Issue Date: FoxTalk June 1996

Solve Common Combo Box Problems

Stephen A. Sawyer

Of all the Visual FoxPro controls, none seem to cause more difficulties than the list objects -- ListBox and ComboBox. This article addresses some of the inner workings of the list objects and how you can overcome some of the more common problems and issues that arise when using these controls.

In working with members of our local user group, and monitoring activity on the VFOX forum on CompuServe, certain questions and problems come up repeatedly with regard to list objects. This shouldn't be too surprising, given the complexity of these controls. By my count, the ListBox has a total of 110 properties, events, and methods that the developer can access, and the ComboBox has 119! This is more than any other single control object.

Inside the list objects

Much of the behavior of the list objects becomes more intuitive once you understand their internal workings. The first step to that understanding is the nature of the list object's List Property.

First, the information displayed by a list object is stored internally in its List property. In the case of any RowSourceType that specifies an external data source, such as any RowSourceType other than 0-None or 1-Value, the List Property contains a *copy* of the external RowSource. The List property can be described as "array-like" because while it does have a row or row-and-column structure, it doesn't display other array-like properties; you don't have to DIMENSION it as you would a normal array memory variable, and you can't use Visual FoxPro's array-manipulation functions such as ASCAN() or ADEL() on the List Property. Thus, unlike a Grid Control object in which the control can be thought of as a "window" on the underlying data (in the case of a Grid, this is always a table or cursor), the list objects display a *copy* of the data being used to populate the List Property.

Second, while array memory variables can contain data of any type (numeric, logical, character, date, and so on), the data contained by the List Property "array" can only be character data. If the list object's RowSource contains numeric, date, logical, or other non-character data, that data is converted to a character value for storage within the List Property.

With these two points firmly in mind, we can proceed to examine some of the common stumbling blocks.

When populating a list object from an array, table or cursor, how do I "refresh" the object when I change the items in the data source? The Refresh() method doesn't seem to work.

First, let's consider what the Refresh() method does. When you have a control that is "bound" to a memory variable, table field, or cursor column -- in other words, you have specified a memory variable or field name for the control's ControlSource Property -- that memory variable or field is automatically changed to reflect the control's Value property. However, if the value of the control source changes, (such as when the record pointer is moved in a view or table), this isn't automatically reflected in the control's Value property, hence the need for the Refresh() method, which causes the Value property to be "re-read" from the control's control source.

As you can see, this has nothing to do with a change to the external data that is used to *populate* a list object (specified with the RowSource property). The row source populates the list object when the object is first instantiated. The Refresh() method *will* refresh a data-bound list control when the value of its control source changes, but not when the value of the row source changes. To re-populate the List Property of a list object when the data contained in its row source changes, you have to call the list object's Requery() method.

Some Visual FoxPro developers have learned that setting the NumberOfElements property will achieve the same effect. Since NumberOfElements determines how many elements of an array are used to populate the List property, any change to that value will cause the list object to automatically do a Requery(), as the NumberOfElements could be increased, which would require a re-read of the array.

How do I use a ComboBox to manipulate a numeric field in a table?

To understand what the problem is here, remember that one difference between the List Property and memory variable arrays is that the List Property "array" can contain only character data. This means that, in most circumstances, the control's Value Property can contain only a character-type value.

If you *do* bind a list object to a memory variable, object property, or table field that is of a numeric data type, this value will be interpreted by the list object as the index of the list item, and selecting an item from the list object will store the ListIndex property of the control to its Value property and to its control source. This can be convenient as long as there is some relation between the item in the list and its index, as with a list of months, days of the week, and so on. However, it's more often the case that the numeric value you want to have placed into the memory variable has no relation to the item's ordinal position in the list. It's for this reason that Visual FoxPro developers are re-examining early intentions to use numeric or integer values for

primary keys in their tables.

Does this mean that you can't use a list object to store a numeric value to a property, memory variable, or table field? No, it simply means that you can't use the simplest technique for doing so, namely specifying a control source to bind the control to the data.

In this example, you would *not* bind the ComboBox to the table field and place the following code in the control's InteractiveChange() method:

```
REPLACE customer.nCust_ID WITH VAL(This.Value)
```

Now, because the control is no longer bound to the field in the table, you also have to take responsibility for refreshing the control's value when you move to a different record. This can be done by placing the following code into the control's Refresh() method:

```
This.Value = LTRIM(STR(customer.nCust_ID))
```

[See John Miller's article in the August 1996 issue, "Use a Combo Box to Select Foreign Keys," for an in-depth treatment of this topic, including the construction of a foundation class to handle numeric keys. -- Ed.]

If a ComboBox is configured as a Drop-Down Combo and bound to a table field, the field isn't replaced with a value typed into the ComboBox unless the value exists in the Combo''s List Property. How can I get the user's typed value into the table?

There are three ways to change the Value Property of a list object. The frst is by selecting something from the object's displayed list of items. The second is to change the value of the control's control source and refreshing the control. The third is to explicitly change the Value property of the control. Typing a value into a Drop-Down Combo's text box does *not* affect the combo's Value property, *unless* the value typed is a value that exists in the control's List property.

