Expert

Black Belt Programming

Share Data Between Object Instances

by Karl E. Peterson

Object purists might claim that one control or class instance shouldn’t need to know about others or share global data. However, I’ve found that objects often need to know about other instances of themselves, as well as be able to share data and events freely with those other instances.

Notwithstanding a previous Black Belt Programming column [“Override ActiveX Controls,” VBPJ/December 1997], you can share data among multiple instances of a Visual Basic-authored UserControl. By adding a standard code module to a UserControl project and declaring Public variables (standard, object, collection, whatever) within it, you give all control instances within the current application access to this global data. And if you ratchet up that strategy a notch, a code module can wrap access to its own private data through various subroutines and functions that massage the data before exposing it to control instances. I’ll show you several methods for doing just that.

Imagine a scenario where all control instances require notification in response to an external cue. For example, I found myself needing such notification in a digital-clock control I wrote. True to form, the colons between the hours, minutes, and seconds blinked on the half-second. But then one user pointed out that when she placed several instances on a form, each instance blinked out of sync with the rest.

Originally, I placed an intrinsic Timer control on the UserControl, updating each clock’s display based on the Timer events of its own timer. But to solve my problem, I had to set up a system timer callback, notifying each control instance when this single timer fired. This concept might apply to other situations as well, including a variety of system hooks to notify you of “events” such as keystrokes, mouse movements, file system changes, or display property changes. Or perhaps you have objects within your application, which provide similar notifications that need to propagate to a variety of controls.

At any rate, remember that the “problem” here is not the issue; it just provides an excuse to try various techniques. What’s interesting is comparing the potential

---

**Table 1** Cost/Benefit of Each Approach. You can use any of three methods of notifying multiple objects of singular events. Each method has its strengths and weaknesses. Event-driven notifications are the easiest to implement, and can be fairly flexible if done with a class. But for the ultimate in speed and flexibility (though you must accept some responsibility for safe-coding), you can’t beat the callback approach.

<table>
<thead>
<tr>
<th>Task/Issue</th>
<th>Event-Driven (Form)</th>
<th>Event-Driven (Class)</th>
<th>Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocked in IDE by MsgBox</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Blocked in EXE/OCX/DLL by MsgBox</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Available through Remote OLE</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Complexity</td>
<td>Minimal</td>
<td>A little more</td>
<td>Considerably more</td>
</tr>
<tr>
<td>Problem potential/Severity</td>
<td>None</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Extensibility/Adaptability</td>
<td>Limited to available controls</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Efficiency/Speed</td>
<td>Lowest</td>
<td>Marginally higher</td>
<td>Highest</td>
</tr>
</tbody>
</table>

---

Karl E. Peterson is a GIS analyst with a regional transportation planning agency and serves as a member of the VBPJ Technical Review and Editorial Advisory Boards. Based in Vancouver, Washington, he’s also an independent programming consultant specializing in ActiveX controls and contributes to various journals. Karl coauthored Visual Basic 4 How-To from Waite Group Press. Online, he’s a Microsoft MVP, and a section leader in several VBPJ online forums. Find more of Karl’s VB samples at http://www.mvps.org/vb.

http://www.devx.com
solutions, while remembering that the problem will vary from project to project. I'll present three workable solutions for this issue, ranging from simple but inflexible to robust but complex (see Table 1). The strategies don't change much, whether you're developing a UserControl, class, form, or UserDocument. However, UserControls do present some odd requirements, because they run in both design and run time, so I'll focus on them.

**ADD A SHARED EVENT SINK**

To notify a group of objects when something happens, the simplest strategy is to add a shared event sink, which in turn raises the event to all concerned objects. In the case of a timer, you can do this by adding a form to the project, placing a timer on it, and referencing the global instance of the form provided by VB from within each object.

For example, on a form called “FSharedTimer,” add an intrinsic Timer control and set its Interval appropriately. Within the form's Timer1_Timer event, raise your own Timer event. In your control or class objects, declare a reference to the global instance of the form using WithEvents, and sink the events as they're raised (download Listing A from the free, Registered Level of The Development Exchange).

