September, 2002

Advisor Answers

Expand an Editbox

Visual FoxPro 7.0/6.0/5.0/3.0

Q: I have a busy form where I need to include a memo field. I'd like to create a one-line editbox and give the user a way (such as double-clicking on it) to open another form with more room for editing. How can I do that?

–Name withheld

A: There are really two parts to the answer to your question. Part 1 is creating an editbox class that supports zooming; part 2 is creating the zoomed form. Let's take the editbox class first.

There are two issues to resolve with the class: how to specify the form to be opened, and how to trigger the zoom. To make the class the most useful, we don't want to lock in one approach for either.

VFP supports both .SCX-based forms and .VCX-based form classes. We can allow our editbox to use either one (as long as it's modal) by adding three properties: cForm, cFormClass and cFormClassLib. For any instance, you specify either cForm (for an .SCX-based form) or cFormClass and cFormClassLib (for a .VCX-based form class).

As for triggering the zoom, the class gains flexibility if we create a custom method for this purpose. Call it Expand, and put this code in it:

```
* Open a new form, containing this editbox's contents.

DO CASE
CASE NOT EMPTY(This.cForm)
   * A modal form was specified.
   DO FORM (This.cform) WITH This
CASE NOT EMPTY(This.cformclass) AND ;
     NOT EMPTY(This.cformclasslib)
   * A form class was specified.
   oEditForm = NEWOBJECT( ;
     This.cFormClass, This.cFormClassLib, "", This )
   oEditForm.Show(1)
OTHERWISE
   * No form specified. Can't do anything
ENDCASE

RETURN
```

Now, you can call the Expand method from the event for whatever action you want to have open the zoom form. For example, to have a double-click zoom the editbox, put:

```
This.Expand()
```

in the DblClick method. In the edtExpand class on this month's Professional Resource CD (PRD), I took this one step farther and added another property, lExpandOnDblClick, to the class. The DblClick method contains this code:

```
* If indicated, expand the editbox
IF This.lExpandOnDblClick
      This.Expand()
ENDIF
```

This means you can set a single property to determine whether double-click is a trigger. For my own applications, I'd be more inclined to trigger the zoom from a context menu, but having double-click available makes it easy to test the editbox without having to create a context menu.

Now, we move on to the second part of the problem, creating a modal form containing the zoomed editbox, and connecting it to the data from the original editbox. This month's PRD contains both frmEditBox.SCX, a form that serves this purpose, and a class called frmEditBox, which is identical, but stored in the class library with the editbox class. The form (shown in Figure 1) contains three controls, an editbox and two buttons.

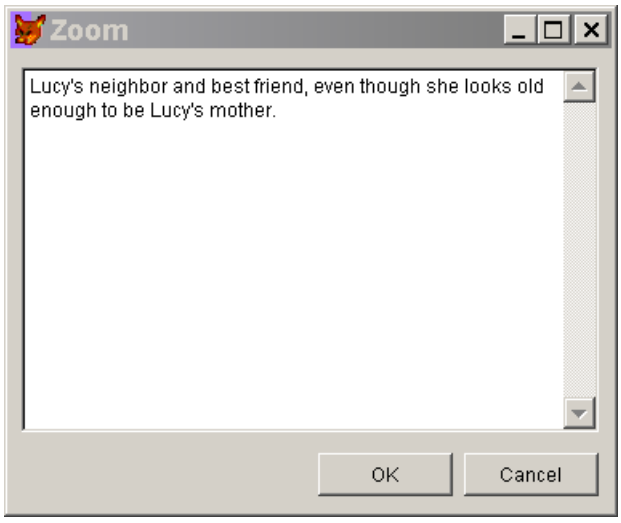

Figure 1. Zoomed editbox – this form is called from the Expand method of the editbox to provide plenty of room for information.

The form has code for two purposes. Code in the Init and Destroy methods works together to transfer the data from the calling editbox to the zoomed editbox and back. Two custom properties are used by this code. oSourceEditbox contains an object reference to the calling editbox. lKeepChanges indicates whether the user's changes should be passed back to the calling editbox; it's set by code in the Click method of the OK and Cancel buttons.

In addition, there's code to allow the user to stretch and shrink the form. Three custom properties are involved:

nHorizDistance contains the horizontal distance between the editbox and the form boundary.

nVertDistance contains the vertical distance between the bottom of the buttons and the bottom of the form.

nVertBetween contains the vertical distance between the bottom of the editbox and the top of the buttons.

All three measures are kept constant when the form is resized. This has the effect of changing only the size of the editbox.

All three properties are set by the Init code. Here's the whole Init method:

```
LPARAMETERS oOriginalEditBox, lDontAutoCenter

This.oSourceEditBox = oOriginalEditBox
```

```
This.edtExpanded.Value = This.oSourceEditBox.Value

* Center the editbox horizontally on the form
* Store the vertical heights

This.nHorizDistance = ;
   INT((This.Width - This.edtExpanded.Width)/2)
This.edtExpanded.Left = This.nHorizDistance

This.nVertDistance = INT((This.Height - ;
   (This.cmdOK.Top + This.cmdOK.Height))/2)
This.nVertBetween = This.cmdOK.Top - ;
   (This.edtExpanded.Top + This.edtExpanded.Height)

* Center the form?
This.AutoCenter = NOT lDontAutoCenter

RETURN
```

By default, the form is centered at start-up. An optional parameter allows you to turn that behavior off.

The Destroy is quite simple. If the user closed the form with the OK button, the changes are copied back to the original control.

```
IF This.lKeepChanges
   This.oSourceEditBox.Value = This.edtExpanded.Value
   This.oSourceEditBox = .NULL.
ENDIF
```

The Resize method contains code to respond to a resize of the form, as follows:

```
* Resize the editbox and adjust the buttons
* when the form is resized
LOCAL nHorizBetween

This.edtExpanded.Width = MAX(10, This.Width - ;
                         2*This.nHorizDistance)
This.cmdOK.Top = This.Height - This.nVertDistance - ;
                 This.cmdOK.Height
This.cmdCancel.Top = This.cmdOK.Top
This.edtExpanded.Height = MAX(10, This.cmdOK.Top - ;
                          This.nVertBetween - ;
                          This.nVertDistance)

nHorizBetween = This.cmdCancel.Left - ;
               (This.cmdOK.Left + This.cmdOK.Width)
This.cmdCancel.Left = This.Width - ;
                      This.nHorizDistance - ;
                      This.cmdCancel.Width
This.cmdOK.Left = This.cmdCancel.Left - ;
                  nHorizBetween - ;
```

```
This.cmdOK.Width
```

As in the class, the form was designed for flexibility. For example, the lKeepChanges property makes it easy to change the ways in which a user can accept or reject his changes.

To connect this form to the editbox class, set the cForm property of the editbox to "frmEditBox." To work with the form class instead, reset cForm to its default and set cFormClass to "frmEditBox" and cFormClassLib to "EditBoxes.VCX".

Although I've mentioned flexibility a couple of times here, you might actually want to go even farther in this direction. For example, you might choose to put the expand functionality of the editbox in a separate class that you instantiate only when it's needed. You might also want to generalize the zoom form so that it can be used in other ways. As it is now, it'll work with any control that has a Value property, but by passing the value itself and returning the changed value (say, through a parameter object), you can make it useful in other situations.

This month's PRD also contains a form, ZoomEditBox.SCX, to demonstrate the zooming editbox. Put both forms and the class library into the same directory. The example form uses the double-click trigger.

–Tamar