"Session #"

Combos and Lists - The Forgotten Controls

Tamar E. Granor
Editor, FoxPro Advisor

Overview

Grids and pageframes may get all the attention, but combo boxes and list boxes are pretty powerful creatures, too. Visual FoxPro adds significant power to lists and combos, making them far more powerful than their FoxPro 2.x equivalents. VFP 5 makes them even better plus fast enough to use, even with large lists. With power comes complexity. This session reviews the basics of these controls, then shows how to put them to work in your applications. Topics covered include using a numeric ControlSource, populating a multi-column list, adding a user's entry to a combo, and more.

The examples in these notes use the data from Visual FoxPro's TasTrade sample application.

The Basics

Combos and lists are controls that let you present the user with a list of choices. Combos also can allow the user to type in a value not on the list. Combos and lists are common in Windows and in Visual FoxPro. For example, the File-Open dialog in Visual FoxPro uses a combo box for the drive and a list box for files in the current directory.

Although they don't look very much alike, combos and lists (formally known as combo boxes and list boxes) have more in common than not. They share most of their properties, events, and methods (PEMs). Most important, each has a group of data items which are shown to the user. Unfortunately, this is where the terminology gets confusing, because the most natural name for that group of items is "list" - that's the term used below. Most of the time, it should be clear from the context whether "list" means the group of data items or a list box.

Combos come in two flavors (controlled by the Style property) - drop-down lists and drop-down combos. What the two have in common is that they both show a single item most of the time and open up to display the full list (generally as a scrollable object). The difference is that drop-down combos also allow the user to enter a new item not on the list. As we'll see below, Visual FoxPro's handling of such items is somewhat non-standard when compared to other Windows applications.

What are Little Lists Made Of?

Two properties control the contents of a combo or list: RowSourceType and RowSource. The list of items can be built in 10 different ways - RowSourceType determines which of those ways we're using. RowSource then indicates the actual data or tells where to find it. The table below shows how to interpret these two properties.

RowSourceType	Meaning of RowSource
0 - None	Not used. Items are added to the list using AddItem, AddListItem or by manipulating List or ListItem.
1 - Value	Contains the actual data to appear in the list. Items are comma-delimited with no extra spaces.
2 - Alias	Contains the name of an open table, cursor or view from which data is drawn. The first ColumnCount columns are used.
3 - SQL Statement	Contains a SQL SELECT which creates a cursor from which the list draws its data. The first ColumnCount columns of the cursor appear.
4 - Query	Contains the name of a QPR file which creates a cursor from which the list draws its data. The first ColumnCount columns of the cursor appear.
5 - Array	Contains the name of an array from which the list draws its data. The first ColumnCount columns of the array appear.
6 - Fields	Contains the names of one or more fields from an open table, cursor or view from which data is drawn. Only the first field includes the alias of the data source.
7 - Files	Contains a file specification optionally including the wildcards "*" and "?". Files matching the specification appear in the list along with directories.
8 - Structure	Contains the name of an open table, cursor or view. The names of the fields appear in the list.
9 - Popup	Contains the name of an existing popup (created with DEFINE POPUP). The bars of the popup appear in the list.

About half the RowSourceTypes are useful. You can forget 7-Files and 9-Popup immediately - they're really included only for backward compatibility with FoxPro 2.x. There's no reason to use the popup type anymore - the ability to add items directly to the list (RowSourceType 0-None) is far more powerful. As for the files type, as in FoxPro 2.x, you don't have enough control over it to make it truly useful.

What about the rest? Type 8-Structure is useful for building developer tools, but you probably don't want to show end-users a list of field names (and unfortunately, it doesn't use the field captions stored in the database).

Type 4-Query isn't restricted to using the output of the Query Designer. You can actually write your own code as long as the file extension is .QPR. The list uses the current work area at the end of the code. However, pretty much anything you can do this way can also be done with one of the other types.

Type 1-Value is another that isn't quite as brain-dead as it sounds. Although the documentation implies that you're stuck with the exact list of items you originally specified, in fact, a type 1 list does respond to the various methods that let you add and remove items, just like type 0-None.

Nonetheless, it's the remaining RowSourceTypes that are really handy for everyday use. Type 2-Alias works best when you're dealing with a cursor or view (where you have control over the order of fields) rather than a table. It always displays the first ColumnCount columns, and a table usually won't have the data of interest in the first column or columns. Type 6-Fields is often a better choice for drawing data from a table (and can, of course, be used for cursors and views, too).

When using type 3-SQL Statement (or type 4-Query), be sure to direct the query results into a junk cursor. The query is executed as part of the initialization of the combo or list. If the query doesn't include either a TO or INTO clause, the query results appear in the default location - a Browse. You don't want your users to see that Browse as your form is being prepared, so add INTO CURSOR junk as part of the query.