In this and the following examples, your object uses the tmr_Timer event procedure to react appropriately to the external notification. In this example, I simply refresh the display. Before the notification event is fired, a static Boolean flag variable (stored in the event generator) is toggled to indicate what state should be used during the next display update.

The simplicity of the shared-form approach is its greatest strength, but it solves only this particular problem. You can't modify the technique easily to notify your objects of other system events or callbacks. The next step up in complexity is declaring a global instance of a class, rather than a form, that fires notification events. Of course, to take a class approach, you need to employ a system timer callback instead of an intrinsic timer control.

With complexity comes flexibility—this approach lends itself to alteration for other sorts of system callbacks. Start by adding two modules—one class and one standard—to the control project. Define the class, CSharedTimer, with Instancing set to Private, making it invisible outside the control/library.

Within the standard module, place the TimerProc callback procedure that the system will call on each timer event, along with a global declaration of a CSharedTimer class object. The CSharedTimer class starts a system timer on Initialize, kills it on Terminate, and provides a Public method called RaiseEvents. On each system call into TimerProc, RaiseEvents is called and the Timer event fires.

The strategy within each control object hardly varies from the shared-form example. The only real difference is that during the ReadProperties event, rather than loading the form the class is initialized (see Listing 1).

You now have a fairly robust, flexible approach to sharing events among object instances. You can extend this class-based method to most any sort of system notification. And it doesn’t take much more code added than the form-based method. However, you can't use events through remote OLE; being inherently late-bound, they're not the fastest method of notification; and they're blocked by MsgBox calls while running within the IDE. Though annoying, you can work around that last problem by calling the MessageBox API directly. And it doesn’t affect the compiled app.

**REGISTER FOR CALLBACKS**

Still, for the most robust strategy, you'll build a collection of objects and iterate on cue. In a standard module, write aRegisterInstance routine that's called as each object is initialized. As with the prior examples, make this call in the
ReadProperties event of UserControls and in the Initialize event of other objects. In controls, you may query the Ambient.UserMode property to determine whether or not to enable notification events. Note that this property tosses an error during an Initialize event, as the control isn’t sited yet. Passing Me to RegisterInstance lets you add a reference to the new instance to a collection (see Listing 2).

RegisterInstance first ensures that the private collection has been initialized by setting it to a New Collection if it’s currently Nothing. Then a pointer to the passed object is stored within the collection. Storing an actual object reference sets up a circular death trap—easily avoided with the pointer (see “Referring to Parent Properties” in Ask the VB Pro, VBPJ May 1998). For the final step in the registration, initiate the external stimulus you’ll be reacting to by setting up a system timer callback.

Again, UserControls might require extra handling. System events firing into a control when it’s running at design time might not always be desirable. Exposing a Friend property (invisible to the outside world) of the control allows the registration routine to determine in which run mode the control is currently operating:

```
Friend Property Get AmbientUserMode() As Boolean
    AmbientUserMode = Ambient.UserMode
End Property
```

As controls come, they also go. So you need to write a corresponding UnRegisterInstance routine that reverses the registration process (see Listing 3). The Terminate event of your object typically passes this routine a reference to Me. Terminate begins by removing the pointer to your object from the collection of objects. If the count has fallen to zero, that provides the signal that the last instance being tracked has terminated and it’s time to shut down (or stop responding to) the external stimulus. With a timer, just kill it.

**CALL ALL YOUR OBJECTS**

As the system timer callbacks start flooding in, the TimerProc routine simply iterates the collection, firing off a method call to each stored object (see Listing 4). As each object pointer is retrieved from the collection, you convert it to an object reference with an easy (albeit sleazy) hack:

```
Dim lpObj As Variant
Dim Cntl As Clock
CopyMem Cntl, 0&, 4
```

CopyMem, very bad things will happen.

Declare a variable of the desired object type, but never instantiate it within the procedure.

