

Appendix C. MFC and ATL

C.1. Introduction

All of the examples given in this text were created using the MFC ActiveX Control Wizard. By this time, you should be quite familiar with the MFC tools. The one topic that we still need to cover is the creation of property pages. These are quite useful, and in some contexts are almost essential for setting the default values of persistent properties. Many of our examples contain property pages, even though we have not discussed them.

Another important topic that we have not covered is ATL components. ATL can be used to create ActiveX components, and can also be used to create COM and DCOM components. We will focus only on ActiveX components. (ActiveX components are actually a special case of COM components.)

ATL components are usually smaller and faster than MFC components, but they are somewhat more complicated to program. In any case the differences between the ATL and MFC programming styles are significant. We will cover the most important differences between these types of components.

C.2. MFC Property Pages

A property page allows the default values of your component's persistent properties to be set without using a development system like Visual Basic. Property pages are quite useful when designing web-based applications. The property page is a dialog box with conventional controls for setting property values. Numeric and string properties are set

using text boxes, Boolean properties are set using check boxes, and enumerations are set using either radio buttons or combo boxes. (A combo box is a drop-down list.)

The first step in creating a property page is to draw the dialog box using the Visual C++ editor. Provide a control (or a set of radio buttons) for each persistent property. To edit the size of the drop-down list for a combo box, click on the button. To make a set of radio buttons act as a group, display the properties of each button, and click the “GROUP” check box for everything except the last radio button in the set. The tab-order of the buttons in the group is important. To see the tab order, go to the Layout menu and select the Tab Order command. The tab order of the radio buttons must be a contiguous range of numbers, with all but the last control in the tab order having the GROUP property set. When the tab order is displayed, you set the order of your controls by clicking on them in the correct order. Select the Layout Tab Order command again to turn off the Tab Order display.

Right-click on each control to display its properties. Change the name of each component to something you can remember. (This name is only for *your* use.) It is best to leave the first four characters (IDC_ etc.) unchanged. For radio buttons it is the name of the first button in the tab order that is important. The others can be left unchanged.

After assigning mnemonic names to all controls, right click on the dialog box and select the ClassWizard command. Go to the second page of the class wizard entitled “Member Variables”. You will see a list of your controls on the left hand side of the dialog box. Select the top one by clicking on it, and then click on the button “Add Variable”. In the Add Variable dialog box, type a variable name. This variable name has nothing to do with anything else in your component, it is a brand-new variable name in a

new class. Use whatever name you wish. *Do not change the category.* In the “Optional Property Name” box, type the name of the persistent property in your component. This is the property name itself, not the name of the variable holding the property. Click OK, and go on to the next variable. When you’ve finished with all the variables, compile your component, and you’re all set. Test the property page in Visual Basic to make sure you’ve specified the property names correctly.

Property page dialog boxes are supposed to all be the same size. There are two acceptable sizes, one large and one small. The MFC ActiveX gives you the smallest one, which may not give you enough room. If this happens, just make the dialog box bigger. Don’t attempt to set it to the correct size. The first time you test your property page you will receive a warning dialog box that says it is the wrong size. It will also give you the correct dimensions. Write down the dimensions of the largest, and exit from the Visual C++ development system. Open your project directory, and find the file with the *.rc* suffix. Edit this file using the windows note pad. In this file you will find an entry for your property page dialog box. This line will contain the dimensions of the box. Replace these with the dimensions you wrote down, and save the file. (This is the *only* way I know of to get the dimensions correct.)

C.3. The Differences Between ATL and MFC

The MFC ActiveX wizard creates an ActiveX component with every possible feature implemented. The ActiveX standard actually allows one to pick and choose among the features that are implemented. To some degree ATL allows you to design an ActiveX component in this fashion. In most cases, however you will want to implement as much of the ActiveX standard as possible to maintain the versatility of your component.

The first step in designing an ATL ActiveX component is to create a new project of type “ATL COM AppWizard.” Don’t change any of the default selections in the wizard, just click Finish. This creates a project with almost nothing in it. You can now populate your project with COM objects.

Add a single COM object of type “Full Control” to your project. Go to the Insert menu and select the command “New ATL Object”. This will produce a dialog box with two panes. In the category pane, select “controls”. From the objects pane select “Full Control”, and click on the Next button. *Be very careful at this point.* Type in a short name for your component. Now click on the Attributes tab and find the check box labeled “Support Connection Points” and make sure it is checked. If you don’t do this, your component will not be able to have events. There is a way to fix this after the fact, but it is not easy.

Go to the stock properties tab and select all of the stock properties you want your component to support. The class wizard is not usable with ATL projects, so you’ve got to make the decision about stock properties *now*. On the miscellaneous tab, you can select a standard windows control to subclass, if you want to do this. There is also one other important check box on this tab, the one that is labeled “Windowed Only”. Normally ATL components do not have their own window, but share a portion of the host’s window. This can cause problems because the CWnd:: functions will not work without a window. If you anticipate using a lot of CWnd:: functions, check this box. Otherwise leave it unchecked. Windowless activation is faster, and usually more efficient than windowed activation. (For containers that don’t support windowless activation, all ATL ActiveX components will have a window.)

Check over everything on all tabs to make sure everything is correct, and then click OK.