Type 5-Array also has one little twist. You need the array to exist for the life of the form. The way to do this is to make it a custom property of either the combo/list itself or the containing form. On the whole, your best bet is to create a combo or list subclass that has such a property and set RowSource to that array property. When you create an array property in the Form or Class Designer, you need to specify it with brackets and a subscript to tell VFP that you're dealing with an array, so you'd type something like:

```
aListData[1]
```

in the New Property dialog. Then for RowSource, you specify:

```
THIS.aListData
```

if you've added the array as a property of the combo or list. If the array is a form property, use:

```
THISFORM.aListData
```

In either case, you can fill the array with data in either the list's Init or the form's Init. Once you put the data in the array, call the Requery method of the combo or list. (More on Requery in the next section.)

Keeping It Filled

Although a combo or list may be based on any of the things in the list above, in most cases, it internally keeps track of the data it contains. As part of the initialization process, the data is copied to the internal representation. The properties List and ListItem give us access to the control's internal list. The section "Why are there two?" below explains the difference between the two properties.

Sometimes, the data in the source for the combo or list changes. When that happens, we need to update the control. That's the Requery method's job. It goes back to the original RowSource and, once again, copies the data into the internal representation. For example, if RowSourceType is 3-SQL Statement, Requery executes the query again and copies the results into the list or combo. With the two table-based RowSourceTypes (2-Alias and 6-Fields), you can update the list after adding or removing a record either by calling Requery or by setting focus to the list or combo.

Saving the User's Choice

A list's or combo's Value property indicates the most recently selected (highlighted) item in a list or the item last chosen from a combo. Value can be either character or numeric. When it's character, it usually contains the actual data for the item; when it's numeric, it contains the position of the item in the list. To see the current item, you can always check Value.

Often, however, you want to store the user's choice to a field or variable. The ControlSource property lets you do this automatically. Assign ControlSource the name of the field or variable, then whenever the user changes the list or combo's value, the named field or variable is updated. We refer to this as binding the list or combo to the field or variable.

Assigning a particular value to Value is one way to position the highlight in the list (and to control what appears when a combo is closed). If you assign a character value, the item with that value is highlighted and Value and ControlSource remain character. If you assign a numeric value, the item in that position in the list is highlighted and Value and ControlSource become numeric.

By default, when Value and ControlSource are numeric, they contain the position of the chosen item. However, the new BoundTo property lets you change that. With BoundTo set to .T., if the data in the list represents numbers, the

value of the chosen item is stored to Value and ControlSource rather than the position. See the section “Dealing with Numeric ControlSources” below for more on this subject.

Why Are There Two?

Like the animals on the ark, many of the properties and methods related to combos and lists come in pairs - List and ListItem, NewIndex and NewItemId, Selected and SelectedId, and so forth. This is because there are two ways of identifying each item in a combo or list.

Each item is assigned a unique id number when it's added to the list (regardless of RowSourceType). It keeps this id number for as long as it remains in the list. (Requery does renumber the items because it's like starting from scratch.) One whole set of properties and methods give you access to the items based on their unique id. These properties and methods have "Item" or "ItemId" or "Id" in their names. So ListItem is the list of items in id order.

The list can also be examined in display order. Sometimes this is the same as id order, but not always. The list's display order is affected by the Sorted property, by users moving items around when MoverBars is true, and by the ability to add and remove items (for some RowSourceTypes). This order, called index order, always reflects what the user sees. Some of the properties and methods related to index order have "Index" in their names; others are identified simply by not having "Item" or "Id" in their names. List is the list of items in index order.

The ListIndex and ListItemId properties always identify the currently selected item in the list. Use ListIndex for the index (position) of the current item; use ListItemId for its unique id.

With one exception, the Index and ItemId properties and methods for a list or combo are identical. However, AddItem (the Index version) and AddListItem (the ItemId version), which populate a list with RowSourceType of 0-None (or 1-Value) do behave differently by design. When you're dealing with a one column list, the difference isn't terribly apparent, but, as soon as you want to fill a multi-column list, it jumps right out at you.

Here's the secret. AddItem adds a new item to the list every single time you call it. So, after a call to AddItem, ListCount is always higher. AddListItem either adds a new item or replaces the data in an existing item. The effect of this difference on multi-column lists is described below.

Some combos and lists can be sorted by setting the Sorted property to .T. This is possible only when the list or combo actually "owns" the data - when RowSourceType is 0-None or 1-Value. Having Sorted set to .T. is one of the cases where the index and itemid of an item are likely to be different.

A Dazzling Array of Choices

Another thing that many of the list and combo properties have in common is that they are arrays - that is, properties with multiple, indexed values. (Unfortunately, many of the built-in array handling functions don't work on these properties.(

The List and ListItem properties that contain the list data are arrays. So are Selected and SelectedId that indicate which items are highlighted and the ItemData and ItemIdData properties that let you associate a number with each item in a list or combo.

