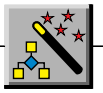

VISUAL PROGRAMMING

Create Windows API Object Controls

Using a Timer control is one thing, but knowing how it works internally is another.

by Richard Hale Shaw

This month, I'll take a break from the DigitalClock control and look at a different issue: creating objects that encapsulate the Windows API.

As any reader of this column knows, objects are the wave of the future. While OLE has been a portent of an object-oriented Windows, OLE is only the beginning. In future versions of Windows, most of your programming will involve using objects. Rather than using APIs, you'll instantiate an object for a particular Windows service, and interact with that object in order to put the service into use.

This kind of object-oriented Windows used to be termed "Cairo," but the concept of Cairo has changed a great deal in the last several months. A lot of what we once thought of as Cairo will appear in Windows NT 4.0 this spring: the Win95 shell, PC card support, and distributed OLE. Instead, Cairo will now include connectivity and Internet-related features that will be more powerful than what was originally envisioned.

But a significant aspect of all new versions of Windows is that most new Windows services—and eventually, many older ones—will be packaged as OLE controls. And many of these controls will provide encapsulations of various APIs. For example, there are plenty of APIs whose use requires a sequence of events such as these:

- Initialize a data structure.
- Call an API to register the data structure with Windows.
- Call another API to establish a callback function address.
- Call various APIs to actually perform the required service.
- Implicitly register an application window with Windows by passing a window handle as an argument to various API functions.
- Respond to invocations of the callback function by Windows.
- Respond to messages sent to your application window as a result of calling various APIs.
- Call an API to de-register the data structure and terminate the service.

This kind of thing goes on all the time in Windows, and a great deal of what an object-oriented development tool such as the Microsoft Foundation Classes provides is encapsulation of the core behavior that makes all this work. But sometimes MFC by itself isn't enough. Using an MFC object means instantiating it and writing code to utilize its member functions and data members. You can access an OLE control, however, simply by dragging and dropping it from a tool palette into your program. And many OLE controls let you access data through properties (exposed in property sheets) and activate functionality by setting other properties (such as an Enabled property). So you often don't have to write any code to use an OLE control.

Richard Hale Shaw is a contributing editor to Visual Basic Programmer's Journal and PC Magazine. He's currently completing Visual Programming++, *a book about Visual C++. He lives in Ann Arbor, Michigan, and can be reached on CompuServe at 72241,155, or the Internet at 3998368@mcimail.com.*

INTRODUCING THE TIMER APIs

A great example of a set of APIs that you can easily encapsulate—not just with an MFC object, but through an OLE control—are the Windows Timer APIs. The Timer APIs let you implement periodic processing that's initiated on receipt of a WM_TIMER message. They let you create a Windows timer that will send WM_TIMER messages to a given window in your application. The timer sends the WM_TIMER messages on a regular basis at an interval that you specify when you create the timer. You have to add WM_TIMER handling to the specified window, or design your program's message loop to monitor the receipt of these messages and then have the window procedure dispatch them to a particular window.

Using the Timer APIs requires you to make a few decisions. First, of course, you have to decide what your program is going to do every time it gets a timer message. You create a handler for WM_TIMER messages (a message-mapped override of CWnd::OnTimer in an MFC application); or, instead, specify a callback function that Windows will directly invoke. You'll also have to decide how long the timer interval—the interval between WM_TIMER messages—will be. You specify this interval in milliseconds: the faster the timer, the smaller the timer interval. Thus, a fast timer might generate WM_TIMER messages 10 times a second (a 100-millisecond interval), a more moderate timer might generate messages every second (a 1000-millisecond interval), and a slower timer would generate a WM_TIMER message every 10 seconds (a 10,000-millisecond interval).

To create a timer, call the SetTimer API function (CWnd::SetTimer in MFC), and specify the interval, the address of the timer callback function (or NULL to handle the WM_TIMER messages yourself), and a timer ID. The timer ID lets you create more than one timer, where each WM_TIMER message includes the ID of the timer that generated the message. (This ID is passed as a parameter to CWnd::OnTimer in an MFC application.) Once you call SetTimer, the WM_TIMER messages will continue at the appropriate interval until you call the KillTimer API (CWnd::KillTimer in MFC), passing it the timer ID as an argument. KillTimer will destroy the timer.

