Create Modern Interfaces With VFP 7

Doug Hennig

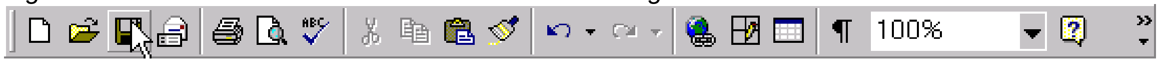
It seems that every new version of Microsoft Office changes user interface standards. Whether you like it or not, your users expect your applications to keep up with this ever-moving target. Fortunately, VFP 7 adds new features that make it easier to create interfaces similar to Office 2000.

In addition to new language features (see my "Language Enhancements in VFP 7" series of articles in the January to June 2001 issues of FoxTalk), database events, support for COM+, Web Services, and a ton of other new features, VFP 7 includes some user interface improvements, including hot tracking and modern-looking toolbars and menus.

Hot Tracking

Hot tracking means controls appear flat (rather than the 3-dimensional appearance we're used to) but change appearance as the mouse pointer moves over them. Most controls will then appear sunken (the way they normally appear with hot tracking off), except for check boxes, option buttons, and command buttons, which appear raised. For an example of hot tracking, look at the toolbars in Microsoft Office 2000 applications. As you can see in Figure 1, toolbar controls appear flat (for example, the command buttons have no outlines) until you move the mouse over them.

Figure 1. Microsoft Office 2000 toolbars use hot tracking.



Hot tracking is easy to turn on in VFP 7: simply set the SpecialEffect property to 2 (for check boxes and option buttons, you also have to set Style to 1-Graphical). For control classes that may have to be used in earlier versions of VFP, you should set this property programmatically (such as in the Init method) rather than in the Property Window to prevent an error when the control is used in those versions. Here's an example (taken from SFToolbarButton in SFBUTTON.VCX):

```
if clVFP7ORLATER
  This.SpecialEffect = 2
endif clVFP7ORLATER
```

clVFP7ORLATER is a constant defined in SFCTRLS.H, the include file for SFToolbarButton, as follows:

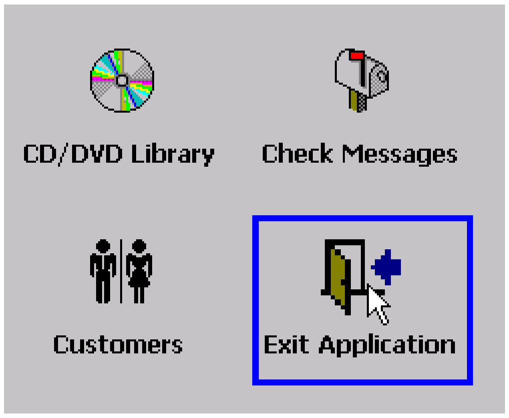
```
#define clVFP7ORLATER (type('version(5)') <> 'U' and ;
  evaluate('version(5)') >= 700)
```

Since VERSION(5) was added in VFP 6, the TYPE() test and use of EVALUATE() in this statement ensure it will work even in VFP 5.

You can create other types of effects with code in the new MouseEnter and MouseLeave events. For example, you can set This.FontBold = .T. in MouseEnter and This.FontBold = .F. in MouseLeave to make a control appear bolded when the mouse is over it. You can also change the foreground or background color, or do pretty much anything else you want in these events.

SwitchboardButton in MYCLASSES.VCX is an example. It's used as a button in "switchboard" forms, forms that provide a quick access to the major functions of an application. In VFP 7, as the user moves the mouse pointer around the form, the SwitchboardButton object under the mouse is surrounded with a blue outline (see Figure 2 for an example). SwitchboardButton is actually a container class with an image and a label. Its BorderColor is set to 0, 0, 255 (blue) and its Init method sets the BorderWidth to 0 (it's left at the default of 1 in the Property Window so you can see it in the Class or Form Designers). The MouseEnter event sets BorderWidth to 3 and MouseLeave sets it back to 0.

Figure 2. The SwitchboardButton class shows an example of hot tracking using MouseEnter and MouseLeave.



In addition to the SpecialEffect property and MouseEnter and MouseLeave events, command buttons have a new VisualEffect property. This property, which is read-only at design time, allows you to programmatically control the raised or sunken appearance of the control at run time. Although you won't often use this, it's handy when several buttons should change appearance as a group. We'll see an example of that later.

