How Do I Accomplish This…?

Whil Hentzen
Hentzenwerke Corporation

Overview

Let's face it - the user doesn't really care what language you're using to build their application. And they don't really care what's under the hood - whether you have a data dictionary or a set of C++ application calls or whatnot. But they do care about the interface - and the demands on you haven't changed - in fact, they've just grown.

This session explores a number of "How do I accomplish this?" topics in the area of the user interface in VFP. For example, building a picklist using a grid, creating child forms and sequentially executing forms in this new modeless world, handling global variables, populating controls according to use choices, and handling multiple instances. When you leave this session, you'll have a number of tools and techniques you can put to use right away in your own applications.

Especially useful for you 2.x mavens who've been wondering "How do I do this in VFP?", you'll also find this session applicable if you're new to FoxPro in general.

It’s important to remember, throughout this session, that while the physical applications of controls and classes are not open to interpretation, the implementation of their use is both a matter of preference and opinion.

Global (Application-Wide) Memory Variables

Creating An Application Object

Instantiating an application object (“oApp”) during the startup of your application provides a handle on which you can hang what we used to refer to as “global memory variables.” As long as oApp is in scope, it’s properties are also available to the entire application. This can be done creating a class, oApp, that is stored in a class library such as BASEAPP. The oApp class contains properties that map to the global variables you need in the application.

Creating Global Properties with An Application Object

Instantiating an application object (“oApp”) during the startup of your application provides a handle on which you can hang variables for the application.. Assigning a value to a property of an application object, such as the name of the current user, is done like so:

Accessing this property to a memory variables for use within a method is done like so:

Multiple Instances

Tracking Multiple Instances

Creating an application that loads forms is fairly straightforward. In a menu bar, simply issue the command DO FORM <name> and you'll be all set.

It's fairly common for the developer to be unaware that, unlike earlier versions, a new version of the form will be created each time the menu option is selected. This causes a number of problems, obviously, including confusion on the part of the user when they suddenly have seven copies of the same form on the screen, and application and system crashes when the system resources run out.

In the form's setup code (say, the Init), you can take care of any necessary housekeeping involved with the creation of a form. Some tasks you may need or want to do each time the form is selected, regardless of how many instances are instantiated, other tasks will only be done once. For example, you would want to add the name of the form to the Window menu every time an instance of the form is created. However, you would only want your toolbar to be created once, and for it to talk only to the current form. You might also want to dim the menu option once the form is created.

The way to handle this situation is to set up a global counter - in other words, a property of the application that tracks how many times a form ("any form", not a specific form) was instantiated. Upon the creation of the first form, the counter is incremented. In the code that increments the counter, specific actions that need to occur upon the first form instantiation are included. For example, the navigation toolbar could be created when the counter jumps from 0 to 1. Conversely, the destruction of a form could fire code that decrements the counter, and that code would also perform tasks like removing the toolbar when the counter drops from 1 to 0.

Controlling Number of Instances

Expanding on this concept, the property of the application that tracks how many forms have been created can be created as a two column array, with the first column of the array used for the names of the forms available for instantiation and the second column used to track how many instances have been created. Should you want to prohibit a form from being instantiated more than once, simply check the count for that form upon form load.

Handling the Form Once Instantiated

Centering the Form

Now that we have forms ready and able to be instantiated, the next typical problem we run into is getting the form to look the way we want. In other words, shouldn't the form be centered on the screen? In FoxPro 2.x, you did this by selecting the Center checkbox in the Layout dialog of the Screen Builder. You can set the AutoCenter property to true for the form, but this causes problems during design time. A better solution is to use the code

in the Init() method of the form. (Doing so in the Init() method will cause the form to be displayed after it's been moved to the center of the screen. If you put this code in the Activiate() method, for example, you can see the form "jerk" from the corner to the center of the screen.

Make a Form Modal

In some cases, you'll want to prohibit the user from switching from one form to another while this form is active. You clicked the Modal checkbox in the More dialog in FoxPro 2.x. Set the WindowType property from 1-Modeless to 2-Modal in VFP.

Sequential Forms

A common requirement is for the command button on a form to call a series of forms. In FoxPro 2.x, this was easy to accomplish with code like so:

This worked because you were essentially running a series of subroutines in procedural order. With VFP, when you run a form, you are creating an object that "hangs around" until you explicitly release it. Thus, the answer to running a series of forms in sequential order is to call the second form from the unload() method of the first form, and so on:

Alternatively, you could set the WindowType property of each form to 2-Modal, but this would prevent access to any other window in the system, which may not be the desired effect.