One problem with the Windows Timer APIs is that there's no convenient way to stop and restart the same timer, or to change the timer interval on a timer. If you want to temporarily stop and then restart a timer, you have to call KillTimer to

destroy the timer, and then SetTimer to re-create it. And you have to use the same approach to change the timer interval. There's no way to change the interval in mid-stream, so you'll have to go through the creation-destruction process again.

This can end up being a lot of work. MFC helps, of course, but its encapsulation of the Timer APIs is kind of thin. And it doesn't solve the stop-restart and interval change problems. So having a timer control—an encapsulation of the Timer APIs in the form of an OLE control—will definitely help.

Fortunately, Visual Basic 4.0 comes with such a control. You probably saw it in the form of a VBX in previous versions of VB. The VB4 Timer control can be drag-and-dropped into a form or dialog. You can specify the timer interval through the Interval property, and just set the Enabled property to True to enable the control. Whenever the Enabled property is True and the Interval property is greater than zero, the Timer control will send Timer Events to its parent form or dialog at the specified interval. Probably the only disadvantage to using the Timer control is that it's available only in the Visual Basic 4.0 environment. You can't use it in VC4 or other OLE control containers.

Just using the Timer control is one thing, but knowing how it works internally is another. So I decided to clone the Timer control and figure out how it works. In this column, I'll show you how I built my own Timer control—one that you can use in VC4 as well as VB4.

ENCAPSULATING THE TIMER APIs

I started by using ControlWizard to generate a pretty vanilla control. The one option that I took advantage of was "Invisible At Runtime," which causes ControlWizard to add the OLEMISC_INVISIBLEATRUNTIME bit to the miscellaneous registry settings in the generated code:

```
static const DWORD BASED_CODE
   _dwTimerOleMisc =
   OLEMISC_INVISIBLEATRUNTIME |
   OLEMISC_ACTIVATEWHENVISIBLE |
   OLEMISC_SETCLIENTSITEFIRST |
   OLEMISC_INSIDEOUT |
   OLEMISC_CANTLINKINSIDE |
   OLEMISC_RECOMPOSEONRESIZE;

IMPLEMENT_OLECTLTYPE(CTimerCtrl,
   IDS_TIMER, _dwTimerOleMisc)
```

With this bit-setting option on, the control container knows to display the control only during development, but not at run time. After all, timers need to be used programmatically, but the end user doesn't need to see them displayed in a form or a dialog. Therefore, the VB4 and VC4 environments will display the control in a form or dialog, but at run time, VB4 and an MFC4 application won't display the control. I made the Timer Control part of an OCX called "SHAW WINAPI OBJECTS.OCX," and the associated programmatic ID SHAWWINAPIOBJECTS.TimerCtrl.1, the name that refers to this control in the system registry. With these steps in place, I could now start modifying the code generated by ControlWizard.

Getting the initial timer support in place wasn't too difficult. The VB4 timer control uses two properties: Enabled, a BOOL stock property, and Interval, a short-integer custom property. I used ClassWizard to add support for these. For the Interval property, I selected the ClassWizard option for creating a pair of Get/Set methods to create a pair of member functions, GetInterval and SetInterval, for controlling access to the Interval property. I additionally needed to create a data member to store both properties, so I used the Visual C++ 4.0 ClassView to add the new data member. You simply select the ClassView tab to display the ClassView, right-click on the class name, select "Add Function" (see Figure 1), and enter the new data member type and name (see Figure 2). With these two properties in place, I had to add some program logic:

• Code to start the timer when the control's window was

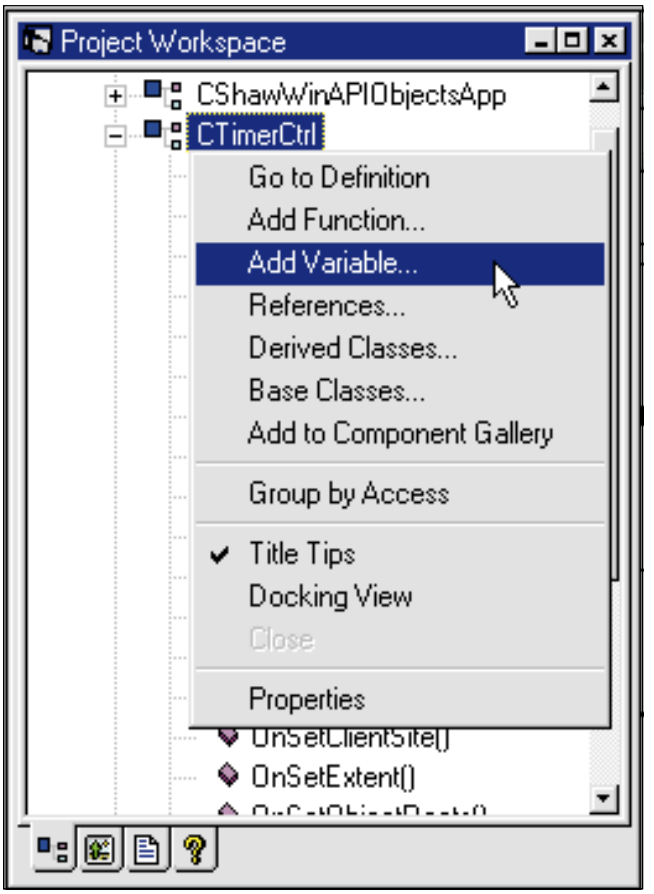

FIGURE 1 ***Right-Click for More Options.*** *Right-click on any class name in ClassView to display this menu. You can use it to add new data members and member functions, look up browsing information, and configure the ClassView window.*

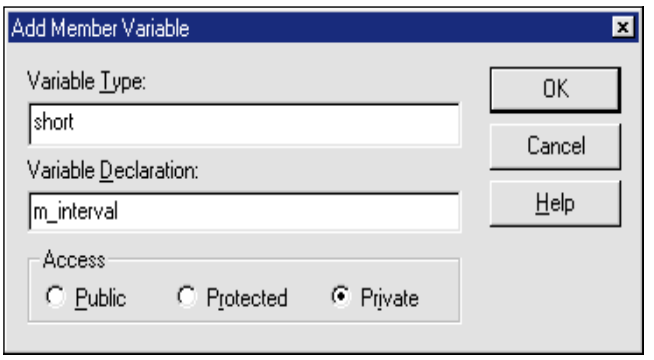

FIGURE 2 ***Adding New Data Members to the Class.*** *After selecting "Add Variable…" in ClassView, you can specify the data type and name of a new data member and add it to the class.*

created, provided that Enabled was True and Interval was greater than zero.
• Code to detect changes to either of these properties, and start or stop the timer depending on their settings.
• Code to save changes to the Interval property whenever it changed.
• Code to stop the timer, provided it was already running, when the control's window was destroyed.

Code that starts a timer would use CWnd::SetTimer; code that stops the timer would use CWnd::KillTimer. To accommodate these changes, I added a data member, m_isRunning, which would always identify the state of the timer control and whether the timer was running. This became particularly useful when the Enabled property changed, because the default implementation of this stock property is through a data member and a member function in COleControl: COleControl::m_bEnabled and COleControl::OnEnabledChanged. This default implementation doesn't give you any granular control over the actual change to the Enabled property, so you have to distinguish between the previous state of the control (as defined by m_isRunning) and the current state of the control (as returned by COleControl::m_bEnabled and my own m_interval). Fortunately this just required a little housekeeping in my OnEnabledChanged override:

```
void CTimerCtrl::OnEnabledChanged( )
{
   COleControl::OnEnabledChanged();
   if(m_bEnabled && (m_interval > 0))
      {
      if(m_isRunning)
         return;
      SetTimer(1,m_interval,NULL);
      m_isRunning = TRUE;
      }
   else
      {
      if(!m_isRunning)
         return;
      KillTimer(1);
      m_isRunning = FALSE;
      }
}
```

I had to add similar logic to the SetInterval function.