C.4. ATL Properties, Methods, and Events

Lets suppose that when you created your ATL component, the simple name that you typed into the ATL dialog box was “MyCtl”. In the class-name pane to the left of the Visual C++ window, you will see two items beginning with the letter “I”. These are IMyCtl, and IMyCtlEvents. (The second actually begins with an underline.) This is where you add properties methods and events to your component. To add properties and methods, right-click on IMyCtl, and select either Add Method or Add Property, depending on which you want. Let’s first create a property named “First” of type long. Right click on IMyCtl, and select Add Property. Type in the name of the property, “First” and select “long” from the list of types. Leave the parameters box blank, and click OK. This creates two functions in your support class called get_First and put_First. To find the bodies of these functions, click on the + in front of CMyCtl. You will see a second line starting with IMyCtl under CMyCtl. Click on the + in front of this line to display the get and put function names. Double click on the function names to see the function bodies. The generated function bodies are given in Figure C.1. Note that both functions return the value S_OK. This is necessary, don’t change it. To return a value from the get_First function, you must assign the value to pVal. In ATL components, all property functions should return S_OK.

```
STDMETHODIMP CMyCtl::get_First(long *pVal)
{
    // TODO: Add your implementation code here

    return S_OK;
}

STDMETHODIMP CMyCtl::put_First(long newVal)
{
    // TODO: Add your implementation code here

    return S_OK;
}
```

Figure C.1. Get and Put Functions.

Creating a method is somewhat more complicated. Let's create a method, `Second`, that returns a long integer and has one long integer operand. Right-click on `IMyCtl` and select "Add Method." In the Add Method dialog box, type in the method name "Second". In the parameters box, type the following line.

```
[in] long InVal, [out, retval] long * OutVal
```

The generated function can be found in the same way you located the `get` and `put` functions. The generated function is shown in Figure C.2.

```
STDMETHODIMP CMyCtl::Second(long InVal, long *OutVal)
{
    // TODO: Add your implementation code here

    return S_OK;
}
```

Figure C.2. Second Function Implementation.

Note that in Figure C.2, the direction indicators `[in]` and `[out, retval]` are missing. Even though the method returns a value, the implementation function is not permitted to do so. It must return a success/failure code. In most cases this will be `S_OK`. The return

value for the method must be placed in the location pointed to by OutVal. When declaring the operands a method, each must be preceded by a direction indicator. All but the last must have the direction indicator [in]. If the return value is void, the last parameter will also have the direction indicator [in]. Otherwise it will have the direction indicator [out, retval] and it will be a *pointer* to the return value type. The return value of the method must be placed in the location indicated by this pointer.

In ATL, events are the methods of the `_IMyCtlEvents` interface. To create an event, right-click on `_IMyCtlEvents` and select Add Method. Let us create an event named “Third” with one long integer parameter “Note.” You define the event just as if it were a method, except the return value must be void. (I.E. there must be no parameter with a direction indicator [out, retval]. All parameters must have a direction indicator of [in].)

To create the event, we right-click on `_IMyCtlEvents`, select Add Method, and type “Third” as the method name. We then type “[in] long Note” in the parameters box and click OK. Technically, this is all you need to do to define an event, but firing the event will be nightmarishly difficult without a wrapper function. Visual C++ can create the wrapper function for you, but the procedure is somewhat complicated, and subject to weird errors. To create the wrapper functions perform the following steps.

1. Create all your events by adding methods to `_IMyCtlEvents`.
2. Compile your program. Fix all errors until you get a clean compile.
3. Right Click on `CMyCtl` and select “Implement Connection Point.”
4. In the resultant dialog box, click the check-box in front of `_IMyCtlEvents`.
5. Click OK.

6. Compile your program again. If it works fine, but in most cases you will get six compile errors.
7. Double Click on the first error message. This will take you to a line containing the defined constant IID__IMyCtlEvents. Change this constant to DIID__IMyCtlEvents. (Just add a capital D to the front.)
8. Compile your program again.

After completing these steps, your support class will have the wrapper function `Fire_Third`, which can be called to fire the “Third” event.

C.5. Windows Native Graphical Functions

For the most part, the ATL drawing functions are the same as those for MFC drawing with a few parameter changes. Nevertheless, these are Windows native drawing functions, not members of some class. In general, a drawing function of the form

```
cdc->DrawFunc(P1,P2, ..., PN);
```

is replaced with

```
DrawFunc(di.hdcDraw,P1,P2, ..., PN);
```

There are usually fewer choices as to parameters as well. The one notable exception is the line drawing function `MoveTo`. This existed in Windows 3.1, but no longer exists in Windows 95 and above. You need to replace this with `MoveToEx`, and set the fourth (and last) parameter of this function to `NULL`.

C.6. Windowless Activation

The default windowless activation of ATL components can cause several problems. First, even though the drawing routine has an apparent window and device context in

which to draw, what it actually has is a portion of the container's window, and a handle to the container's device context. In most cases, the upper left corner of the drawing window will not be (0,0). If your component assumes that the upper left corner is (0,0), your drawings will not display correctly.

The second problem is that `CWnd::` functions do not work in windowless activation. The most commonly used of these is `CWnd::Invalidate`. Replace this with `FireViewChange` and everything will work. If you need more extensive access to the `CWnd::` functions, you should avoid windowless activation. Although there are ways to duplicate the actions of these functions, they are not always as efficient as using a window.

C.7. Fixed Sized MFC Components

Creating a fixed sized component is not particularly difficult, but you can spend (literally) weeks digging through the documentation trying to figure out how to do it.

Let's suppose you want to create a control of fixed size 32x32. The first step is to add the following line to the constructor of your support class.

```
SetInitialSize(32,32);
```

Then, using the first page of the ClassWizard, add the `OnSetExtent` function to your support class. Comment out the body of this function, and return a zero instead. Keep the body of this function as a comment to remind yourself how to resize a control if you want to resize it yourself later.

For an example of a Fixed Sized ATL component, see the Cards component.

(Although the example was presented as an MFC component, it is implemented as an ATL component on the CD.)

C.8. Conclusion

There are many books on MFC and ATL programming that can be used to extend your knowledge of these subjects. In addition there is extensive documentation on both MFC and ATL in the Microsoft Developer Network Library.