Because of this behavior, you have to take extra steps to update the control source with the value that the user typed into the ComboBox when the value isn't in the control's List property. You can easily determine when this is the case, since the control's ListIndex will be 0, indicating that none of the list items are selected. Therefore, the following code placed into the control's LostFocus() method does the trick:

```
IF This.ListIndex = 0
   ThisForm.Text1.Value = This.DisplayValue
ENDIF
```

This "blanks" the DisplayValue property (the new Value property doesn't appear in the object's List "array"). Things become a bit more complicated when you want the ComboBox to display the current .Value property and the value of the control source. You might try to use the same approach as that described earlier for updating the control source, but you will quickly become frustrated because the effect of assigning a value to the DisplayValue property that doesn't appear in the object's List property will "blank" the Value property. Likewise, assigning a value to the Value Property that doesn't appear in the object's List property, will blank the DisplayValue property.

This can have disastrous effects if the control is bound to table data. If the table contains a value that doesn't appear in the control's List property, and you store this value to the DisplayValue Property, this will blank both the Value property, and the bound control source. Here's a solution that may prove appropriate for some situations.

Bind (set the ControlSource) the combo not to the table field but to the value property of a TextBox, then bind the TextBox's value to the table field. Configure the ComboBox not as a Drop-Down Combo (Style = 0) but as a Drop Down List (Style = 2). Size the TextBox so that it can be placed directly on top of the combo, with only the drop-down button at the right end of the combo showing. The visual effect is that of a combo box, with no clue that you're actually looking at two controls.

Select the TextBox and select Format/Send to Back from the system menu. This will force the TextBox to instantiate *before* the ComboBox. Otherwise, an error will be triggered if the ComboBox instantiates and the control specified in its ControlSource doesn't exist.

Finally, you have to get the TextBox on *top* of the ComboBox, so place the following line of code in the Combo.Init():

```
ThisForm.TextBox.Zorder(0)
```

This will move the TextBox in front of the combo.

An added advantage of this technique is that you gain the ability to apply an InputMask and Format to the user's input, which you can't do with a ComboBox. If you frequently need this type of control, you can save the whole thing as a class, so it's always available.

If your ComboBox is using 0 - None or 5 - Array as the RowSourceType, there's another and (in my opinion) better way to handle this situation. Remember, the problem (the combo's control source not being updated by the user's entry) arises because the user has typed in a value that isn't in the list. With the "0 - None" or "5 - Array" RowSourceType, you can remedy that problem by simply adding the user's entry to the list.

As in a previous example, you check whether the user entered a "non-list" value by checking the combo's ListIndex in the LostFocus() method code. If it's 0, you then add the combo's DisplayValue to the list:

```
*Combo.LostFocus for RowSourceType = "0 - None"
IF This.ListIndex = 0  AND ! EMPTY(This.DisplayValue)
   This.AddItem(This.DisplayValue)
   This.ListIndex = This.NewIndex
ENDIF

* Combo.LostFocus for RowSource = "5 - Array"
* Code snippet assumes RowSource is a form array property
* aListItems1
IF This.ListIndex = 0  AND ! EMPTY(This.DisplayValue)
   DIMENSION ThisForm.aListItems1[This.ListCount + 1]
   ThisForm.aListItems1[This.ListCount+1] = This.DisplayValue
   This.Requery()
   This.Value = ThisForm.aListItems1[This.ListCount]
ENDIF
```

The code for use with a RowSourceType of 0 - None uses the NewIndex property to set the newly added item as the currently selected item, and thereby insures that the control assumes the value that the user just entered. In the case of the code for use with a RowSourceType of 5 - Array, the Value property is simply set to correspond to the value of the newly added array element.

I have a ComboBox with approximately 2,000 items bound to a table field. It takes about five seconds for the form to refresh each time the record pointer is moved. If I eliminate the refresh of the ComboBox, everything refreshes in a blink, but then the combo doesn't display the right value for the current record.

First, using a list object to present a list of 2,000 items isn't an optimal use of list objects in my opinion. I think a good rule of thumb is that when performance begins to go down (as in this case), it's time to think about another approach.

You can understand why performance goes down in this case when you consider the behavior of a Drop-Down Combo. If the Value property is set to a value that doesn't appear in the control's List property array, the DisplayValue is blanked. This means that every time the Value property is changed (as it is when the value of the control object's control source changes), the control has to look through the list of items to see if there's a match for the new Value property. If you have 2,000 items in the list object's List property, that means that it has to make up to 2,000 comparisons before it can be refreshed.

A picklist using a grid control would be a far superior choice.

How can I create a multi-column list object with the RowSourceType set to 0 - None? The AddItem() method doesn't seem to work properly.

For months I insisted that there was a bug in the list objects' AddItem() method. I finally got it through my head that the bug was in my understanding of how the two methods, AddItem() and AddListItem(), worked, and the clue to this understanding is clearly stated in the documentation for the two methods.