CREATING A CUSTOM EVENT

With the timer running, the control's window would get WM_TIMER messages at the appropriate interval (remember that because COleControl is a CWnd-derivative, every COleControl has a window—an HWND—associated with it). I needed to fire an event to the control container's window, which is the form or dialog that contains the control. So I used ClassWizard to add support for a custom Timer event, using its Add Event pane. ClassWizard will make a few changes to your control source code when you add this custom event. First, it adds an entry to the event map (a lot like an MFC message-map) to define a new custom event and specify the event-firing function:

```
BEGIN_EVENT_MAP(CTimerCtrl,
   COleControl)
   //{{AFX_EVENT_MAP(CTimerCtrl)
   EVENT_CUSTOM"Time", FireTimer,
      VTS_NONE)
   //}}AFX_EVENT_MAP
END_EVENT_MAP()
```

It also defines the event-firing function as an inline function in your control class definition:

```
//{{AFX_EVENT(CTimerCtrl)
void FireTimer()

{FireEvent(eventidTimer,EVENT_PARAM(
      VTS_NONE));}
//}}AFX_EVENT
```

The variable, eventidTimer, is simply an ID that ClassWizard assigns to a custom event in a set of enumerated values:

```
enum {
//{{AFX_DISP_ID(CTimerCtrl)
dispidInterval = 1L,
eventidTimer = 1L,
//}}AFX_DISP_ID
```

Note that ClassWizard uses this same set of enums to assign an ID to the Interval property as well.

With the custom event support in place, the control can now fire Timer events to the control container by calling the FireEvent function. So I added an override of CWnd::OnTimer—which will be called whenever the control gets a WM_TIMER message from the timer—and had the override call FireEvent:

```
void CTimerCtrl::OnTimer(UINT
   nIDEvent)
{
   if(m_isRunning)
```

```
        FireTimer();
    COleControl::OnTimer(nIDEvent);
}
```

Note that the implementation here lets the control fire a timer event whenever it gets a WM_TIMER message, and as long as m_isRunning is True. This would let me alter the design of OnEnabledChanged in a future implementation of the control so that the timer would keep running even if Enabled was False, but the timer messages would be translated into events only if Enabled was True.

CREATING A NONRESIZABLE WINDOW

With this, the basic timer control infrastructure was in place. But now I had two additional problems. The first was to draw a nonresizable bitmap on the control window during development (remember, at run time the control won't be seen because of the OLEMISC_INVISIBLE-ATRUNTIME bit). Drawing the bitmap wasn't too difficult, and that was added as the implementation of the control's OnDraw function. OnDraw, as you may recall, is a virtual function in COleControl that will be invoked whenever the control needs to redisplay itself. By overriding this virtual function in a COleControl-derivative, you can add your own drawing code to the control. The first parameter to OnDraw is the device context to draw on. The second and third parameters specify the boundaries of the control's rectangle and the portion of that rectangle to be re-drawn, respectively.

I screen-snapped the bitmap used by the VB4 timer control to create a new bitmap, pasted it into the VC4 bitmap editor, and then added it to the control's resources. The OnDraw override in the Timer control begins by loading that bitmap from the resources:

```
CBitmap bitmap;
bitmap.LoadBitmap(IDB_BITMAP1);
```

CBitmap is the MFC bitmap class, derived from CGdiObject, which encapsulates most of the Windows GDI APIs. OnDraw instantiates a CBitmap object, and LoadBitmap loads the bitmap and attaches it to that object.

Next, OnDraw instantiates a CPictureHolder object to display the bitmap in the control's window. CPictureHolder was designed for implementing picture properties (an OLE control feature that originated in VBXs), and for displaying bitmaps, icons, and metafile picture properties in a window. You can use CPictureHolder::CreateFromBitmap to associate a CBitmap with a CPictureHolder:

```
CPictureHolder picHolder;
picHolder.CreateFromBitmap(bitmap,
    NULL,
    FALSE);
```

Then, tell the CPictureHolder object to draw the bitmap in a specified device context with CPictureHolder::Render:

```
picHolder.Render(pdc, rcBounds,
    rcBounds);
```

CBITMAP IS THE MFC BITMAP CLASS, DERIVED FROM CGDIOBJECT, WHICH ENCAPSULATES MOST OF THE WINDOWS GDI APIs.