ListIndex and ListItemId can be used to find the current item in any of the arrays. That is, within a method of the list or combo, the currently selected item in the list can be addressed as:

```
THIS.List[ THIS.ListIndex ]
```

or

```
THIS.ListItem [ THIS.ListItemId ]
```

Combos Are Different

By now, it should be clear that, most of the time, it doesn't matter if you're working with a list box or a combo box. Most of what you do is the same. However, there's one area in which combo boxes are quite different. When you set a combo to Style 0-Drop-down combo, the user can type in a value not on the list. (In the Windows 3.x interface, you can tell a drop-down combo from a drop-down list by the position of the arrow. In a drop-down combo, there's a space between the text portion and the arrow. In the new Windows interface, by design, drop-down lists and drop-down combos look identical.)

The value which the user enters in the text portion of a combo box is stored not in the Value property, but in the DisplayValue property. Unlike other Windows applications, the newly entered value is not automatically added to the list. If you want to keep it, you have to add it yourself. You can check whether the user entered a new value by comparing THIS.Value to THIS.DisplayValue in the combo's Valid method.

The technique for adding the new value varies depending on the RowSourceType. With type 0 - None or 1 - Value, it's as easy as calling AddItem or AddListItem to add it to the combo's internal list. For the table based types (2 - Alias and 6 - Fields), you have to add a record to the table, cursor or view, then Requery the combo. Similarly, for an array-based combo, add the new item to the array and Requery. For the other RowSourceTypes, it's more complex and not usually worth the work.

Here's code that works in the case of RowSourceType 0 or 1:

```
* Check whether there's a new item
IF NOT (THIS.Value == THIS.DisplayValue)
   THIS.AddItem( THIS.DisplayValue )
   THIS.Value = THIS.DisplayValue
ENDIF
```

Here's code for a combo based on an array stored in a property of the combo called aComboData:

```
* Check whether there's a new item
IF NOT (THIS.Value == THIS.DisplayValue)
   DIMENSION THIS.aComboData[ ALEN(THIS.aComboData,1)+1 ]
   THIS.aComboData[ ALEN(THIS.aComboData,1) ] = THIS.DisplayValue
   * If the array needs to be sorted, call ASORT() here
   THIS.Requery()
   THIS.Value = THIS.DisplayValue
ENDIF
```

Firing Line

Although there are lots of similarities between them, the firing sequences for combos and lists are somewhat different. In fact, combos have some events that lists don't.

As with other controls, When and GotFocus fire when you land on a list or combo. LostFocus fires when you leave. But some events fire a lot more than you'd expect with these controls.

For lists, InteractiveChange, Click and When fire every time you change from one item to another. If you use the keyboard to move, KeyPress fires before any of them. Choosing an item by clicking on it does *not* fire the list's Valid. That's fired by double-clicking or pressing Enter on an item. The firing order in that case is KeyPress (if appropriate), DoubleClick, When, Valid. If you click on an item which isn't the currently highlighted item, the sequence is longer and more complex (InteractiveChange, Click, When, DoubleClick, When, Valid).

Combos have a similar sequence, but there are some differences. When you use the arrows to move through an open combo, only InteractiveChange fires. Then, choosing an item by pressing Enter or spacebar fires Click, Valid and When. Clicking an item in an open combo performs that whole sequence (InteractiveChange, Click, When, Valid) at once.

Typing into a drop-down list box (that is, no new entries permitted) fires InteractiveChange and Valid, then fires When once for each character that matches an entry. However, if the first letter typed doesn't match any of the entries, only When fires. InteractiveChange and Valid fire again each time the characters entered are enough to move to a different item in the list.

In a drop-down combo, InteractiveChange fires for each character typed.

In addition, combos have some special events. When the combo is opened by clicking on the down-arrow or pressing Alt+DnArrow, the DropDown event fires. In VFP3, it didn't fire until the list was open; that bug is fixed in VFP5. The UpClick and DownClick events fire when you click on the up-arrow and down-arrow for scrolling the opened list. These events fire once you release the mouse.

Speed Demons

In FoxPro 2.x and Visual FoxPro 3.0, combos and lists were too slow to use with anything other than small lists. Fortunately, in VFP5, that's not the case anymore. Both scrolling through list with the mouse and searching with the keyboard have been dramatically improved. (Populating a list with AddItem hasn't changed - use a RowSourceType other than 0 for large lists.)

Tests show that scrolling from top to bottom (by holding the mouse on the down arrow key) is more than three times faster than in VFP3 in a list of about 2,800 items and more than 11 times faster in a list of over 46,000 items. In VFP5, it appears that the time to scroll per item goes down as the list size goes up, while in VFP5, it was fairly steady.

