Issue Date: FoxTalk March 1997

Using OptionGroups in Visual FoxPro

Jefferey A. Donnici
jdonnici@compuserve.com

The OptionGroup control in Visual FoxPro is sort of like that oil filter wrench hanging on the wall in the garage -- it's not something that you use on a regular basis, but when you really need it, there's just no alternative! Jeff points out some of the useful tricks you can perform with OptionGroups and discusses a few approaches to making sure your OptionGroup classes use your own sub-classed OptionButtons.

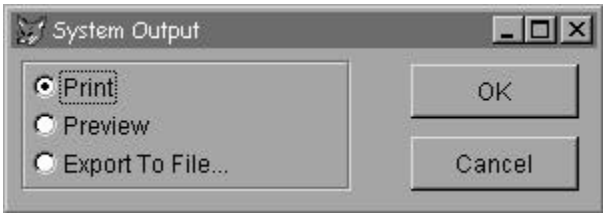
One of the most important aspects of developing software for a graphical user interface (GUI) is staying within the generally accepted guidelines for that interface. This means knowing exactly which control to use for a given interface need. As developers, we're also end users of the development products and tools we use, so think back on a piece of software you used recently that had a truly horrible interface. Chances are that the interface was horrible because the developers of that product chose the wrong controls for the context of the information you were working with. The result is an experience that feels awkward or unwieldy, likely leading to a dissatisfied customer.

OptionGroups are normally used in a portion of an interface where the user may choose from a fixed, relatively small, set of choices. If the set of choices is large (more than a half-dozen or so) or is driven dynamically by the data, then a Listbox control is usually a better choice. When there are only two options to choose from, and those options are the opposite of one another (such as "on" and "off"), then a Checkbox control should be used. I like to think of OptionGroups as meeting the need for all the cases that fall into the "gray area" between Checkbox and Listbox.

The anatomy of an OptionGroup

The OptionGroup control itself is merely a container for two or more OptionButton controls. In versions of FoxPro prior to 3.0, these were called "radio buttons" because their behavior is like that of a car radio. This description still fits quite well because, regardless of how many options you have in the group, you can select only one at a time, and the act of selecting any one will automatically de-select all the others (see **Figure 1**).

Figure 1. A System Output "radio button".



Like other containers in Visual FoxPro, the OptionGroup has a collection property that provides access to its member OptionButton controls. In this case, the Buttons member array contains one element for each OptionButton object found in the OptionGroup. The OptionGroup also has a ButtonCount property, which contains the current number of member OptionButton objects in the OptionGroup container. Using these two properties and the new FOR EACH construct in Visual FoxPro 5.0, you can write code in a specialized OptionGroup subclass that performs some action on each OptionButton member object:

```
LOCAL loButton
FOR EACH loButton IN THIS.Buttons
  *-- loButton is an object reference to the
  *-- current OptionButton in the collection.
ENDFOR
```

One nice enhancement that came when FoxPro 2.x "radio buttons" became Visual FoxPro OptionGroups is the ability to space the buttons using any sort of alignment. The buttons are aligned in a vertical column by default, but you can arrange them individually into horizontal rows, multiple columns, or anywhere else within the confines of the OptionGroup container.

The individual OptionButtons also contain a Style property that indicates the type of button to be displayed. As with check box controls, the choices are Standard (the default) and Graphical. The behavior of the OptionButtons and their parent

OptionGroup container isn't affected by the Style property. It simply indicates the type of visual representation the OptionButton should use. If this property is set to 1 (Graphical), the OptionButton will look like a command button; if it's the selected OptionButton within the OptionGroup, it looks like a command button in a "pressed down" state.

Using the OptionGroup with data

To bind most controls to a data source, you'd set the ControlSource property for the control to the data item it refers to. If you populate the ControlSource of an OptionGroup, however, its Value property will contain the *Caption* for the selected member OptionButton. This probably isn't the behavior you'll normally want, because a change to any button's caption will introduce an inconsistency into the data. If you leave empty the ControlSource property for the OptionGroup, the Value property for that object will contain the index for the member OptionButton that's currently selected. For example, the OptionGroup in **Figure 1** would have a value of 1 because the first object in its Buttons collection is currently selected.

Oddly enough, the class definition for individual OptionButtons contains a ControlSource property as well. You probably won't ever bother with it, simply because it doesn't make a great deal of sense to have a group of OptionButtons in the same OptionGroup bound to different data sources.

In my opinion, the cleanest approach to using an OptionGroup with data would be to *not* use a ControlSource on the OptionGroup but instead to store the numeric value into the data. That way, if you change the captions of the individual OptionButtons you won't affect the data. For reporting purposes, you'd refer to a lookup table to indicate which of the possible choices each numeric value corresponds to. As an alternative, you might use the numeric value from the OptionGroup to search the lookup table containing the actual values and simply store that value into the main table. Either way, a slightly more complex design makes maintenance significantly easier when the client later decides to change the interface or add new options to a group.

And where should you put the code that saves this numeric value or performs the lookup? That depends on when your data is being saved. If your form has a method that handles the saving of new or changed records, then that would be the logical place to poll the OptionGroup for its Value. If the OptionGroup is part of a business object that's dropped onto a form, then the business object should be responsible for gathering the OptionGroup's settings.

The relationship between the values of the individual buttons and the group's value is fairly straightforward. Each OptionButton within the group has its own Value property. If a given OptionButton is the one currently selected within the group, it will have a value of either 1 or .T. (depending on whether the value was set to a numeric or logical value at design time). With that one selected, all the other buttons in the group will contain either 0 or .F.