This takes care of getting the bitmap to display properly when the control is drag-and-dropped on a form. Note, by the way, that OnDraw doesn't have to release the original bitmap through a call to CGdiObject::DeleteObject: the CGdiObject destructor does this for you automatically.

The second and more difficult problem involved getting the control to display in a nonresizable window. As with the original VB4 timer control, I didn't want a developer to be able to resize the control window. Instead, the control should snap back to its original size whenever you try to resize it. After a great deal of hunting (and a few flurries of e-mail to a friend on the MFC team at Microsoft), I found two helpful functions in COle-Control: OnSetExtent and OnSetObjectRects.

COleControl::OnSetExtent encapsulates the implementation of the control's IOleObject::SetExtent function, to handle resizing the control. By overriding this function, I was able to force the

control always to be resized to the same size:

```
BOOL CTimerCtrl::OnSetExtent( LPSIZEL
   lpSizeL )
{
   lpSizeL->cx = lpSizeL->cy =
      MinMaxSize;
   return TRUE;
}
```

Returning True indicates that the size change was accepted (when, of course, the function has forced a specific size). The MinMaxSize value contains the control's bitmap dimensions.

The other function, COleControl::OnSetObjectRects, implements IOleInPlace-Object::SetObjectRects to handle repositioning and resizing of the control's window. I didn't need an override of this function in place to get the control to display properly in VB4 or VC4—but it made a difference in how the control was displayed by the control Test Container, which comes with VC4. Test Container appears to do some of its drawing differently from other containers. I wasn't worried about this, because control users have no reason to use Test Container. But, to make the control look a little better there, I overrode this function and plugged in the appropriate control dimensions:

```
BOOL CTimerCtrl::OnSetObjectRects(
   LPCRECT lpRectPos,
      LPCRECT lpRectClip )
{
   CRect* lpPos = (CRect*)lpRectPos;
   CRect* lpClip = (CRect*)lpRectClip;
   lpPos->right = lpClip->right =
      (lpPos->left + MinMaxSize);
   lpPos->bottom = lpClip->bottom =
      lpPos->top + MinMaxSize;
   return
COleControl::OnSetObjectRects(lpPos,lpCl
   ip);
}
```

Finally, I had one particular problem concerning VB4 itself. I noticed that the timer simply didn't work correctly in VB4, even though it worked just fine in the control Test Container as well as in an MFC4 application. In VB4, the timer wasn't being created when Enabled was True and Interval was greater than zero.

A little debugging showed that the control's OnCreate override wasn't being called when the control was used in VB4, and thus that the control's window was never being created. I had to conclude that, by default, VB4 doesn't get the control to create a control window if the control is "Invisible At Runtime" like this control was. (Apparently, there's no standard on what a container should do with such a control, hence the differences in behavior between VB4 on the one hand, and VC4 and the Test Container on the other hand.) The only way to ensure that the control's window was created was to override COleControl::OnSetClientSite, which is called whenever the container initializes an OLE object's client site: the point at which the object connects to and communicates with the container.

A GREAT EXAMPLE OF A SET OF APIs THAT YOU CAN EASILY ENCAPSULATE ARE THE WINDOWS TIMER APIs.

Because this has to be done by *every* container for *every* control, an override of OnSetClientSite could call COleControl::RecreateControlWindow, which forces a control to destroy its window (if it has one), and then re-create that window or initially create it, if it didn't have one. Even though the control worked fine in Test Container and MFC4 applications without this code, it allowed my control to work in VB4 also:

```
void CTimerCtrl::OnSetClientSite()
{
   RecreateControlWindow();
}
```

I've provided a VB4 program that shows the use of both my timer control clone and the original timer control in VB4 (see Figure 3). You can download this program from the Magazine Library of the *VBPJ* Forum on CompuServe (type "GO WINDX" with WinCIM and search for VP0396.ZIP), the *VBPJ* Development Exchange World Wide Web site (http://www.windx.com), the *VBPJ* site on The Microsoft Network (GO WINDX), or the VBCD. ×