In addition, using keystrokes to quickly move to the desired record in a list has improved dramatically. In VFP3, the amount of time to jump to the appropriate record after pressing a key was not only tied to the number of records to skip, but the time per record increased with the number of records. In VFP5, the time per record seems constant regardless of the number of records to jump, and it's been dramatically reduced.

These changes mean that lists and combos can now be used for data sets of any size (though good interface design may rule them out in many cases).

Looking Good

Several properties have been added to combos and lists in VFP5 to improve their overall appearance. Combos now include Format and InputMask properties, which let you specify the layout of both user input and choices from the list. The interaction of these properties with the items on the list is a little tricky. The user can choose items from the list even if they don't conform to the InputMask, but only the portion which conforms is displayed when the combo closes. For example, if a Combo's InputMask is set to "AAA" (for three alphabetic characters), and an item in the list is "John", when the user chooses that item, the closed combo shows only "Joh", but the Value and DisplayValue both become "John". The Format is applied to what is displayed (so in the example above, a Format of "!" results in display of "JOH"), but again does not affect Value or DisplayValue.

Lists have two new properties that improve their appearance. IntegralHeight reduces the displayed height of the list so that only rows which can be seen in full show. (The actual Height is unchanged.) This prevents the user from seeing part of an entry at the bottom of the list.

ItemTips lets you fit a list into less space than it really needs. When ItemTips is .t., you can hold the mouse over any item which is partially cut-off, and the complete text of the item appears. This is especially useful when only a few items in a list are overly long - you don't have to make the whole list wider. Item tips appear only for items which don't fit inside the width of the list.

Figure 1 shows a combo with phone number formatting and a list with an item tip showing.

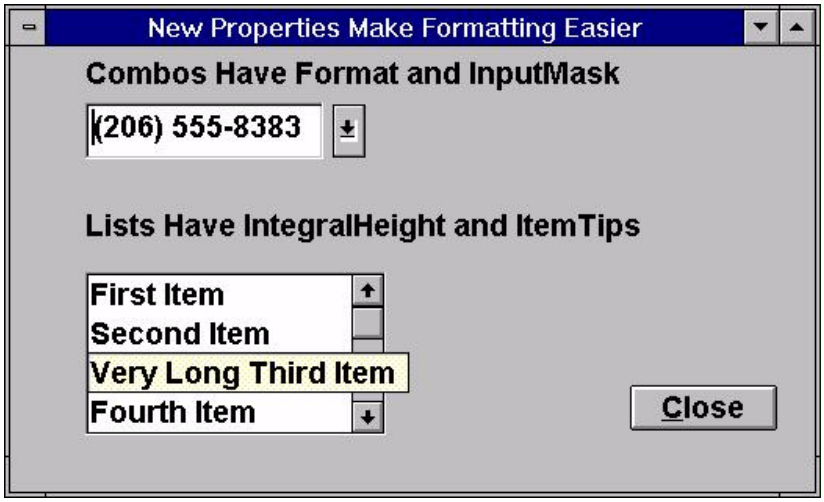

Figure 1 - Both combos and lists have improved formatting capabilities in VFP 5. Format and InputMask let combos control input and format output. ItemTips means a list needn't be as wide as its widest item.

Getting What You Really Want

With the information above, putting together a simple list or combo isn't that hard. But when you go even a little bit beyond the basics, a number of issues can arise to make life more difficult.

Multiple Columns - Easy as Pie

One of the biggest frustrations for users of FoxPro 2.x for Windows or Mac was that making a multi-column list meant giving up proportional fonts. Even then, the technique wasn't obvious. You had to concatenate all the data you wanted in the list into a single field separated by an appropriate character (usually CHR(179)). The good news is that, in Visual FoxPro, you can create a multi-column list that looks good by setting just a few properties.

ColumnCount indicates the number of columns from the RowSource that are to appear in the list. ColumnWidths solves the proportional font problem by specifying how many pixels each column is to use. If ColumnLines is .T., the dividers appear automatically between columns at the positions indicated by ColumnWidths. Voila, an attractive multi-column list. Figure 2 shows a multi-column list, neatly aligned. Here are the properties set to make it happen:

```
ColumnCount = 2
ColumnWidths = "210,100"
RowSourceType = 6
RowSource = "Products.Product_name,Unit_price"
```


