Custom Classes

Doug Hennig

This month's column uses great ideas from other applications to create some custom classes you can include in any application.

Last month, we looked at properties, events, and methods (PEMs) of subclasses of the Visual FoxPro base classes I've created for my use. In this article, we'll take a look at some special subclasses of these classes: an EditBox that automatically expands keywords to complete text (similar to the AutoCorrect function in Microsoft Word), a ComboBox that supports a feature similar to Quicken's "quick fill" function, and a TextBox that makes it obvious when it has focus. The great part about using custom classes like these is that they can become "black box" objects: just drop them on a form, set a few properties, and don't even worry about how they do what they do.

AutoCorrect EditBox

Several years ago, the company I worked for used a time and billing program called TBR to track consulting time and bill clients. Although it was a clunky program to use, one thing I really liked about it was the idea of dictionary codes: two letter codes that would expand to a complete phrase. Dictionary codes made data entry much faster because you could predefine common phrases, such as "met with", "to discuss", "project status", or even "Prepared system documentation for project", and then enter those phrases by just entering the two letter code. You could mix and match regular text with the dictionary codes; TBR used "@" to indicate that the next two letters were a code. For example, "@mw Bob Jones @td @ps" would expand to "Met with Bob Jones to discuss project status." The thing that was a pain about it, though, was that it wouldn't expand the text on the screen, only in printed copy (such as timesheets and invoices).

I liked this feature so much that I implemented a version of it in FoxPro 2.x. I called it a "memo dictionary" because I generally only used it in memo fields. Unlike TBR, I made the Valid clause of the memo field process the text, looking for "@" characters and expanding the codes by looking them up in a table called MEMODICT. This simple table had just two fields: CODE (the code entered by the user) and DESCRIP (the text to put in place of the code). This feature allowed the user to see their complete text as soon as they left the field. While this was an improvement, it still wasn't as handy as similar features in Microsoft Word (the AutoCorrect feature) and the Cob Editor Extensions (a FoxPro editor add-on written by Randy Wallin and Ryan Katri). What I *really* wanted was the ability to expand the code as soon as the user pressed the space bar. Unfortunately, that just was doable in FoxPro 2.x using native FoxPro code.

Then came VFP. The KeyPress event in the EditBox control provides us exactly what we need: the ability to trap every keystroke, check whether the user entered a code or not, and if so, expand it immediately. The SFMemoDictEditBox class is the result.

This class, which is contained in CONTROLS.VCX on the Developer disk, is based on the SFEditBox class I discussed last month (also in CONTROLS.VCX). I added several public properties to this class, listed in Table 1. Among other things, these properties allow you to customize the name, alias, and tag of the memo dictionary table, and the name of the field that contains the expanded text. I also added one protected property: lOpened, which is .T. if this control opened the memo dictionary table.

Table 1. SFMemoDictEditBox Public Properties.

Property Name	Purpose
cDictAlias	the alias of the memo dictionary table (default = MEMODICT)
cDictCodeChar	the character used to indicate the start of a dictionary code (default = @)
cDictField	the field containing the expansion text for the code (default = DESCRIP)
cDictFile	the name of the memo dictionary table (default = MEMODICT.DBF)
cDictTag	the tag to use for the SEEK in the memo dictionary table (default = CODE)
cExpandKeyCode	a comma-delimited list of the keypress codes used to terminate the entry of a dictionary code (default = 9,13,32,44,46,59 which is Tab, Enter, Space, comma, period, and semi-colon

The Init() method opens the memo dictionary table if necessary and sets lOpened if we did so:

```
with This
  if not empty(.cDictAlias) and ;
    not used(.cDictAlias) and ;
    ((empty(dbc()) and file(.cDictFile)) or ;
    indbc(.cDictAlias, 'Table'))
    .lOpened = .T.
    use (.cDictFile) alias (.cDictAlias) ;
      again shared in 0
  endif not empty(.cDictAlias) ...
endwith
```

The Destroy() method closes the memo dictionary table if necessary:

```
with This
  if .lOpened and used(.cDictAlias)
    use in (.cDictAlias)
  endif .lOpened ...
endwith
```