Although you can use hot tracking wherever you want, I personally don't care for hot tracking except in controls in toolbars (none of the dialogs in Microsoft Office use hot tracking, for example). So, rather than setting SpecialEffect to 2 in my base classes (those in SFCTRLS.VCX), I'll do it in specific subclasses that I use for toolbars.

To see an example of hot tracking for different types of controls, run TESTHOTTRACKING.SCX and see what happens as you move the mouse over each control.

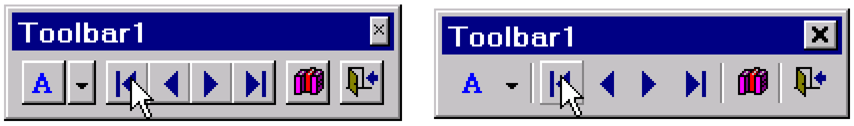
Toolbars

Like other "modern" applications, toolbars in VFP 7 now have a vertical bar at the left edge when docked to provide a visual anchor to grab to move or undock the toolbar (see Figure 1). Another improvement related to toolbars is the addition of a Style property to the Separator base class; setting this property to 1 makes a Separator appear as a vertical bar at run time (at design time, Separators are still invisible, which is kind of annoying). As with hot tracking, you might want to set this property programmatically to prevent problems with earlier versions of VFP; I use the following code in the Init method of SFSeparator (in SFCTRLS.VCX):

```
if clVFP7ORLATER
  This.Style = 1
endif clVFP7ORLATER
```

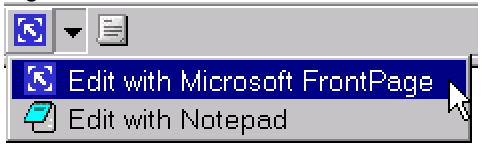
Figure 3 shows the same toolbar running in VFP 6 and 7. The VFP 7 version looks and acts like a toolbar in a more modern application.

Figure 3. The same toolbar in VFP 6 (left) and 7 (right).



A new style of toolbar button showing up in more and more applications is the dual button/menu control. Figure 4 shows an example of such a button, taken from Internet Explorer 5.5. Clicking on the left part of the control (the button with the image) causes an action to occur, while clicking on the down arrow displays a drop down menu of choices. Another place I've seen such a control used is in West Wind Technologies' HTML Help Builder to open help projects. Clicking on the button displays an Open File dialog while clicking on the down arrow displays a "most recently used" (or MRU) list of files. The advantage of this control is that it doesn't take up much screen real estate, yet it can have a large list of choices.

Figure 4. Dual button/menu controls are becoming more popular.



SFBUTTON.VCX has a couple of classes used to create such a control. SFDropDownMenuTrigger is a subclass of SFToolbarButton that's sized appropriately and displays a down arrow (Caption = "6", FontName = "Webdings", FontSize = 6). It also has assign methods on its FontName and FontSize properties so they aren't inadvertently changed programmatically by something like SetAll(). SFDropDownMenuButton is based on SFContainer, our container base class in SFCTRLS.VCX, and it contains an SFToolbarButton object named cmdMain and an SFDropDownMenuTrigger object named cmdMenu. The MouseEnter and MouseLeave events of each button set the VisualEffect property of the other button to 1 and 0, respectively, so the buttons' hot tracking are synchronized. The Click event of cmdMain calls the ButtonClicked method of the container, which is empty since this is an abstract class and the desired behavior must be coded in a subclass or instance. The MouseDown event of cmdMenu has the following code to display the dropdown menu:

```
lparameters tnButton, ;
  tnShift, ;
  tnXCoord, ;
  tnYCoord
local loObject
with This
  do case
    case not clVFP7ORLATER

* If the menu was displayed and we clicked on this
* button again, re-enable the raised visual effect.

    case .VisualEffect = 0
      .VisualEffect = 1
      .Parent.cmdMain.VisualEffect = 1
      return

* Turn on the sunken visual effect.

    case .VisualEffect = 1
      .VisualEffect = 2
  endcase

* Display the menu.

  .Parent.lMenuActive = .T.
  .Parent.ShowMenu()
  .Parent.lMenuActive = .F.

* Turn off the visual effect for this button and the
* other one if the mouse isn't over this button (this
* prevents flicker if the user clicks this button again
* to hide the menu).

  if clVFP7ORLATER
    .VisualEffect = 0
    loObject = sys(1270)
    if vartype(loObject) <> 'O' or ;
      not loObject.Name == This.Name
      .Parent.cmdMain.VisualEffect = 0
    endif vartype(loObject) <> 'O' ...
  endif clVFP7ORLATER
endwith
```