AddItem()*inserts* an item into the list object's List property. AddListItem()*replaces* an item in the List property. To see this in action, and to understand why AddItem() often gives unexpected (but perfectly consistent and correct) results, let's "cheat" a bit. Not only can you populate the List property by using the AddItem() and AddListItem() methods, you can directly assign values to the List property array.

Create a form, and place a ListBox onto it. Set ListBox.RowSourceType to 0 - None. Place the following code in the ListBox's Init():

```
FOR I = 1 to 12
  ldDate = CTOD(LTRIM(STR(I)) + "/01/96")
  lcMonth = CMONTH(ldDate)
  This.List[I,1] =  LTRIM(STR(I))
  This.List[I,2] =  lcMonth
  This.List[I,3] = DTOC(ldDate)
ENDFOR
```

Set the following properties for the ListBox:

```
ColumnCount = 3
Width = 200
ColumnWidths = "20,80,50"
```

When this form is run, you'll see what you should expect; a three-column ListBox showing the numbers 1 to 12, the 12 months, and the date of the first of each month.

With the form running, type the following command in the Command window:

```
_SCREEN.ActiveForm.List1.AddListItem["Julio",7,2]
```

As advertised, the value "Julio" *replaces* the value "July" that occupies the ItemID of 7 and the column of 2 in the list. (Because the list hasn't been sorted or re-arranged in any way, the ListItemId of each item corresponds to its ListIndex.) If you check the ListCount property of the ListBox, you'll see that it remains at 12.

Next, type the following command in the Command window:

```
_SCREEN.ActiveForm.List1.AddItem("Septiembre",9,2)
```

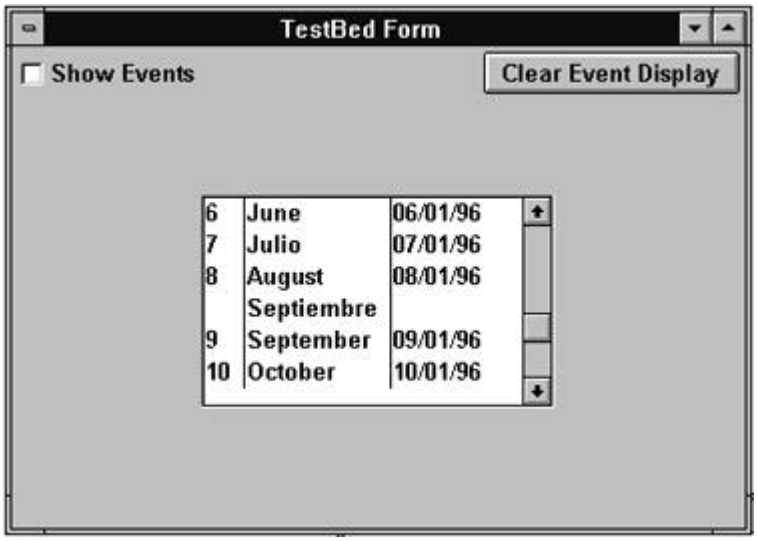
This time, (again, as advertised) the value of "Septiembre" is inserted into the list at the specified (ninth) position, and in the specified (second) column. Checking the ListCount property will show that the ListCount is now 13 instead of 12.

This effect is very difficult to see if the ColumnWidths property isn't set. The following command in the Command window makes it difficult to tell which column the word "Septiembre" appears in:

```
_SCREEN.ActiveForm.List1.ColumnWidths = ""
```

Figure 1 shows the effect of the AddListItem() and AddItem() methods.

Figure 1. The TestBed Form.



How can I prevent the first column of a Drop-Down List from displaying? I have a surrogate key value in the first (bound) column, and have the ColumnWidths set so that the width of the first column is 0. Everything works fine until the user selects an item from the list, and the value of the *first* column in the list is displayed the ComboBox.

The short answer to this is "you can't." The DisplayValue will take on only values from the first column of a multi-column list object. Therefore, the solution is to use the second, or subsequent, columns to hold a surrogate key.

For future reference
When given the opportunity to contribute *The Visual FoxPro Forms Designer* volume to Pinnacle's Pros Talk Visual FoxPro series, my first reaction was "Oh no, now I have to learn how to use combo and lists boxes!" However, I've learned that while they can be overwhelming at first, patient experimentation will usually yield a solid understanding of how each of the properties and methods affect the list objects' operation. To assist in studying the list objects (as well as all other form

controls), I've come to rely on a "Testbed" form. This form (available in the accompanying **Download file**) includes the following:

- Several blocks of code in the Load() method that populate one of two form array properties that are useful for populating list objects
- Several general purpose properties of different data types that can be used as control sources for form controls.
- Method code to trap and report events.

In conjunction with my Testbed form, I've found it productive to use the ability to manipulate forms and form controls from the Command window, as well as the Trace and Debug windows, and I'm continually thankful for the utility of the two functions, SYS(1270) (for storing an object reference to a memvar) and SYS(1272) (to return an object's container hierarchy). With these tools and some perseverance, the list objects soon become no more mysterious or difficult than the command button.