The code in the KeyPress() event checks each keypress to see if the user typed a character that could terminate a code, such as the space bar, Enter, or punctuation, and if so, calls FindCode() to see if a code was entered.

```
LPARAMETERS nKeyCode, nShiftAltCtrl
local lcKey
lcKey = ltrim(str(nKeyCode))
if lcKey $ This.cExpandKeyCodes
  This.FindCode()
endif lcKey $ This.cExpandKeyCodes
```

Valid() also calls FindCode() since the user could have entered a code as the last characters before leaving the field:

```
This.FindCode()
dodefault()
```

The real work is done in two custom public methods. FindCode() scans the content of the EditBox, starting from the current cursor position backwards to the start, to see if a "start code" character (the default is "@") was entered. If so, it calls ExpandCode() to look up the code in the memo dictionary table and substitute the expanded text for the code. Here's the code for FindCode():

```
local lnI
with This
  for lnI = .SelStart to 1 step -1
    if substr(.Value, lnI, 1) = .cDictCodeChar
      .ExpandCode(lnI, .SelStart)
      exit
    endif substr(.Value, lnI, 1) = .cDictCodeChar
  next lnI
endwith
```

Here's ExpandCode():

```
lparameters tnStart, ;
  tnEnd
local lcCode, ;
  lcExact, ;
  lcExpand
with This
  lcCode  = substr(.Value, tnStart + 1, ;
    tnEnd - tnStart)
  lcExact = set('EXACT')
  set exact on
```

```
  do case
    case empty(lcCode) or empty(.cDictAlias) or ;
      not used(.cDictAlias)
    case seek(upper(lcCode), .cDictAlias, .cDictTag)
      lcExpand  = trim(evaluate(.cDictAlias + '.' + ;
        .cDictField))
      .Value    = stuff(.Value, tnStart, ;
        tnEnd - tnStart + 1, lcExpand)
      .SelStart = .SelStart + len(lcExpand) - ;
        (tnEnd - tnStart + 1)
  endcase
  if lcExact = 'OFF'
    set exact off
  endif lcExact = 'OFF'
endwith
```

To see how this class works, run the SAMPLE1 form on the Developer disk. It uses MEMODICT.DBF as the memo dictionary table. Several codes have already been defined in this table (TD, MW, TD, and DS), but of course you can add new ones. The codes can be up to five characters long (although simply changing the size of MEMODICT.CODE will allow you to use shorter or longer ones). Try entering something like "@mw Bob Jones @td proposal" and notice that the codes are expanded as you type.

AutoFill ComboBox

More than a decade ago, I used a database program on the Macintosh called OverVUE. This program was revolutionary in many ways, but one was a feature it called "precognition". As you typed in a field that had this feature turned on, OverVUE would automatically look to see if another record had these same characters entered, and if so, fill in the rest of the field with the complete entry. This is similar to an incremental search feature, but with a twist: the characters filled in by the program are highlighted, so if the user keeps typing, the new characters replace the selected text, and the incremental search tries again. This feature, which has been implemented in other programs such as Quicken, is great for data entry of frequently used values, because the user can enter just enough characters so the value they want is filled in, and then move to the next field. An obvious use of this is a lookup field into another table.

As with the memo dictionary feature, I decided I liked this so much, I created a class to provide it. Rather than using a TextBox, though, SFAutoFillComboBox is based on the SFComboBox class I discussed last month (both classes are contained in CONTROLS.VCX on the Developer disk). I decided to use a ComboBox because this would allow the user three ways to enter a value: typing the entire string, typing just a few characters and letting the field "autofill" with the nearest value, or selecting it from the dropdown portion of the ComboBox. However, this functionality could easily be added to a TextBox class as well; this might even be a better choice if there are a lot of records in the lookup table.

The public properties I added to this class are listed in Table 2. There's also a protected property: lFoundMatch, which is .T. if an autofill match was previously found (needed so we can determine what to do if the user presses BackSpace).

Table 2. SFAutoFillComboBox Public Properties.

