The Last Pick List You'll Ever Need

Doug Hennig

Last month's column looked at classes dealing with lookup tables. Closely related to lookups are pick lists. Pick lists were ubiquitous in FoxPro 2.x applications, but even with comboboxes and listboxes, are still useful in today's applications. This month, Doug presents reusable classes that provide pick list features, and expands our base class library to provide row highlighting in grids.

Why would you want to use a pick list in an application? Aren't comboboxes the proper mechanism for allowing the user to choose from a predefined list of values? Sometimes they are, but pick lists are useful in several cases:

- For heads-down data entry, comboboxes are a poor choice because you force the user to take their hands off the keyboard, click on a dropdown arrow, and then scroll through a list of choices (the incremental search feature of comboboxes is OK, but could be better). A better mechanism is to have the user type a lookup code into a field (after all, they're likely either entering information off a form which already has the code on it or they're know the list of codes well enough to just type the one they want). For example, in a call center application I wrote, the users have to enter the mode (how the call came in), subject of interest, ad campaign and magazine that sparked the call, state or province of the caller, etc. Each of these has a range of acceptable values defined in a lookup table. Since there are a relatively small set of values for each field, the users enter a mnemonic code for each value ("TX" for Texas in the case of the caller's location, "RD" for Reader's Digest in the case of magazine, and "F" for fishing in the case of subject). Typing a code into a textbox is much easier than choosing the full description from a combobox, but for new codes or inexperienced users, a picklist of values (either on demand or when an incorrect code is entered) provides the same benefit as a combobox.

- Some users like a list of records for navigation in data entry forms. Obviously, this isn't a great idea in client-server or Web applications, but against local tables, it works well. If you think about it, there isn't much difference between a navigation list and a pick list of lookup values.

- Although their performance has improved in each version of VFP, in my opinion, comboboxes and listboxes aren't the correct control when you want to display a list of a lot of records. A grid is a better choice.

The classes described in this column are available from the Subscriber Downloads site. Because some of the code is fairly lengthy, I won't list all of the methods of these classes in this article, but will discuss their important points. All the methods are well-commented, so feel free to follow this narrative along with the code listings.

SFGridTextBox Class

SFGridTextBox is a new class in SFCtrls.vcx, our base class library. It isn't actually specific for pick lists; I use it instead of the VFP base TextBox class in columns in all grids because it has a behavior I want: highlighting a complete row in a grid by setting the background color of all items in that row to a different color. This is way more obvious than lame effect you get by setting the HighlightRow property of a grid to .T. Getting this to work requires collaboration between the grid and the textbox objects in its columns, so I made some changes to SFGrid (our base grid class also in SFCtrls.vcx). Here's how this scheme works.

The Init method of SFGrid calls a new custom method, SetupColumns(). This method has the following code:

```
with This
  if .HighlightRow
    if not empty(.RecordSource)
      .ColumnGotFocus()
    endif not empty(.RecordSource)
    .nBackColor = .BackColor
```

```
    .SetAll('DynamicBackColor', ;
      'iif(recno(This.RecordSource) = This.nRecno, ' + ;
      'rgb(' + .cSelectedBackColor +'), ' + ;
      ltrim(str(.BackColor)) + ')')
  endif .HighlightRow
endwith
```

This code uses the SetAll() method to set the DynamicBackColor property of all objects in the grid to the color specified in the custom cSelectedBackColor property (I set this property to 0,255,255, which is cyan, but it could easily be set to a user's preference) for the selected row. How do we know which is the selected row? When the record number of the RecordSource for the grid matches a custom nRecno property. When does nRecno get set? Ah, that's where the collaboration comes in. I originally figured that nRecno could be set to RECNO(This.RecordSource) in the AfterRowColChange method of the grid. Unfortunately, that fires too late to affect the row coloring. I tried it in the BeforeRowColChange method, but that fires too early. The right place to do it would be when any column gets focus, but columns don't have a GetFocus method.