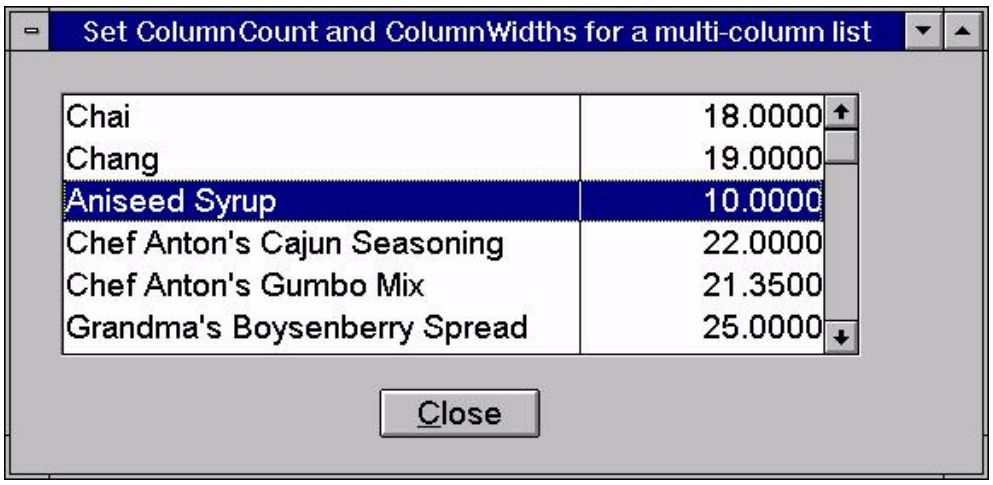

Figure 2 - Creating multi-column lists is as easy as setting ColumnCount and ColumnWidths

BoundColumn lets you indicate which column is to be bound to the list or combo's ControlSource. Since ColumnWidths can specify 0 for any column, you can even show one column, but store the value of another. Figure 3 shows a list of products where the product id code is hidden, but is the one bound to ControlSource. The textbox is there only to show the hidden id value. Here are the properties set for this list:

```
BoundColumn = 3
ColumnCount = 3
ColumnWidths = "210,110,0"
RowSourceType = 6
RowSource = "products.product_name,unit_price,product_id"
ControlSource = "cProductCode"
```

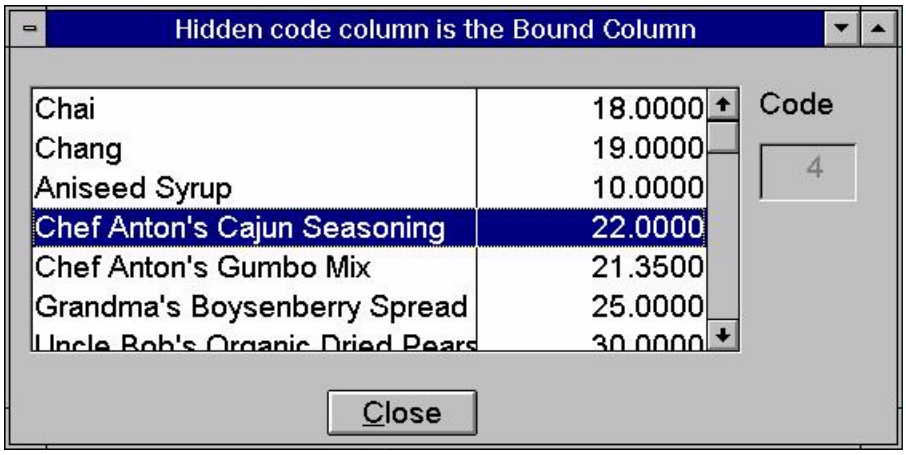

Figure 3 - Set a column's width to 0 to hide it, but you can still bind it to the ControlSource

Fill 'er Up

So what's the problem? The thing that seems to catch most people is trying to populate a multi-column list. If your RowSourceType is anything other than 0-None, this isn't an issue. VFP is smart enough to figure out what constitutes a column and behave accordingly (except for a bug with 7-Files).

But the behavior of the AddItem method with multi-column lists and combos is very confusing. AddItem *always* increases the number of items in the list. So, if you call AddItem once for each column for each item you want to add, you don't get what you expected. Instead, you get several times as many items as you wanted and each one has data in only one column. (VFP 5.0 and 5.0a have a bug - when you AddItem to a column other than the first, the text

of the new item is added in both the first column and the specified column.) Here's the code most people try first - Figure 4 shows the result in VFP5:

```
LOCAL nItemCnt
nItemCnt = 0
SELECT Products
SCAN
   nItemCnt = nItemCnt + 1
   THIS.AddItem(Product_Name,nItemCnt,1)
   THIS.AddItem(English_Name,nItemCnt,2)
ENDSCAN
```

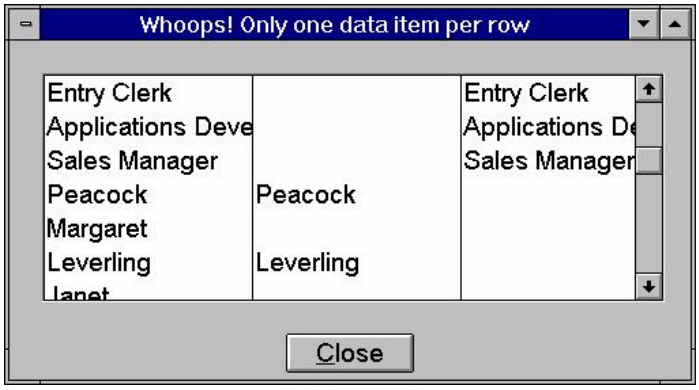

Figure 4 - AddItem can't put data properly in multiple columns. A new bug in VFP5 makes things worse than before.