Property Name	Purpose
cAutoFillAlias	the alias to search for an autofill entry (only required if RowSourceType is 3-SQL Select or 4-Query)
cAutoFillTag	the tag to search on for the autofill entry (only required if RowSourceType is 2-Alias, 3-SQL Select, or 4-Query)
cAutoFillValue	the field to return for the autofill entry (only required if RowSourceType is 2-Alias, 3-SQL Select, or 4-Query)
lAutoFill	.T. to enable the autofill feature (default = .T.)
lUpper	.T. if the entered text should be upper-cased before searching for a autofill value (only required if RowSourceType is 2-Alias, 3-SQL Select, or 4-Query)

The majority of the work in this class is done in the InteractiveChange() event. It looks for the value the user entered in one of various places, depending on the RowSourceType property. For example, if

RowSourceType is 5-Array, it searches the array specified in RowSource. If it's 2-Alias, 3-SQL Select, or 4-Query and the specified cursor is indexed, it does a SEEK in the cursor specified in RowSource. Here's the code for this method:

```
#define cnBACKSPACE 127
local llFound, ;
  lnCursor, ;
  lcText, ;
  laArray[1], ;
  lnRow, ;
  lcAlias, ;
  lnPos, ;
  lcValues, ;
  lnStart, ;
  lnI, ;
  lcValue

* If we're doing autofill, save the current cursor
* position and get the current text entry.

llFound = .F.
with This
  if .lAutoFill and .Style = 0
    lnCursor = .SelStart
    lcText   = alltrim(.Text)

* If we previously found a matching item and the user
* presses backspace, we need to manually remove the
* last character.

    if lastkey() = cnBACKSPACE and .lFoundMatch and ;
      not empty(lcText)
      lnCursor = lnCursor - 1
      if lnCursor < 1
        store '' to lcText, .DisplayValue
      else
        lcText = left(lcText, lnCursor)
      endif lnCursor < 1
    endif lastkey() = cnBACKSPACE ...

* If the user entered something, let's try to find a
* match.

    if not empty(lcText)
      do case

* If the combobox is based on an array, look for the
* text in the array.

        case .RowSourceType = 5
          = evaluate('acopy(' + .RowSource + ', ;
            laArray)')
          lnRow   = .ArrayScan(@laArray, lcText, 1)
          llFound = lnRow > 0
          if llFound
            .DisplayValue = laArray[lnRow, 1]
          endif llFound

* If the combobox is based on an alias or an indexed
* SQL select, seek for the value based on the
* specified tag and return the specified search value.

        case inlist(.RowSourceType, 2, 3, 4) and ;
          not empty(.cAutoFillTag) and ;
          not empty(.cAutoFillValue)
          lcAlias = iif(.RowSourceType = 2, .RowSource, ;
            .cAutoFillAlias)
          lnPos   = at('.', lcAlias)
          lcAlias = iif(lnPos = 0, lcAlias, ;
```

```
            left(lcAlias, lnPos - 1))
          lcText  = iif(.lUpper, upper(lcText), lcText)
          llFound = seek(lcText, lcAlias, .cAutoFillTag)
          if llFound
            .DisplayValue = alltrim(evaluate(lcAlias + ;
              '.' + .cAutoFillValue))
          endif found()

* If the combobox is based on a list of values, look
* for the text in the list.

        case .RowSourceType = 1 and ;
          lcText $ This.RowSource
          lcValues = alltrim(.RowSource)
          lcValues = lcValues + ;
            iif(right(lcValues, 1) = ',', '', ',')
          lnStart  = 1
          for lnI = 1 to occurs(',', lcValues)
            lnPos   = at(',', lcValues, lnI)
            lcValue = substr(lcValues, lnStart, ;
              lnPos - lnStart)
            lnStart = lnPos + 1
            llFound = lcValue = lcText
            if llFound
              .DisplayValue = lcValue
              exit
            endif llFound
          next lnI

* If the combobox is based on "none", look for the
* text in the list of added items.

        case .RowSourceType = 0
          for lnI = 1 to .ListCount
            llFound = .List[lnI] = lcText
            if llFound
              .DisplayValue = .List[lnI]
              exit
            endif llFound
          next lnI
      endcase

* Put the cursor back to its former position (changing
* the DisplayValue resets it) and highlight the text
* after the cursor position if we found a value (if we
* didn't, no text is selected text). Flag if any text
* is selected.

      .SelStart  = lnCursor
      .SelLength = iif(llFound, ;
        max(0, len(.DisplayValue) - lnCursor), 0)
    endif not empty(lcText)
    .lFoundMatch = .SelLength > 0
  endif .lAutoFill ...
endwith

* Call the AnyChange() method (which might contain some
* custom code) if we haven't already changed the value
* (which would fire the ProgrammaticChange event which
* would call the AnyChange() method).

if not llFound
  This.AnyChange()
endif not llFound
```

