Using Forms with TurboCAD DLLs

Written by: Rob Raine, 26th January 2005

We've all been there. You've created half a dozen macros that do most of your work for you, but you're sick and tired of opening menus and dialogs to run them; and IMSI can't seem to get around to making it simple to put them on a toolbar. Then you come across an article explaining how to mistake your own toolbars using VB to make a DLL. Hallelujah! A couple of hours later (most of it designing pretty pictures for your buttons) and you're ready to try it. But your forms don't work? Well, I'm going to tell you how to get them working.

Why they don't work

It's something with being in process and out of process, stuff like that. A form isn't allowed to stand alone as a class module, it seems – it has to be encapsulated inside a class module. However, the details don't matter much, because the main point is to get it working, right? Right.

Getting it working

So, apparently we need to hide our form in a class module. TC is allowed to talk to a class module, and a class module is allowed to talk to a form. Think of it as a translator, think of it as a sales rep. Pick your metaphor and let's get on with it.

Quite an event

In many years of VB and VBA programming, I've somehow managed to avoid putting my own events into class modules. But we can't do this form business without events, so I had to learn this first. The MSDN archives helped here but you won't have to go through that because I'm going to give you the condensed version.

Getting started

"Enough of the rhetoric!" I hear you cry. Fair enough, let's get started. If you haven't already made a toolbar using the example code, then you should go back and figure that out first. From here on in, I'm assuming you have the source code for the PaperColors DLL example loaded into VB (avail at http://throsn.tripod.com/maketools.html).

Making the form

Good news: this is the easy part. Unfortunately, I haven't worked out how to get VBA forms into VB in an editable format. They seem to come in as some kind of read-only template. Nothing in the help files seem to mention it, so I just started from scratch. Anyway, add a form and call it `frmDialog`. Add a text box, call it `txtString` and set its "Text" property to "Test Text". Add a button, call it `cmdOK`, and give it a caption. "OK" might be an appropriate choice!

Adding code to the form

Now, this form is going to need some code, so open the code view and enter the following code:

```
Option Explicit

Event Feedback(ByVal strString As String)
```

The first is doubtless no stranger to you. It’s there to stop us being lazy and, more importantly, avoid spelling mistakes. The second line might be new. It is custom event that will enable to form to talk back to the class module it belongs to. More on this soon, just trust me for now and add the following code:

```
Private Sub cmdOK_Click()
   RaiseEvent Feedback(txtString)
   Unload Me
End Sub
```

Here’s some familiar stuff and something new. Now I’ll finish the explanation about events. You no doubt recognise this as code for a command button. This is, of course, an event. When you click the button, it creates an event that is handled by this subroutine. Think of the button as another piece of code. It’s watching the mouse move around, and waiting to be clicked. When you click it, it changes itself to look like a pressed button and then tells the form to run the code in this routine. Then it takes control again, and re-draws itself as an un-pressed button before going back to counting clock ticks, pixels, or whatever a PC does when nobody’s poking it with a mouse or keyboard. Think of the RaiseEvent keyword as the way the button gives control to the form. You’re familiar with the idea of using methods to run subroutines in other modules – an event is the other way around. It’s a way to run code that’s back in the module you came from. So, when you press the OK button on this form, it will run any code in the “Feedback” event of the form’s parent (that’s the class module that created it, which you’ll meet very soon), and then unload itself.

Before we finish the code for the form, there’s another chunk of code that needs to be added:

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
   ' Handle the closing of the form.
   Select Case UnloadMode
      Case vbFormControlMenu
         Cancel = True
         Me.Visible = False
      Case vbFormCode
         ' Form is unloading.
   End Select
End Sub
```

Don’t ask me what this is all about. It’s something that runs when the form unloads and there are, it seems, a number of ways to unload it. I don’t know why and I don’t much care at the moment. No doubt the time will come when we need to know and one of us will figure it out and write an article about it. It doesn’t seem necessary for this demo to work, but you might need it later.

A touch of class

Now, the problem is that the MultToolEx class module can't deal directly with this form; otherwise we wouldn't be messing around with all this custom event nonsense. If you've ever used Microsoft's Common Dialog control, this is a similar thing. All the bits and pieces are hidden inside a class. Now we need to create that class. This class will be the go-between (insert your chosen metaphor here) that will allow the form and TC to talk.

First create a class, and call it clsDialog, then add the following code:

```
Option Explicit

Private WithEvents mDialog As frmDialog
Private mString As String

Event Feedback(ByVal strString As String)
```

You know about the usual first line, for tidy programmers like us. Then we need to have private control of the form, which is allowed – this class will deal with the form. We need to declare it using the WithEvents keyword because, well, it has events. If you don't, it won't work. Do you need to know about thermodynamics to drive to work? No, you don't. Unless you're a thermodynamic engineer or something – but even then you probably won't use that knowledge until you've had your first cup of coffee at the lab, anyway. But I digress. Then we declare a private variable for storing the returned string from the form. Later we'll make a property so the MultToolEx class can read it. The last line is also familiar. We're not going to use it in this thrilling instalment, but this event is how the form can communicate directly with TC in real time, if need be. All you would need is code in the MultToolEx class to handle events. I think. All right, I admit it: I haven't tried it yet, but that's what the MSDN tutorial did, anyway.

Now we'll need some stuff that you'll be very familiar with if you've ever made your own class with objects in it:

```
Private Sub Class_Initialize()
   Set mDialog = New frmDialog
End Sub

Private Sub Class_Terminate()
   Unload mDialog
   Set mDialog = Nothing
End Sub
```

In case you haven't, these two events are run when you create (initialise) an object from this class module and when you destroy (terminate) it. Yes, I did say initialise with an 's' – I'm British, so I don't use a 'z' unless I have to! What they do is create and destroy the form object. You must destroy objects when you've finished with them, otherwise they'll wander around in memory, making trouble. Windows is a pretty flaky system at the best of times, but having loose objects banging about is asking for trouble. The terminate event also unloads the form, just in case it wasn't already unloaded. Belt and braces will make sure your trousers don't fall down. Remember this analogy next time you're having a dream about arriving at work naked.

Now let's do something interesting – some event handling code for our custom event in the dialog form. Add this:

```
Private Sub mDialog_Feedback(ByVal strString As String)
  RaiseEvent Feedback(strString)
  mString = strString
End Sub
```

This code is pretty straightforward – any time frmDialog does a RaiseEvent command, this code will be run. It will take the string that is fed back (remember, frmDialog gives it the content of the txtString control) and put that value into the private mString variable. It also raises its own event, in case you want to do something back in the MultToolEx class. Feel free to experiment with that, but don't blame me if it all ends in tears.

So, the contents of that textbox have made it back to clsDialog. That's good news, but we need to make it visible to the MultToolEx class. One way to do this would just be to make mString a public variable. But that would make it read/write, and I don't want that. So we'll add this code, so it will be a read-only property from the MultToolEx class's point of view:

```
Property Get Text() As String
   Text = mString
End Property
```

This simply creates a property called Text that can be read but not written to – more on that later. The only thing left is to create some code to actually kick off the form so we can open it up and type stuff into it, so add this:

```
Public Sub GetText()
   ' This subroutine will show the dialog.
   mDialog.Show vbModal
End Sub
```

All this function will do is show the form and make it modal. In normal language, "modal" means the form has the focus and keeps it. You can't get back to TC until the form is closed, so you can't do anything sneaky behind its back. The other method, modeless, means you could switch back to TC, so you could have some kind of active tool that would work on a selection. I haven't tried anything like this yet, though.

Right, so now we have our class module for handling the form. It has one method, GetText, which will open the form, and a property, Text, which we will use to find out what was entered on the form.

Adding to the MultToolEx class

Now all we need to do is add a bit of code to the MultToolEx class so we can run the form.

First we need a variable to be able to deal with the clsDialog class module, so add the following code to the declarations section of the MultToolEx class:

```
Private WithEvents mDialog As clsDialog
```

Now all we need is some code to kick off the dialog. So replace this code:

```
Case 4
   tcApp.Properties("PaperColor") = RGB(0, 0, 0)
```

…with this code:

```
Case 4
   Set mDialog = New clsDialog
   mDialog.GetText
   tcApp.ActiveDrawing.Graphics.AddText mDialog.Text, 0, 0, 0, 10
   Set mDialog = Nothing
```

Instead of turning the paper black, the fifth button will now create mDialog, and run the GetText function. This will open the form and show it on the screen. Type something in the text box and click the button. This closes the form, and feeds the contents back to mDialog. Then it will draw the text on the drawing at the origin, in text 10 units high. Finally, it kills off the mDialog object to keep things tidy.

You're ready to take off the stabilisers

That's it! I'm sure your head is now filled with all the wonderful macros and toolbars you're going to create, so off you go and do it before you forget. If you have any questions about this, you can contact me on rxraine@yahoo.com. Be patient – it's good for your health, and I will reply as soon as I can. If there's enough interest, I'll expand on this a bit – feeding data back from the form in a class module, and validating the form, for example. Feel free to make suggestions for future tutorials – if there's enough demand I'll see what I can do.