AddListItem doesn't have this problem - it only adds a new item if you omit the ItemId or give it a new one. Otherwise, it changes the data in an existing item. If you call it several times with the same ItemId, but different column numbers, it fills the columns of the same item. Working together with the NewItemId property (which always contains the ItemId of the most recently added item), it's easy to fill a multi-column list using AddListItem. Here's an example that fills a three-column list with the first name, last name and title of each of the employees in the TasTrade sample:

```
SELECT Employee
SCAN
   THIS.AddListItem(First_Name)
   THIS.AddListItem(Last_Name,THIS.NewItemId,2)
   THIS.AddListItem(Title,THIS.NewItemId,3)
ENDSCAN
```

Since we want to add the items in the order we're processing the table, we don't want to assign the ItemIds ourselves. So we call AddListItem without an ItemId for the first column. Then NewItemId tells us the id for the item we just added and we use it to call AddListItem for the other two columns.

Another approach takes advantage of the (relatively undocumented) fact that the List and ListItem collections can be modified directly. It adds the first column with AddItem, but then plugs the other columns directly into the List collection:

```
SELECT Employee
SCAN
  THIS.AddItem(First_Name)
  THIS.ListItem(THIS.NewItemId,2) = Last_Name
  THIS.ListItem(THIS.NewItemId,3) = Title
ENDSCAN
```

On the whole, though this approach feels simpler, the first is a better choices for two reasons. One, the ability to address List and ListItem directly in this way is undocumented, which means it could change. Second, it depends on

knowledge of the internals of lists and combos; it's better to use the methods provided which will continue to have the same result even if the product changes internally.

Dealing With Numeric ControlSources

The documentation for the List and ListItem collections refers to each as a "character array." The use of the word "character" is this context is a clue to an issue that confounds lots of people. The problem is getting lists and combos to store numeric data to their ControlSources.

Regardless of the RowSourceType and no matter what method is used is to add data to the list, once it gets there, the actual data in the list is Character. (You can see this by looking at an element of the List or ListItem collection in the Debug/Watch window.) Even if the data you put in the list is numeric, even if the RowSourceType is a table and the column in question is numeric, inside the list it's converted to character.

Why do we care? Because Value and ControlSource don't normally convert the data back to its original type. As indicated above, Value can be either character or numeric. When it's character, it holds the data from the selected item (that is, List[ListIndex] or ListItem[ListItemId]). By default, the behavior of a numeric Value for a combo or list is a throwback to FoxPro 2.x - if you make the Value numeric (by assigning it a numeric value), it holds the Index for the currently selected item - *even if the item itself is numeric*.

In VFP3, that was it. To store numeric data from a list or combo, you had to do a lot of tricks. (The conference disk includes a pair of classes called AnyTypeList and AnyTypeCombo that let you bind any data type to a list or combo. They show the necessary work-around.)

Fortunately, VFP5 makes it easy to handle numeric data. Simply set the list or combo's BoundTo property to .T. and, if the contents of the BoundColumn represents a numeric value, Value and ControlSource receive that numeric value rather than the character string stored in List. (However, if ControlSource is character to begin with, the numeric value is stored as a string, just as if BoundTo were .F.)

So Where Am I?

Another task that couldn't be done in FoxPro 2.x was figuring out which part of a list was currently displayed. If the user scrolled the list, you couldn't tell which items he could see right now. In VFP, this one is simple - just check the TopIndex or TopItemId property. They always contain the index and itemid, respectively, of the item now at the top of the list. For lists, but not for combos, you can set these properties, too, so the user is looking at the part of the list you want him to. The form in Figure 5 lets you modify the TopIndex of a list of suppliers. The textbox shows you the current value of TopIndex. If you scroll in the list, the textbox isn't updated until you release the mouse - there's no event that fires when you're scrolling in a list.