The solution was to set it in the GotFocus method of a textbox in a column of the grid. To avoid complicating SFTextBox (our base textbox class) with code that only matters when the textbox is in a grid, I created SFGridTextBox. This class has a few properties set so the textbox appears better in a grid (Height is 15, BorderStyle is 0, and Margin is 0). The GotFocus method of this class calls a custom ColumnGotFocus method of the grid. ColumnGotFocus simply stores RECNO(This.RecordSource) to its nRecno property.

The effect of all this is that as the user moves through the grid, the GotFocus method of the textboxes in the grid call the grid's ColumnGotFocus method, which changes nRecno to the current record number, which causes the background color of all objects in the current row to change to a defined color and the background color of all objects in other rows to return to their normal value. Thus, the entire current row is highlighted.

SFIncSearchTextBox Class

SFIncSearchTextBox is a subclass of SFGridTextBox located in SFCCtrls.vcx. This class has the following behavior:

- If the custom cDisplayTag or cSeekTag properties are filled in, the index order for the table changes to the specified tag. This has the effect of reordering the grid when the user tabs to or clicks in a column in the grid. The code from the GotFocus method that does this is shown below. You might think this should be a straightforward bit of code, but it has a complication: changing the order for the table and refreshing the grid causes the record pointer to change, because the grid wants to maintain the same relative row in its display, which may not be the same record in the table. So, this code has to save the record pointer, refresh the grid, restore the record pointer (using a custom Reposition method that ensures the record number is valid), and refreshes the grid again. Seems goofy, I know, but it works.

```
local lcOrder, ;
  lcAlias, ;
  lnRecno
dodefault()
with This
  if not empty(.cDisplayTag) or not empty(.cSeekTag)
    lcOrder = iif(empty(.cDisplayTag), .cSeekTag, ;
      .cDisplayTag)
    lcAlias = .Parent.Parent.RecordSource
    if not upper(order(lcAlias)) == upper(lcOrder)
      set order to (lcOrder) in (lcAlias)
      lnRecno = recno(lcAlias)
      Thisform.LockScreen = .T.
      .Parent.Parent.Refresh()
      .Reposition(lnRecno, lcAlias)
      .Parent.Parent.Refresh()
      Thisform.LockScreen = .F.
    endif not upper(order(lcAlias)) == upper(lcOrder)
  endif not empty(.cDisplayTag) ...
```

```
endwith
```

- It supports incremental searching in the ControlSource field if the cDisplayTag or cSeekTag properties are filled in. This is handled in the KeyPress method of the class. Why do it there rather than in the InteractiveChange method? Because we don't actually want to change the contents of the field. What we want is to allow the user to enter a string and search for records that have the current field starting with that string. So, we're going to grab the key the user pressed in the KeyPress method, add it to the end of a string we're building up (stored in the cSearchString property), do a SEEK (with SET NEAR ON) on that string to find the nearest matching record, and then issue NODEFAULT so the keypress isn't entered into the field. Of course, it's a little more complicated than that; the code has to handle things like non-character values; handle Home and End by moving to the first and last records, respectively; allow the user to hit Backspace to erase the last character they typed; and reset the search string when the _DBLCLICK time period has passed between keystrokes.

- When the user hits Enter or double-clicks on the textbox, that means they've selected a record, so we'll probably want some action taken, such as closing a pick list form. This is handled in the KeyPress method (and in DblClick by calling KeyPress as if Enter was pressed) by evaluating the contents of the cEnterAction property. For example, the SFPickList class, which we'll look at later, sets this property to Thisform.cmdOK.Click(), so pressing Enter or double-clicking in the grid acts as if the user clicked on the OK button.

The reason for having two properties that define what tag is used for the order of the table is that one (cDisplayTag) is used as the order for display purposes while the other (cSeekTag) is used for SEEKs when the user types search strings. Most of the time, these are the same, so I just fill in cDisplayTag and leave cSeekTag blank. However, I've run into situations where the searching is done on a different tag that the displaying, so these properties take care of those cases.

While SFGridTextBox is used in columns in most grids, pick list grids use SFIncSearchTextBox, even if incremental searching isn't used in a particular column. This gives us the last behavior (having an action performed on Enter or double-click). Let's take a look at a pick list grid class now.