Since SFContainer already has methods and code for handling shortcut menus (see my column in the February 1999 issue of FoxTalk, “A Last Look at the FFC”), why reinvent the wheel? As a refresher, the ShowMenu method of SFContainer instantiates an SFShortcutMenu object (defined in SFMENU.VCX), which is an adaptation (not subclass) of the FFC _ShortcutMenu class. SFShortcutMenu handles all of the work of displaying a shortcut menu; you just call the AddMenuBar and AddMenuSeparator methods to define the bars in the menu, then call the ShowMenu method to display it. SFContainer.ShowMenu calls the ShortcutMenu method to do the actual work of defining the bars (that method is abstract in SFContainer).

However, one issue SFDropDownMenuButton has to address that SFContainer doesn’t is menu placement. SFShortcutMenu automatically places the menu at the current mouse position, but if you look at Figure 4, you’ll notice the menu appears directly below the control, aligned with its left edge. To support that, I added nRow and nCol properties to SFShortcutMenu so you can control the position of the menu; if they contain 0, which they do by default, SFShortcutMenu will figure out where the menu should go, so the former behavior is maintained. The ShortcutMenu method of SFDropDownMenuButton, however, has to place the menu at the right spot, so it calculates the appropriate values for the nRow and nCol properties.

What’s the right spot? That depends on if and where the toolbar hosting the control is docked. If the toolbar is docked at the right or bottom edges, the menu has to be placed to the left or above the control so it appears inside the VFP window. Otherwise, it has to be placed below and at the left edge of the control. The code to perform these calculations is fairly long and complex (I adapted, okay, ripped off <g>, the code from NEWTBARS.VCX in the SOLUTION\SEDONA subdirectory of the VFP samples directory), so it isn’t shown here.

To use SFDropDownMenuButton, drop it or a subclass on a toolbar. To see an example, look at the instance named ColorPicker in the MyToolbar class in MYCLASSES.VCX, included with this month’s Subscriber Downloads. ColorPicker is just a simple demonstration of this control; it allows the user to change the background color of the active form from either a pre-selected list of colors (the drop down menu) or a color dialog (when you click on the button). The ButtonClicked method, called when the user clicks the button, displays a color dialog and sets the background color of the active form to the selected color:

```
_screen.ActiveForm.BackColor = ;
  getcolor(_screen.ActiveForm.BackColor)
```

The ShortcutMenu method has the following code:

```
lparameters toMenu, ;
  tcObject
toMenu.AddMenuBar('Red', ;
  '_screen.ActiveForm.BackColor = rgb(255, 0, 0)')
toMenu.AddMenuBar('Green', ;
  '_screen.ActiveForm.BackColor = rgb(0, 255, 0)')
toMenu.AddMenuBar('Blue', ;
   '_screen.ActiveForm.BackColor = rgb(0, 0, 255)')
toMenu.AddMenuBar('Grey', ;
  '_screen.ActiveForm.BackColor = rgb(212, 208, 200)')
dodefault(toMenu, tcObject)
```

toMenu is a reference to the SFShortcutMenu object. The first parameter for the AddMenuBar method is the prompt for the bar and the second is the command to execute when that bar is chosen.

Menus

Modern applications usually provide many different ways to perform the same action: main menu selections, toolbar buttons, shortcut menu selections, and so on. We’ve already discussed toolbars, and the SFShortcutMenu class makes it easy to create shortcut menus for every form and object in your application. So, let’s talk about the main menu.

Menus haven’t changed much in FoxPro since FoxPro 2.0 (although in my August 2001 column, “Objectify Your Menus”, I presented a set of classes that make it easy to create object-oriented menus).

New in VFP 7, however, are the abilities to specify pictures for bars (either the picture for a VFP system menu bar or a graphic file) and to create inverted bars that only appear when the user clicks on a chevron at the bottom of a menu popup ("MRU" menus, although the meaning of MRU here is different than I used it earlier). These features allow us to create Office 2000-style menus.