ArrayScan() is a protected method that searches a particular column in an array for a specified value. This is the same code that was presented in my January column on arrays, but was added here so this control is completely self-contained.

```
lparameters taArray, ;
  tuValue, ;
  tnColumn, ;
  tnOccur
external array taArray
local lnColumn, ;
  lnOccur, ;
  lnRow, ;
  lnStartElement, ;
  lnFound, ;
  lnColumns, ;
  lnElement, ;
  lnCol
lnColumn       = iif(type('tnColumn') = 'N', tnColumn, 1)
lnOccur        = iif(type('tnOccur')  = 'N', tnOccur,  1)
lnRow          = 0
lnStartElement = 1
lnFound        = 0
lnColumns      = alen(taArray, 2)

* Use ASCAN to find the value in the array, then
* determine if it's in the correct column. If not,
* change the starting element number and try again.

do while .T.
  lnElement = ascan(taArray, tuValue, lnStartElement)
  if lnElement <> 0
    lnCol = iif(lnColumns > 1, ;
      asubscript(taArray, lnElement, 2), 1)
    if lnCol = lnColumn and (type('tuValue') <> 'C' or ;
      taArray[lnElement] = tuValue)
      lnFound = lnFound + 1
      if lnFound = lnOccur
        lnRow = iif(lnColumns > 1, ;
          asubscript(taArray, lnElement, 1), lnElement)
        exit
      endif lnCol = lnColumn ...
    endif lnCol = lnColumn ...
    lnStartElement = lnElement + 1
  else
    exit
  endif lnElement <> 0
enddo while .T.
return lnRow
```

The SAMPLE1 form on the Developer disk also has a demo of this control. This form has the CUSTOMER table from the VFP 5 sample data in its DataEnvironment; you'll need to modify the form if the sample data isn't installed or if you named the directory something other than VFP5. The SFAutoFillComboBox is bound to the COMPANY field in the CUSTOMER table. Try entering "A" (the case is important because this table is not indexed on UPPER(COMPANY)) and notice the rest of the field fills in with the name of the first company starting with "A" (Alfreds Futterkiste), and all but the first character is selected. Next enter "r" (again, the case is important), and notice that the first company starting with "Ar" (Around the Horn) is now displayed. Press BackSpace, and Alfreds Futterkiste is once again displayed.

Visible Edit TextBox

The contact manager we use at Stonefield Systems Group is GoldMine for Windows. One thing I like about its user interface is that you can very easily tell which field you're currently editing: all fields appear plain with a grey background, while the field with focus appears 3-D with a white background. No more searching around, wondering where the cursor is, especially on a laptop. I liked this feature, so I created a simple SFVisibleEditTextBox class (based on SFTextBox, both of which are in CONTROLS.VCX) that provides similar functionality. (By the way, in case you're noticing a trend here, yes, I do like to "borrow" the best features from programs I like. Yeah, that's it, borrow).

To create this class, I simply set the BackStyle property to 0-Transparent so the background color of the container shows through and SpecialEffect to 1-Plain. In the GotFocus() method of the control, I set BackStyle to 1-Opaque and SpecialEffect to 0-3D, and in LostFocus(), I set them back to the original values.

To see this class in use, run the SAMPLE2 form on the Developer disk. As with SAMPLE1, this form includes the VFP sample CUSTOMER table in its DataEnvironment, so you may need to modify the form if necessary. Notice only the field with focus has a white background and appears in 3-D. It's very obvious where the cursor is located now.

Conclusion

I love the object-oriented nature of Visual FoxPro. It allows us to subclass common controls like TextBoxes and ComboBoxes and add new functionality to them, then forget about how the new features were implemented. I hope you'll find the classes described in this article as useful as I have.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Sask., Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. 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 user groups and regional conferences all over North America. CompuServe 75156,2326.