When a user attempts to add a new record, you might set the OptionGroup's Value property to the numeric value that corresponds to the default choice within the group. This way, if the user doesn't change that selection, you've still got a valid default value in the data. There may be cases, however, when you don't want to have any of the buttons in the group initially selected. The user will then know to select something from the list of possible options. To do this, set the OptionGroup's Value to 0 initially and none of the member buttons in the group will be selected. If a choice is required, though, make sure to check that the value is no longer 0 before saving the user's new record.

What's new in VFP 5.0

Before you get too excited, there weren't a lot of changes to the OptionGroup or OptionButton base classes in Visual FoxPro 5.0. But, given all the other enhancements to the product, I think we can probably look the other way this time. As **Table 1** illustrates, the new properties, events, and methods (PEMs) added to both these classes are primarily the same ones added to most other Visual FoxPro base classes.

Table 1. New PEMs added to the OptionGroup and OptionButton base classes.

OptionGroup	OptionButton
ColorSource property	MiddleClick event
Destroy event	MouseIcon property
MiddleClick event	MouseWheel event
MouseIcon property	Parent property
MouseWheel event	ReadExpression method
Parent property	ReadMethod method
ReadExpression method	RightToLeft property
ReadMethod method	ShowWhatsThis method
ShowWhatsThis	

method	UIEnable event
WhatsThisHelpID property	WhatsThisHelpID property
WriteExpression method	WriteExpression method

Two changes with regard to the OptionButton and OptionGroup classes are worthy of note. First, do you notice anything odd about the OptionGroup list in Table 1? Yes, the good folks at Microsoft fixed that bug found in VFP 3.0 and 3.0b wherein the OptionGroup class definition never had a Destroy( ) event! Now, if it's appropriate to the interface, you can create a generic OptionGroup class that saves the user's selected option into the registry and then restores it in the Init( ) event.

The second change is a minor enhancement in your ability to create OptionButton class definitions. Prior to VFP 5.0, you could create OptionButtons visually using the Class Designer. Unfortunately, this ability wasn't obvious because the dialog resulting from the CREATE CLASS command didn't contain the OptionButton as a base class. Instead, you had to issue something like the following:

```
CREATE CLASS opbMyOptionButton OF
     ClassLib AS OptionButton
```

Thankfully, the CREATE CLASS dialog has been modified in VFP 5.0 to include the OptionButton in the combo box that lists the VFP base classes.

And speaking of subclasses...
I wish I could report that VFP 5.0 has made it easier to use specialized OptionButton classes as members of your OptionGroup class definitions. Sadly, that's still not the case in version 5.0. If you create an OptionButton class and add three instances of it to an OptionGroup container in the Class Designer, VFP won't save those OptionButtons in the class definition when you save the container. VFP doesn't save a record into the Visual Class Library (VCX) table for each member OptionButton, as with most container-member class definitions. Instead, it writes "ButtonCount = 3" into the Properties memo field for the OptionGroup container class' record. The next time you open the OptionGroup class in the Class Designer, VFP evaluates the class definition and adds three instances of the VFP OptionButton base class to your subclassed OptionGroup container.

Obviously, this can wreak all sorts of havoc for developers who strive for the highest level of reusability (and who among us doesn't?). The idea of creating a custom OptionGroup builder that lets you use specialized OptionButton classes is appealing, but nobody wants to write a builder that VFP un-builds!

There's also the problem of the ButtonCount property in subclassed OptionGroups. When a specialized OptionGroup class is used in a form or other container, the number of buttons in the group can't be decreased because those members reside in the original class definition. You can always add member buttons to each instance of the OptionGroup, but then those buttons won't have any of the unique functionality possessed by the original buttons from the class definition.

There are, however, a few different ways that you can solve both these problems at once. You can use your specialized OptionButton classes in an OptionGroup class and make sure that each button in the group has the necessary functionality. The trick is to place the buttons you want into the container at runtime instead of design time so that VFP doesn't undo your changes. The following list describes three possible approaches that you might consider. (You'll find an example of each in the **Download file**.)

- You can add a custom property, cButtonClass, to your OptionGroup class definition and populate it with the name of the OptionButton class you want the group to use. Then add a custom method named ReplaceButtons( ) to the class and call it from the OptionGroup's Init( ). Add code to this new method that goes through the group's Buttons collection and, for each member button found, performs the following tasks: Saves the caption and position-related properties for that button, removes it from the OptionGroup container, replaces it with a button of the class specified in cButtonClass, and resets the saved properties from the original button. Then you can simply set up each OptionGroup instance using the default OptionButton base class. At runtime, the ReplaceButtons( ) method will replace the default buttons with your specialized button class.
- Another approach is to create an OptionGroup class with no option buttons in it and add an AddButtons( ) method to the class definition. Place code in this method to receive two parameters: the caption for the added button and the class name for the button. This method will also need to find the last OptionButton in the Buttons collection and add the new button below it. Next, add code to the Init( ) for each OptionGroup instance that calls the AddButtons( ) method to add the necessary member buttons to the container.
- The last approach is the one that I think developers will find most appealing. It's similar to the previous approach but uses an array to maintain the list of buttons to create during instantiation. This class definition contains no option buttons. Instead, an array property, aButtons, is added to the class along with two methods -- FillButtonsArray( ) and AddButtons( ). To use the class, simply write code in the FillButtonsArray( ) method that populates the aButtons array property with one row for each button to be added. The array would contain two columns: the first for the button caption and the second for the OptionButton class definition to be used. The AddButtons( ) method will loop through this array and add each button in turn. In the Init( ) of the class definition, the FillButtonsArray( ) method would be

called just before the AddButtons( ) method, allowing the OptionGroup to build itself.

Hopefully you'll be able to use one of these approaches, or some combination of them, in your own development to get the most flexible OptionGroup classes possible. Remember, there are times when no other control will do!