Copying the stored pointers to this early-bound object variable provides a tidy way to hijack a reference to the original object. But be sure not to let this affect the object’s reference count; letting the stolen reference go out of scope or setting it to Nothing will destroy the original! Manually dereference the borrowed reference with a second memory copy after you’ve finished using it—or else.

In the first two event-driven methods, I use an event procedure to accept notification. But this third callback method requires exposing a Friend procedure in the UserControl (or class):

```
Friend Sub Timer(ByVal hWnd As Long, ByVal uMsg As Integer)
    Me.Refresh
End Sub
```

If your object notifications need to stop and start at will, a few steps are required with any of my three approaches. A control might expose an Enabled property that the user could toggle at any point. In that case, rather than automatically register the control at instantiation, you’d only register the control if its En-
abled property were set to True. Likewise, at termination, you’d only unregister if Enabled were True. Within the Let Enabled property procedure, add additional code to register or unregister as appropriate:

Public Property Let Enabled(ByVal NewVal As Boolean)
    If NewVal <> m_Enabled Then
        m_Enabled = NewVal
        PropertyChanged “Enabled”
        If m_Enabled Then
            Call RegisterInstance(Me)
        Else
            Call UnRegisterInstance(Me)
        End If
    End If
End Property

With the callback registration scheme, you have to keep track of your current state at all times. If it gets out of sync and tries to reregister an already registered instance, you’ll experience minor annoyance in debugging at the very least; or the system could tank if unanticipated events continue to stream into your module after the objects are all gone. As your design complexity increases, consider more rigorous error-checking in the registration routines.

NOTIFY AN INTERFACE

So far I’ve only talked about notifying a specific interface—that of a known object. But what if you have multiple objects of different types, each needing to be notified of these events? The event-driven models handle that requirement easily, but would the callback model make you code multiple collections and iterate each?

Here’s a perfect situation for Implements to come to your rescue. You can create a notification interface that each of your varying objects will implement. In the case of my clock example, the ISharedTimer class would look like this:

Option Explicit
Public Function AmbientUserMode() As Boolean
End Function
Public Sub Timer(ByVal ShowColons As Boolean)
End Sub

You can then add another UserControl to your project; perhaps one that counts down rather than up, placing this code in both modules:

Implements ISharedTimer
Private Function _
    ISharedTimer_AmbientUserMode() As Boolean
    ISharedTimer_AmbientUserMode = Ambient.UserMode
End Function
Private Sub ISharedTimer_Timer(ByVal ShowColons As Boolean)
    Me.Refresh ShowColons
End Sub

You need small modifications to the registration and TimerProc routines, replacing all references to the primary interface (Clock) with references to the common secondary interface (ISharedTimer). This highlights one of the cool things about secondary interfaces: You can still pass a reference to Me when the parameter is As IWhatever. The received reference will be queried to ensure the proper interface is implemented.

You might not think you’d have multiple controls within the same project, all needing simultaneous notice of a singular event. But remember, these techniques work with all object types. It might be much more likely for you to have several classes, all needing to be notified when your Internet connection dies, or your server crashes, or any number of other things.

By implementing a common interface and properly registering each instance, you can notify all with one simple loop, whenever the need arises.

Of the three methods, I think callbacks offer the most benefits, but also require the most diligence to code. Note you can only use the callback method through Remote OLE. Making callbacks into your objects always provides more speed than raising events, making this solution more suitable to time-critical code. Last but least, callbacks are never blocked by a MsgBox.

Code Online

You can find all the code published in this issue of VBPJ on The Development Exchange (DevX) at http://devx.vbpj.com. For details, please see “Get Extra Code in DevX’s Premier Club” in Letters to the Editor.

Share Data Between Object Instances

Locator+ Codes

Listings for the entire issue, including a routine that adds new objects to a collection as they register themselves (free Registered Level): VBPJ0898
 Listings for this article only, including the routine listed above, plus three fully functional control project groups, each demonstrating one of the methods discussed in the column (subscriber Premier Level): BB0898