods, I ensured that the appropriate updates happen automatically.

The Library application has only a couple of items in its Preferences form, shown in **Figure 10**. Use large toolbars maps directly to an application property; it uses BindEvent() to trigger resizing of the buttons in the toolbar. (Note that this code is incompatible with the toolbar resizing code discussed earlier in this document.)

The checkbox that controls whether the application shuts down after a specified period of inactivity also maps onto an application property. However, the timer that tracks inactivity (described in "Monitoring user activity" earlier in this paper) needs to be appropriately enabled or disabled when the user changes the checkbox. Similarly, the spinner that determines how long to wait for activity needs to set the Interval for that timer. (None of this happens until the user closes the Preferences form.) Both items use Assign methods to ensure that the appropriate changes occur.

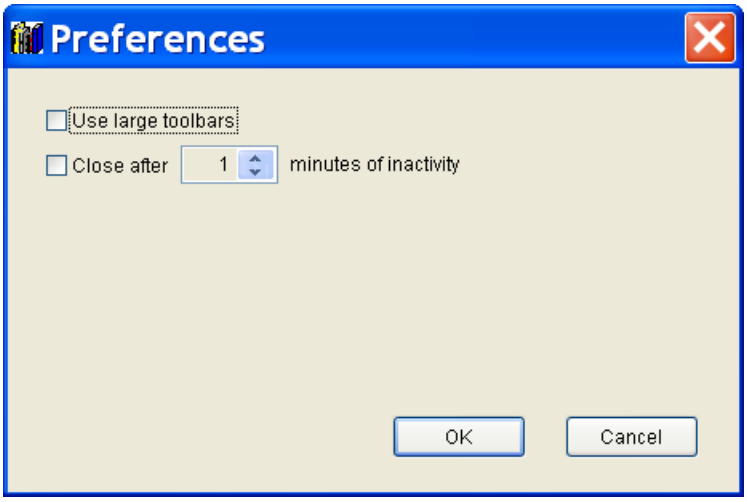

Figure 10. The Library application's preferences form relies on Assign methods to ensure that the inactivity timer is set appropriately.

When the user closes the form, the control values are stored to corresponding application properties. The spinner value is multiplied by 60000, the number of milliseconds in a minute, first. The adjusted spinner value is stored in an application property called nActivityTimerInterval. The nActivityTimerInterval_Assign method sets the timer's Interval, as shown in **Listing 49**. Note that we need to store the value in the application property, as well as setting the timer's Interval. Otherwise, the next time we open the Preferences dialog, it won't set the spinner to the current value.

Listing 49. This assign method sets the Interval for the activity timer when the user changes the setting in Preferences.

```
PROCEDURE nActivityTimerInterval_Assign(nNewValue)
* Something changed interval for activity timer. Propagate to timer

IF VARTYPE(m.nNewValue) = "N" AND m.nNewValue > 0 AND ;
   m.nNewValue <> This.oActivityTimer.Interval
    This.nActivityTimerInterval = m.nNewValue
    This.oActivityTimer.Interval = m.nNewValue
ENDIF

RETURN
```

The checkbox value is stored to the lTrackUserActivity property. Its Assign method enables or disables the timer, as shown in **Listing 50**.

Listing 50. When the user changes the preference for tracking user inactivity, the Assign method of lTrackUserActivity fires.

```
PROCEDURE lTrackUserActivity_Assign(lNewValue)
* Tracking decision changed. Set up or disable timer

IF VARTYPE(m.lNewValue) = "L" AND m.lNewValue <> This.lTrackUserActivity

   This.lTrackUserActivity = m.lNewValue
```

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

RETURN
```

Check for validity

One of the things you can do with an Assign method is check the new value of a property for validity and reject invalid values. For example, I wrote a Sudoku game in VFP a few years ago (as a demonstration of business objects; it's described in http://tinyurl.com/y84fj5p3). The bizGame object, which represents the game as a whole, has a property named nSize that holds the board size (the number of cells in either direction). Since Sudoku requires the game size to be a perfect square, nSize has an Assign method that checks; it's shown in **Listing 51**.

Listing 51. The nSize_Assign method of the bizGame object of a Sudoku application ensures that the specified game size is valid.

```
* Ensure that size is a perfect square
LPARAMETERS tuNewValue

IF SQRT(m.tuNewValue) = INT(SQRT(m.tuNewValue))
   This.nSize = tuNewValue
ENDIF

RETURN
```

Propagating data inside a container

One of the main ways I use Assign methods is to push data down a hierarchy inside a container, so that only the container is exposed to the world. There are a variety of behaviors along these lines that are useful.

Earlier in this document, I showed how to use Access methods to provide dynamic tooltips and to have the same tooltip for a container and its contents. Assign provides an alternate way to do the latter, in situations where you don't need dynamic tips. That is, this approach works when you want to assign a fixed tooltip to a container and have all the controls inside show the same tip.

My code for this uses a custom logical property, lPropagateTooltipsDown, in the base container class. The ToolTipText_Assign method contains the code in **Listing 52**. If the container has the flag set to .T., when the ToolTip changes, we loop through the objects in the container and change ToolTipText for each of them.

Listing 52. This code in the ToolTipText_Assign method lets you propagate a container's ToolTipText to the contained objects.