Specifying a picture is easy. In the VFP Menu Designer, click on the button in the Options column for a menu bar, and in the Prompt Options dialog, select File if you want to specify a graphic file or Resource if you want to use the picture for a VFP system menu bar. If you select File, you can either enter the name of the file in the picture textbox or click on the button beside the textbox and select it from the Open File dialog. If you chose Resource, either enter the name of the VFP system menu bar (for example, "_mfi_open") or click on the button and select it from the dialog showing the prompts of system menu bars. In either case, a preview of the picture is shown in the Prompt Options dialog. The settings result in the PICTURE or PICTRES clauses being added to the DEFINE BAR command that will ultimately be created for this bar. If you're using the OOP menus I presented in August, set either the cPictureFile or cPictureResource property of an SFBar object to the desired value.

The MRU feature is more difficult to use, and much more difficult to implement in a practical manner. The DEFINE BAR command has new MRU and INVERT clauses, but because there are no specific options for either clause in the Menu Designer, you end up having to use a trick: enter ".F." followed by either "MRU" or "INVERT" in the Skip For option for the bar. VFP 7's menu generator, GENMENU.PRG, is smart enough to see that you're really use the Skip For setting as a way of sneaking other clauses into the DEFINE BAR command that the generator will create, so it leaves off the SKIP FOR .F. part of the command.

However, that's only the beginning. You're responsible for managing what happens when the user selects the MRU bar (the chevron at the bottom of the menu) yourself. Typically, you'll remove the MRU bar from the menu and add bars with the INVERT clause to the menu, but since the Menu Designer doesn't create those bars for you, you have to code the DEFINE BAR statements yourself (although you could create the desired bar in the Menu Designer, generate the MPR file, copy the DEFINE BAR statement for the bar from the MPR, then remove it in the Menu Designer). Also, once the user has selected one of the inverted bars, you have to add the MRU bar back to the menu and remove the inverted bars, except perhaps the selected one, which you may decide to leave in the menu as Office applications do, although then you have the complication of changing it from an inverted bar to a normal one and *not* adding that bar the next time the user selects the MRU bar, and then that'll only last until the user exits the application. See what I mean by "much more difficult to implement in a practical manner?"

I can't think of any application I've written in the past 20 years that was complex enough to actually use this type of MRU feature, but at least the OOP menu classes I presented in August manage a lot of this stuff for you. Set the lMRU property of an SFPad object to .T. if that pad should have an MRU bar in it, and set the lInvert property of any SFBar object to .T. to have that bar appear when the MRU bar is selected and disappear after a menu selection is made. You'll have to subclass SFPad if you want different behavior, such as changing an inverted bar into a normal one if it's selected.

A more useful version of an MRU feature is the one I referred to earlier: a list of things the user has accessed recently. Office 2000 applications use this: the bottom of the File menu shows a list of the most recently accessed documents. Most VFP applications don't use the concept of "documents", but they do use records. It might make sense in some applications to put the most recently accessed records at the bottom of a menu so a user can quickly return to a record they were working with before. Rather than automatically doing that, you might want to provide a function the user can select to add the current record to the MRU list.

The sample application included with this month's Subscriber Downloads has an example of such a feature. First, a button in the Mytoolbar class, used as the toolbar for the customers form, allows the user to "bookmark" the current record; it does so by calling the Bookmark method of the active form. That method in CUSTOMERS.SCX has the following code:

```
lcCommand = [iif(type('_screen.ActiveForm.Name') = ] + ;
  ['C' and _screen.ActiveForm.Name = ] + ;
  ['frmCustomers', _screen.ActiveForm.Seek('] + ;
  CUSTOMER.CUST_ID + ['), oApp.DoForm('Customers ] + ;
  [with "] + CUSTOMER.CUST_ID + ["'))]
lcCaption = trim(CUSTOMER.COMPANY)
```

```
oBookmark.AddBookmark(lcCommand, lcCaption)
```

This code expects that the Bookmark class, which we'll look at in a moment, has been instantiated into a global variable called oBookmark. The AddBookmark method of that class expects two parameters: the command to execute when the bookmark is selected and the caption for the bookmark. In this case, the command tells VFP that if the active form is the customers form, call the Seek method of that form with the customer's CUST_ID value (that method positions the form to the specified key value); if there is no active form or it isn't the customers form, call the DoForm method of the application object, telling it to run the customers form and passing the CUST_ID value (the Init method of the customers form accepts an optional CUST_ID value and calls the Seek method if it's passed). The company name is used as the caption for the bookmark.