SFPickGrid Class

SFPickGrid, a subclass of SFGrid located in SFCCtrls.vcx, is kind of a boring class. It doesn't have any code in any methods and doesn't add any new custom properties. It just has a few properties set: DeleteMark and RecordMark are .F., ReadOnly is .T., and ScrollBars is 2 - Vertical (pick lists which you have to scroll horizontally drive me crazy).

Let's see this class in action. The TESTPICK form has a pageframe with two pages: Edit, which shows a few fields from the CUSTOMER table, and List, which provides a grid for navigation of this table. The grid is an SFPickGrid bound to CUSTOMER and with three columns: one for company, one for contact, and one for city. These columns have SFIncSearchTextBox objects in them with cEnterAction set to Thisform.SetFocusToFirstObject(). This method (found in SFForm, our form base class) sets focus to the first object on the first page, meaning that when the user chooses a record by pressing Enter or double-clicking, the first page of the form is activated with fields from that record displayed. This makes for a nice, simple navigation tool. Navigation is further enhanced by setting the cDisplayTag properties of the SFIncSearchTextBox bound to the company and city fields to COMPANY and CITY, respectively. Run this form, select the List page, and notice how incremental searching works in both of these columns and that the order of the grid changes when you set focus to either column.

OK, that's fine for a navigation tool, but how do I use it for lookups? Would I have to create a custom form with a custom grid for each lookup table? Of course not; as FoxTalk editor Whil Hentzen says, "that would be too hard". Instead, we'll create a generic class for doing pick lists in their own form. All we'll have to do is instantiate this class and tell it what we want displayed.

SFPickList Class

SFPickList is a subclass of SFForm. It has an SFPickGrid with no columns defined, and OK and Cancel buttons. Its Init method does the following:

- It accepts the following parameters: tcRecordSource (the table to be displayed in the grid), taColumns (an array of information about the columns in the grid), tcCaption (the caption for the form), tcReturnExpr (an expression representing the value to return to indicate which record the user selected), tuSeek (an optional parameter that indicates which record the table should initially be positioned to), and tcSeekTag (the tag to use for the tuSeek value).

- SFPickList has a private datasession so it doesn't interfere with the datasession of the form or program that called it, so it has to open a copy of the table specified in the tcRecordSource parameter.

- It calls a custom SetupGrid method to do the dirty work of setting up the grid's columns.

- It resizes the form to fit the grid (SetupGrid sizes the grid so all columns are visible) and centers the OK and Cancel buttons.

- It sets the form's Caption to the tcCaption parameter.

- If tuSeek was passed, it positions the table to the nearest record with that value using the tcSeektag index.

The SetupGrid method accepts an array containing information about the grid's columns as a parameter. The array has one row per grid column and the following column structure: the ControlSource for the grid column is in column 1, the width is in column 2 (if it isn't specified, the grid column will be sized as appropriate for the field), the header caption is in column 3 (if it isn't specified, either the Caption property for the field in the database or a "cleaned up" field name is used), the alignment to use is in column 4 (the default alignment is used if it isn't specified), and the cDisplayTag property for the SFIncSearchTextBox is in column 5. This method adds the appropriate number of columns to the grid, sets the column properties, and creates an SFIncSearchTextBox object in each column.

The Click method of the Cancel button simply hides the form so execution returns to the code that instantiated the form. The Click method of the OK button evaluates the custom cKeyExpression property (which gets its value from the tcReturnExpr parameter passed to the class) and stores the resulting value into the uKeyValue property. This is used so the calling form can determine the key value of the record the user selected and do something appropriate with it (such as SEEKing that value and refreshing the form or a control). This method also sets the form's lSelected property to .T. so the calling form will know the user actually selected a record and didn't just click on Cancel, and then hides the form.

SFPickListButton Class

The last class we'll look at is SFPickListButton (defined in SFCCtrls.vcx), which is based on SFPictureButton (located in SFButton.vcx). The Picture property of this class contains MAGNIFY.BMP so the button looks like a "find" button. It has several custom properties: aColumns, an array of column specifications for the SFPickList class; cAlias, the alias the pick list is based on; cCaption, the caption for the pick list form; and cTag, the name of the primary tag for the pick list table.