```
LPARAMETERS tuNewValue
```

```
This.ToolTipText = tuNewValue

LOCAL oObject
IF THIS.lPropagateToolTipDown
   FOR EACH oObject IN THIS.OBJECTS FOXOBJECT
      IF PEMSTATUS(m.oObject, "ToolTipText", 5)
         m.oObject.ToolTipText = m.tuNewValue && don't use ToolTipText on the right
         && in case it also has an access method
      ENDIF
   ENDF
ENDIF
```

Note that we don't have to do this recursively, even if the container contains other containers. The ToolTipText_Assign method for the contained objects will fire and, as long as they have lPropogateToolTipDown set to .T., handle their contained objects.

Adjust a label

In one application, I needed to have a label running vertically inside a shape. In the relevant form (shown in **Figure 11**), the label's captions are determined dynamically based on the data. I created a label subclass and gave Caption an Assign method. That method calls a custom TurnCaption method to rebuild the caption adding the appropriate punctuation to make it display vertically. For the double-wide shapes in the top row, the same method breaks the caption into two strings of about the same length and builds the appropriate caption string. TurnCaption is shown in **Listing 53**. (Note that CHR(160) is a non-breaking space.)

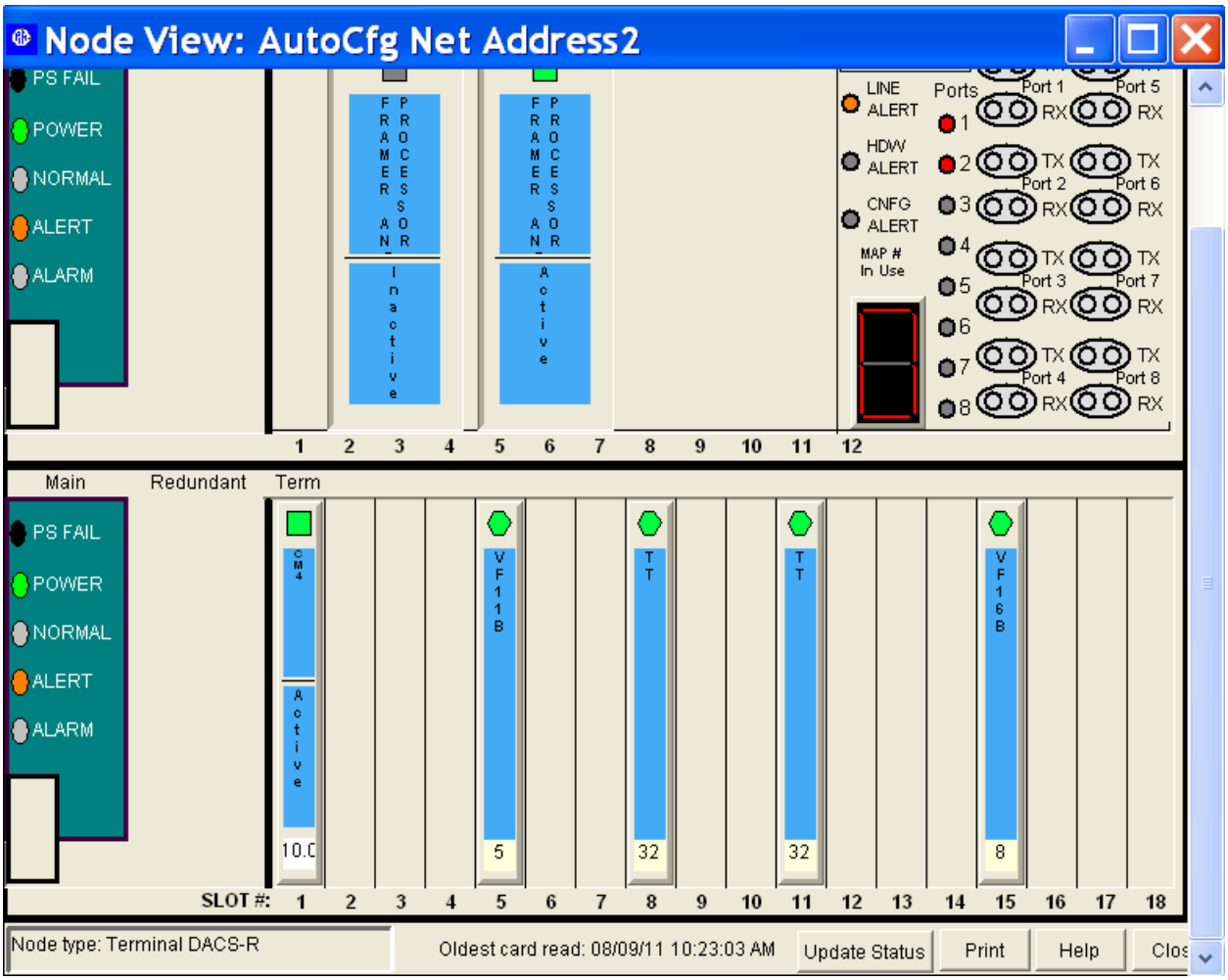

Figure 11. The vertical labels inside the boxes here are set up by an Assign method.

Listing 53. This code is called by the Caption_Assign method of the vertical labels shown in **Figure 11**.