The Bookmark class, in MYCLASSES.VCX, is a simple class based on SFCustom. It has a two-dimensional array called aBookmarks to store the bookmarks; the first column is the command to execute and the second is the caption. The nMaxBookmarks property determines how many bookmarks can be stored. The AddBookmark method adds a bookmark to the array and to the bottom of the File menu. Here's the code:

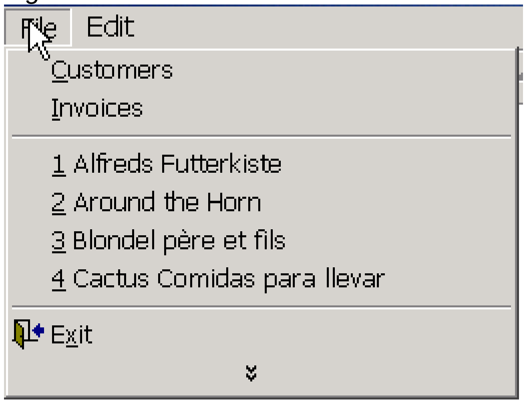
```
lparameters tcFunction, ;
  tcCaption
local lnRows, ;
  loSeparator, ;
  lcBar, ;
  loBar

* Find the next available slot for the bookmark. If we've
* exceeded the maximum number of bookmarks, exit.

with This
  lnRows = iif(empty(.aBookmarks[1]), 1, ;
    alen(.aBookmarks, 1) + 1)
  if lnRows > .nMaxBookmarks
    return .F.
  endif lnRows > .nMaxBookmarks
  dimension .aBookmarks[lnRows, 2]
  .aBookmarks[lnRows, 1] = tcFunction
  .aBookmarks[lnRows, 2] = tcCaption
endwith

* If this is the first slot, add a separator bar above
* the Exit bar in the File menu.

if lnRows = 1
  loSeparator = oMenu.FilePad.AddBar('SFSeparatorBar', ;
    'SFMenu.vcx', 'FileBookmarkSeparator')
  with loSeparator
    .cBarPosition = 'before ' + ;
      transform(oMenu.FilePad.FileExit.nBarNumber)
    .Show()
  endwith
else
  loSeparator = oMenu.FilePad.FileBookmarkSeparator
endif lnRows = 1

* Add the bookmark to the File menu above the separator
* bar.

lcBar = 'FileBookmark' + transform(lnRows)
loBar = oMenu.FilePad.AddBar('SFBar', 'SFMenu.vcx', ;
  lcBar)
with loBar
  .cCaption        = '\<' + transform(lnRows) + ' ' + ;
    tcCaption
  .cOnClickCommand = tcFunction
  .cBarPosition    = 'before ' + ;
    transform(loSeparator.nBarNumber)
  .Show()
```

```
endwith
```

Before the first bookmark is added to the File menu, a separator bar is added above the Exit bar. Then, bars for the bookmarks are added above that separator. The cBarPosition property is used to control the bar positions.

Figure 5 shows an example of the File menu after I bookmarked four records. Selecting a bookmark opens the customers form (if necessary) and displays the chosen record.

Figure 5. Records are bookmarked near the bottom of the File menu.



The Bookmark class has a couple of other methods, SaveBookmarks and RestoreBookmarks, that save and restore the bookmarks, using BOOKMARKS.DBF. These methods ensure the user's bookmarks are persistent between application sessions.

Tying it All Together

The sample application shows all of the techniques discussed in this article. DO MAIN to start the application. MAIN.PRG instantiates some objects, including a simple application object and the Bookmark class, and creates a menu for the application. It then runs the SWITCHBOARD form and issues a READ EVENTS. The only functions in the menu and switchboard that do anything are Customers (which runs CUSTOMERS.SCX) and Exit.

The switchboard form uses the SwitchboardButton class mentioned earlier to show hot tracking. The menu shows the use of MRU and inverted bars, includes pictures for some bars, and demonstrates the use of most recently used (bookmarked) records. The customers form isn't fancy, but the toolbar it uses shows the SFDropDownMenuButton class (as a color picker), includes a button to bookmark the current record, and demonstrates the new features of VFP 7 toolbars, including buttons with hot tracking and vertical separator bars.

VFP 7 has several new features that make it easier to create applications that look and act like Office 2000. Of course, Office XP raises the bar yet again, but for now, our applications can look more modern than VFP 6 applications could.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), co-author (along with Tamar Granor and Kevin McNeish) of "What's New in Visual FoxPro 7.0" from Hentzenwerke Publishing, and author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's Pros Talk Visual FoxPro series. He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals", both from Hentzenwerke Publishing, Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com Email: dhennig@stonefield.com