The Click method of the class instantiates an SFPickList object and passes it the parameters it expects to do its job. Upon return, it examines the lSelected property of the SFPickList object and if it's .T., SEEKs the uKeyValue property in the pick list table and refreshes the form so any controls bound to a field in that table will show the appropriate values. Here's the code:

```
local laColumns[1], ;
  lnSelect, ;
  lcKeyExpr, ;
  luKeyValue, ;
  lcDatabase, ;
  lcAlias, ;
  loLookup
with This

* Ensure the aColumns array has been properly set up,
* the specified table is open, and the tag defined.
```

```
  assert type('.aColumns[1, 1]') = 'C' and ;
    not empty(.aColumns[1, 1]) ;
    message 'SFPickListButton: aColumns not set up'
  assert type('.cAlias') = 'C' and ;
    not empty(.cAlias) and used(.cAlias) ;
    message 'SFPickListButton: cAlias not properly ' + ;
    'specified'
  assert type('.cTag') = 'C' and not empty(.cTag) and ;
    tagno(.cTag, '', .cAlias) > 0 ;
    message 'SFPickListButton: cTag not properly ' + ;
    'specified'

* Set up the parameters we'll pass to the SFPickList
* class.

  acopy(.aColumns, laColumns)
  lnSelect = select()
  select (.cAlias)
  lcKeyExpr  = key(tagno(.cTag))
  luKeyValue = evaluate(lcKeyExpr)
  lcDatabase = cursorgetprop('Database')
  lcAlias    = iif(empty(lcDatabase), '', ;
    lcDatabase + '!') + .cAlias
  select (lnSelect)

* Create the picklist and display it.

  loLookup = newobject('SFPickList', 'SFForms.vcx', '', ;
    lcAlias, ;
    @laColumns, ;
    .cCaption, ;
    lcKeyExpr, ;
    luKeyValue, ;
    .cTag)
  loLookup.Show()

* If the user selected a record, find it and refresh
* the form.

  if type('loLookup') = 'O' and ;
    not isnull(loLookup) and loLookup.lSelected
    = seek(loLookup.uKeyValue, .cAlias, .cTag)
    Thisform.RefreshForm()
  endif type('loLookup') = 'O' ...
endwith
```

To see an example of how this button is used, look at the TESTPICK sample form. On the first page, it has an SFPickListButton object beside the COMPANY textbox. This button's cAlias property is set to CUSTOMER (normally you'd use a pick list on a lookup table rather than the main table for a form, but this is just a simple sample form), cCaption is set to "Select Customer", and cTag is set to CUST_ID. The button sets up the array of columns for the pick list with the following code in its Init method:

```
with This
  dimension .aColumns[3, 5]
  .aColumns[1, 1] = 'customer.company'
* .aColumns[1, 2] = 150
* .aColumns[1, 3] = 'Company'
  .aColumns[1, 5] = 'company'
  .aColumns[2, 1] = 'customer.city'
* .aColumns[2, 2] = 100
* .aColumns[2, 3] = 'City'
  .aColumns[2, 5] = 'city'
  .aColumns[3, 1] = 'customer.contact'
* .aColumns[3, 2] = 100
* .aColumns[3, 3] = 'Contact'
endwith
```

Uncomment the commented lines to use a fixed rather than calculated width and different captions for the pick list columns.

Run this form, click on the button, and notice the pick list form that appears. It has incremental searching on the COMPANY and CITY columns and when you press Enter or double-click in the grid or click on the OK button, the sample form displays the field for the selected record.

An obvious extension of this scheme would be to make the pick list data-driven. You could create a table that contains the information necessary for each pick list, and in the SFPickListButton class, look up the appropriate record in this table and set up the aColumns array based on the values in that record. That way, you could easily change the layout of the pick list without having to modify any code, or even allow the user to define what the pick list should look like.

Conclusion

While pick lists may not make it into every application, they still have their place in developers' toolkits. In addition to the classes involved directly with pick lists, we've expanded our ever-growing class library to provide row highlighting in grids.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997 and 1998 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). CompuServe 75156,2326 or dhennig@stonefield.com.