```
LPARAMETERS cCaption, lTwoColumns

#DEFINE LF CHR(10)
LOCAL cTurnedCaption, nLength, nChar

m.cCaption = CHRTRAN(CHRTRAN(m.cCaption, CHR(160), " "), LF, "")
nLength = LEN(m.cCaption)

IF m.lTwoColumns
   * Find a dividing point
   nWords = GETWORDCOUNT(m.cCaption)
   IF m.nWords > 1
      * Break on a word break
      cColumn1 = GETWORDNUM(m.cCaption, 1)
      cColumn2 = ALLTRIM(SUBSTR(m.cCaption, LEN(m.cColumn1) + 1))
      cNextWord = GETWORDNUM(m.cColumn2, 1)
      nWord = 2

      DO WHILE m.nWord <= m.nWords-1 AND ;
```

```
         LEN(m.cColumn1) + LEN(m.cNextWord) < LEN(m.cColumn2) - LEN(m.cNextWord)
            cColumn1 = m.cColumn1 + " " + m.cNextWord
         cColumn2 = ALLTRIM(SUBSTR(m.cColumn2, LEN(m.cNextWord) + 1))
         cNextWord = GETWORDNUM(m.cColumn2, 1)
         nWord = nWord + 1
      ENDDO

      nLength = MAX(LEN(m.cColumn1), LEN(m.cColumn2))

   ELSE
      * Just break it in half
      cColumn1 = LEFT(m.cCaption, FLOOR(m.nLength/2))
      cColumn2 = RIGHT(m.cCaption, CEILING(m.nLength/2))

      * Pad shorter string
      cColumn1 = PADR(m.cColumn1, LEN(m.cColumn2))
   ENDIF
ELSE
   cColumn1 = m.cCaption
   cColumn2 = ""
ENDIF

cTurnedCaption = ""

FOR nChar = 1 TO nLength
   cTurnedCaption = m.cTurnedCaption + EVL(SUBSTR(m.cColumn1, nChar, 1),CHR(160))
   IF m.lTwoColumns
      cTurnedCaption = m.cTurnedCaption + CHR(160) + CHR(160) + ;
                       SUBSTR(m.cColumn2, nChar, 1)
   ENDIF
   cTurnedCaption = m.cTurnedCaption + LF
ENDFOR

RETURN m.cTurnedCaption
```

For the Library application, you could use this code if you want to display books graphically with titles running down the spine.

Debugging with BindEvent(), Access and Assign

As you can imagine, having bound events and Access and Assign methods makes debugging more complicated. Sometimes a method seems to run out of the blue, with no sign of it being called.

There are no magic bullets to make this kind of debugging easier. However, the same good debugging practices that work for VFP code generally are useful here. In particular, I find the DEBUGOUT command, breakpoints and tracing to be the most effective tools.

When the sequence of events doesn't seem to make sense to you, scatter DEBUGOUT PROGRAM() through your code, putting it into every method that seems involved. Then, when you run the code with the Debug Output window available, you'll see the sequence of

methods called. (Better yet, you don't have to remove these statements before distributing your application. DEBUGOUT is ignored in the runtime environment.)

Once you've narrowed things done to a particular section of code, set a breakpoint to stop execution at the beginning of that section, and step through the code one line at a time to see exactly what's happening.

In a few cases, I've found that even tracing doesn't solve the mystery for me, or that tracing the code interferes with whatever process I'm looking at. In such cases, I use the Coverage Logging tool. I turn logging on at the point when I think the issue begins, then run through the code until I've executed the problem section. I then use the generated Coverage log to show me exactly what lines of code executed in what order.

The other tool that's helpful when debugging BindEvent() issues is the AEvents() function. I can use it at any breakpoint to see exactly what's bound, as well as to see whether a bound event got me to this point.

For more general debugging tips, check out my paper at http://tinyurl.com/ycqrjufe.

Set it and forget it

As I wrote this paper, I was actually surprised at how many ways I've found to use BindEvent(), Access and Assign. Looking through the examples here, one of the main themes is "set it and forget it." That is, often what these features let you do is set up a desirable behavior in your base classes, and then use it over and over without having to remember how you did it. In some cases, you have to set a property or two, but in general, the techniques in this article cut down the amount of code you have to write. Isn't that the goal of OOP?