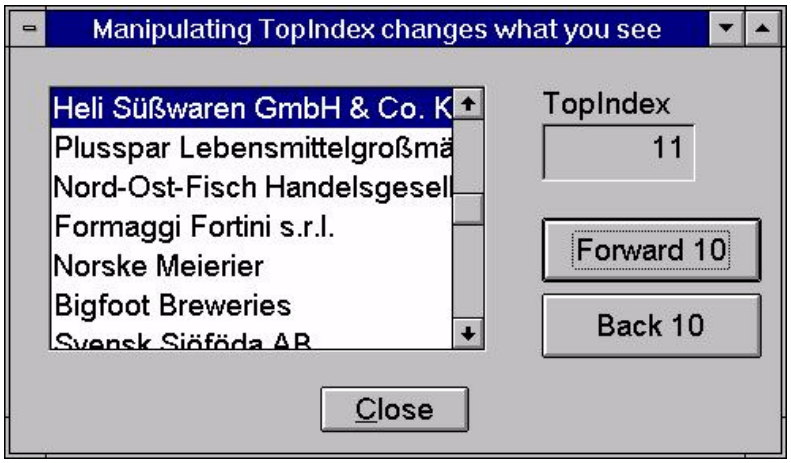

Figure 5 - TopIndex indicates which items in a list are visible at any time.

Special Lists

List boxes have a few special capabilities that combos don't share. The two that are most widely used are multiple selection and mover bars. These are controlled by the MultiSelect and MoverBars properties. Multiple selection also affects the Selected and SelectedId collections.

Multi-Select Lists

Setting MultiSelect to .T. allows users to highlight multiple items in a list at once. The usual Windows techniques (Shift-click to select a range, Ctrl-click to select individual items, Shift-Ctrl-click to select another range) work.

The Selected and SelectedId collections contain one element for each item in the list, in Index or ItemId order, respectively. When that item is highlighted, the appropriate elements in Selected and SelectedId are set to .T. Multi-select lists work much better in VFP5 than in VFP3 - a number of serious bugs have been fixed.

Mover Lists

Mover bars let users rearrange a list. When MoverBars is .T., the items in the list can be moved around with the mouse. Figure 6 shows a list of employees and their titles with MoverBars set to .T. Click on the little button next to an item and drag it to a new position.

Figure 6 - Use mover bars to change the order of items in the list.

Only lists whose data really belongs to the list pay attention to MoverBars. That is, the value of MoverBars is ignored unless the list's RowSourceType is 0 - None or 1- Value.

You can respond to items being moved because the InteractiveChange method fires when the move is finished. Unfortunately, it fires whether or not you actually move the item.

In VFP3, if you're careful, you can move items around without changing their highlight status. That is, if several items are selected, as long as you click only on the mover bars and not on the items themselves, when you're finished, the same items will be selected in their new positions. Unfortunately, a bug in VFP5 (both 5.0 and 5.0a) prevents this behavior - when you move items around, the highlight stays in the same position rather than with the same items.

Cool Stuff

The techniques above let you get your work done. The things in this section are just plain cool. You can live without them, but life's more fun with them.

Do It in Requery

It's not unusual to have a combo or list where a user entry elsewhere on the same form changes what appears. For example, Figure 7 shows a form containing a combo and a list. The user chooses a country from the combo and the suppliers in that country appear in the list. The list could be based on a query, a parameterized view or, as in this case, an array. Regardless of the RowSourceType, when the user makes a choice from the combo, we need to update the list.

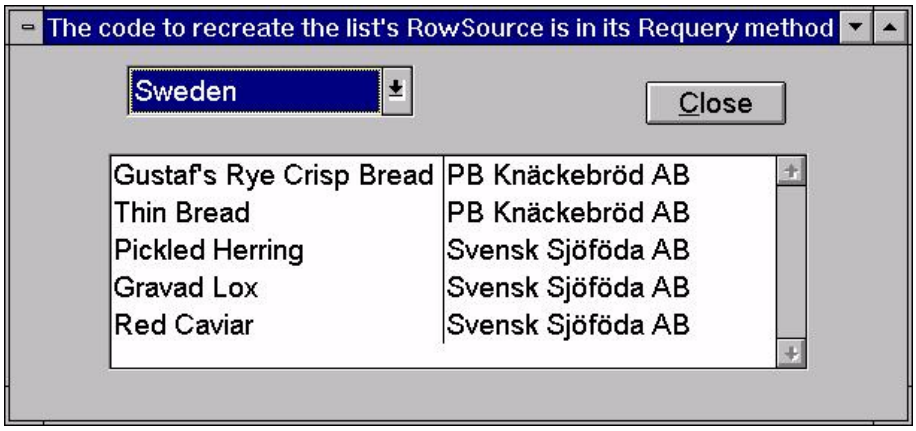

Figure 7 - The Requery method can contain the code to refill the RowSource.

If the list is based on a query (RowSourceType = 3), you can just call the Requery method for the list and you're done. But with a parameterized view, you need to call the REQUERY() function first to rerun the view. With an array, you need to refill it before you Requery the list.