Passing Parameters to a Form

Two related issues to the problems of having one form call another and having forms execute sequentially are passing values from one form to another, and returning values back. Let's look at each issue separately. In FoxPro 2.x, you could pass a parameter to a screen simply by calling the screen (which, again, was really a program) with an argument:

(You also had to include a paramters statement in the appropriate SECTION of the screen setup code.) Since a VFP form isn't a "program" any longer, it isn't as obvious how to pass a parameter to it. Sure, you can guess that you can issue a command like:

do form MYFORM with "Hello Mars"

but where does the "parameter" statement go? Since the load() method is fired first, you might guess that this is where you'd put the parameter statement, but actually, you need to put the statement in the init() method of the form:

Success here, however, is short-lived, since the parameter that is accepted to the init() method is also scoped to the init() method. Remember that the methods of an object can be thought of as subroutines. Once init() has completed, the value of the parameter disappears since it has gone out of scope. The way to keep the parameter around for the life of the form is to create a property of the form and stuff the value of the parameter into the property:

The property, of course, goes out of scope once the form does.

Returning Values from a Form

And once we're done with the form, how do we return a value to the calling program? Of course, we'll use a "return" statement, but the trick is where? It goes in the unload event() of the form:

In this case, the value being returned is the form property cRetVal. In practical use, this is how you'll do it since form properties are the only values that will be in scope by the time you get to the unload() event. The other question, now that we know how to return a value from a form, is "where do we retrun it to?" We are, in essence, treating the form like a function, but the syntax is slightly different:

Finally, be sure to set the WindowType to 2-Modal.

Creating a Picklist

In FoxPro 2.x, a picklist could be fired from a pushbutton. The picklist was fashioned from a browse in a separate window and provided a read-only navigation tool through the table. If the last key pressed was Enter, the current record position was passed back to the calling form and the record pointer was repositioned to refresh the form. If the last key pressed was Escape, the record pointer was repositioned on the original record number.

In VFP, the mechanism is similar, but the tool used is no longer a browse window but a grid object. The init() of the grid form contains code to create and populate as many columns as desired for the picklist. The init() takes parameters to set up the columns so that the picklist can be built as generic and the parameters passed are used to customize the picklist as needed.

Each grid column contains a textbox with code in the keypress() method that traps each keystroke, appends the key if it's alphanumeric, and performs a seek on the accumulated string. Once the keystroke chr(13) (the Enter key) is pressed, the number of current record is passed back from the grid form to the calling form and the grid form is released.

The grid form contains the following code in the unload() method:

Relating Data Controls to Table Fields

Mapping Fields to Controls

In simple forms, the fields in a table map directly to the controls on the form. When buffering is enabled, the buffer of the form is automatically filled with data from the table and the form controls reflect the contents of the buffer. Moving from one record to another or issuing a tableupdate() command cause the contents of the buffer to be flushed to the table, issuing a tablerevert() causes the buffer (and thus the controls on the form) to be refreshed from the table.

However, there is often not a one to one relationship between fields in a table and controls on a form. For example, an option group may have captions of “Spring”, “Summer”, “Fall” and “Winter.” However, in the interest of saving space, the values “SP”, “SU”, “FA” and “WI” are stored in the table. In this situation, the mapping doesn’t work. There needs to be an intermediary that translates the values in the table to the values displayed in the controls.

Another example is the stuffing of multiple fields based on the selection of a particular value of one field. Suppose an Orders table contains a field that contains the foreign key for the Region from which the order was placed. The field Region contains a key, such as “WI”, “NY” or “ON”. The lookup table for the regions contains not only the long description of the Region (“Wisconsin”, “New York” or “Ontario”) but also additional attributes for the Region, such as whether the region has a sales tax, and, if so, what the sales tax rate is.

Since the selection of the Region - the data that is placed in one field - drives the data that is placed in several other controls in the form, but those controls do not map to the table, we again need an intermediary that translates the values in the table to the values displayed inthe controls.

Obviously, you could create a series of methods that handle this population of the form from the buffer, but they’ll each need to be called during navigation and editing. The solution to this type of situation is the use of a method to collectively map data in the buffer to data controls on the form. I call this method “XferToForm.” If the controls on the form are not read-only, but can also be used to enter data back into the tables, a second method would be used to do so, called “XferToTable.” This method would convert the values from the controls back to a format that can be placed into the buffer, and, correspondingly, to the table.

Extending Generic Code

Add Custom Hooks into Generic Routines

A form class typically has a number of methods that are common to all instantiations of the form. Often, however, these methods, while fully functional, need to be extended in specific instances. One mechansim for doing so is to place hooks in the common method that will call routines in the instantiated class. A basic save() method for a form class might look like this:

We can extend this save() method by making calls to methods in the local form that allow us to extend the save functionality:

The B4Save() method is part of the form class as well, and by default, returns true:

The method can be overridden when necessary, but no additional work is required if it doesn't need to be changed. By providing hooks before and after the tableupdate() function, as well as in the event of a failure of the tableupdate() function, significant discrete control can be gained without having to rewrite or ignore the generic save() class.