Ideally, though, the combo shouldn't have to know anything about how the list works - it should be able to just call the list's Requery method and be done. In fact, you can do it that way by putting the code to repopulate the RowSource into the Requery method.

The native methods have a behavior that's just what we need in this case. They execute any code we supply first, then (assuming we don't specify NODEFAULT) perform whatever their normal, default behavior is. In the case of Requery, this means it'll run any code we put in first, then refill the list, just what we want. The Requery method for the list in Figure 7 contains this code:

```
SELECT Company_Name FROM Supplier ;
   WHERE Country = THISFORM.cboCountry.Value ;
   INTO ARRAY THISFORM.aSuppliers
```

The combo's InteractiveChange method needs only:

```
THISFORM.lstSuppliers.Requery()
```

Another benefit of this approach is that we don't have to put the code to populate the array in more than one place. The list's Init method simply calls its Requery method (which it would have to do anyway with an array), which fills the array and then fills the list.

Picture This

A picture's worth a thousand words, right? And icons are all the rage in user interfaces. So it would be really cool if you could put a little icon next to each item in a list or combo to help the user identify it. Not only can you do so, but it's really easy.

Combos and lists have a Picture property (just like buttons and checkboxes). The difference is that for combos and lists, Picture is an array capable of containing a different value for each item in the list. It gets cooler. You can address Picture either as a single value or as an array. If you assign a bitmap to Picture without specifying an element, that picture is used as a default for the list. Then, any bitmaps you specify for individual items override the default for that item only.

The biggest weakness of Picture is that you can't assign it in the Property Sheet - you have to do it in code.

Figure 8 shows a list of countries. The flag of each country is used as an icon. (The flag bitmaps come from the collection of bitmaps supplied with FoxPro 2.6 for Windows.) The fox icon is used as a default for any country for which we don't have a bitmap.

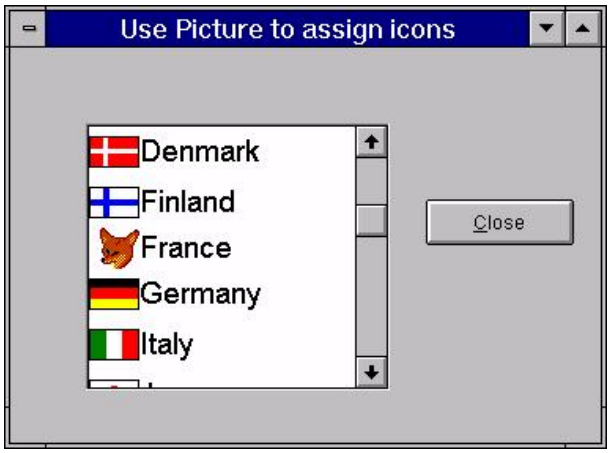

Figure 8 - The Picture property assigns icons to individual items and as a default for the list as a whole.

Picture is affected by Requery. Since Requery replaces all the list data, it's assumed that your assigned icons are no longer valid and Picture is cleared out as part of Requery's default action. This means, among other things, that you can't simply assign values to Picture in the Requery method and let Requery do its thing afterwards. You can use Requery, but it takes a little fiddling.

In the example in Figure 8, a table called flags contains the name of each country and the name of the bitmap for its flag. (We're assuming the bitmaps are somewhere in the path.) The Requery code that fills the list and assigns each item a bitmap looks like this:

```
* Get unique country names
SELECT Country FROM Supplier GROUP BY 1 INTO ARRAY THIS.aContents

* Call parent class Requery to refill list
=DODEFAULT()
* In VFP3, you need the next line instead of DODEFAULT()
* =EVAL(THIS.ParentClass+"::Requery()")
NODEFAULT

* Now add bitmap references
* first default
THISFORM.arraylist1.Picture = "Fox.BMP"

* now individuals
LOCAL nCount
FOR nCount = 1 TO ALEN(THISFORM.arraylist1.aContents,1)
   IF SEEK(UPPER(THISFORM.arraylist1.aContents[nCount]),"flags")
      THISFORM.arraylist1.Picture[nCount] = ALLTRIM(flags.bitmap)
   ENDIF
ENDFOR
```

Once we've refilled the array the list is based on, we call the parent class' Requery method to perform its default action, then issue NODEFAULT to prevent the default action that normally occurs at the end of Requery. (This is a common technique for making a default action occur earlier.) Once that happens, we can refill Picture without having it overwritten.

As in the last example, the list's Init calls Requery to populate it initially

Summary

Lists and combos, despite some shortcomings, are extremely powerful controls. The techniques in these notes should help you get the most out of them.

However, there's still more to these controls. Don't forget to check out the IncrementalSearch property, as well as the ItemData and ItemIdData collections. As always, a little experimentation is the key to producing interfaces your users will appreciate.