Developing Modern Interfaces With VFP 7

Doug Hennig
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Voice: 306-586-3341
Fax: 306-586-5080
Email: dhennig@stonefield.com
Web site: www.stonefield.com

Overview

VFP 7 introduces some new features that allow us to create applications with up-to-date interfaces (such as that in Office 2000). However, there's a lot more to creating an exciting interface than just turning on "hot tracking". In this document, you'll learn how to create modern-looking menus (complete with Most Recently Used features), toolbars (including command buttons with drop-down menus), dialogs (using new Windows 2000 features), and forms. These techniques will help you freshen the user interface of your applications to give them a longer shelf life.

Introduction

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.

To help make your application look and behave more modern, we'll look at shortcut menus, hot tracking, toolbars, application menus, dialogs, and browser forms.

Shortcut Menus

Shortcut menus make it easier for a user to select a function because they don't have to move the mouse all the way to the menu bar and because the list of items in the shortcut menu is usually context-sensitive, so they don't have to figure out which functions are applicable right now. Shortcut menus are one of the things that make an application look more professional.

Although there's nothing new in VFP 7 for shortcut menus, we'll discuss them first, because they're easily implemented using the classes I'll show, and because some of the new features we'll discuss later will use these menu classes.

The FoxPro Foundation Classes (FFC) that come with VFP include a class named _ShortcutMenu (in _MENU.VCX) that's an object-oriented wrapper for VFP's non-object-oriented shortcut menu system. It's one of the most useful classes in the FFC because it provides a simple way to programmatically (and therefore dynamically) create shortcut menus. This class has the following methods:

AddMenuBar: adds a new bar to the menu. You can specify the prompt and optionally the code to execute when the bar is selected (if nothing is specified, the code in the cOnSelection property is executed) or another menu object to display as a submenu, other clauses for the bar (such as a SKIP FOR clause), the location of the bar in the menu, whether the bar is marked or not, whether the bar is disabled or not, and whether the bar appears in bold or not.

AddMenuSeparator: adds a separator bar to the menu.

ShowMenu: displays the menu, executes the appropriate action for the selected bar, and then hides the menu.

ClearMenu: removes all bars from the menu.

NewMenu: instantiates another _ShortcutMenu object without having to use CREATEOBJECT() or NEWOBJECT(). This is frequently used when a menu bar has a submenu.

_ShortcutMenu is pretty good, but it has some limitations, the biggest of which is that the parameters passed to AddMenuBar are static; they can't be expressions evaluated when the menu is displayed. This means that once you've defined a menu, you have to clear it and redefine all the bars if you want to change its appearance in some way. So, I created SFShortcutMenu (in SFMENU.VCX), which I use instead. Because of the way it was designed, it wasn't feasible to subclass _ShortcutMenu (that would've also added some baggage to any project using such a subclass), so I stole, oops, I mean adapted, the code in _ShortcutMenu to create SFShortcutMenu.

I like SFShortcutMenu so much that I added hooks for it in nearly every class in SFCTRLS.VCX (my base class library). I added an oMenu property to contain an object reference to an SFShortcutMenu object and an lUseFormShortcutMenu property, which contains .T. if the menu of the form should be included in the menu for an object on that form. I made the Destroy method set oMenu to NULL to avoid dangling object references. I added a ShowMenu method (which is called from RightClick) that instantiates an SFShortcutMenu object, calls the custom ShortcutMenu method to populate it, calls the ShortcutMenu method of a hooked object (if there is one) to add to the menu, calls the ShortcutMenu method of the form the control is on (if there is one and lUseFormShortcutMenu is .T.) to add to the menu, and finally displays the menu if it has any bars. Here's the code for ShowMenu:

```
private loObject, ;
  loHook, ;
  loForm
with This

* Define reference to objects we might have menu items
* from in case the action for a bar is to call a method
* of an object, which can't be done using "This.Method"
* since "This" isn't applicable in a menu.

  loObject = This
  loHook   = .oHook
  loForm   = Thisform

* Define the menu if it hasn't already been defined.

  if vartype(.oMenu) <> 'O'
    .oMenu = MakeObject('SFShortcutMenu', 'SFMENU.VCX')
  endif vartype(.oMenu) <> 'O'
  if vartype(.oMenu) = 'O'

* If there aren't any bars in the menu, have the
* ShortcutMenu method populate it.

    if .oMenu.nBarCount = 0
      .ShortcutMenu(.oMenu, 'loObject')

* Use the hook object (if there is one) to do any further
* population of the menu.

      if vartype(loHook) = 'O' and ;
        pemstatus(loHook, 'ShortcutMenu', 5)
        loHook.ShortcutMenu(.oMenu, 'loHook')
      endif vartype(loHook) = 'O' ...

* If desired, use the form's shortcut menu as well.

      if .lUseFormShortcutMenu and ;
        type('Thisform.Name') = 'C' and ;
        pemstatus(loForm, 'ShortcutMenu', 5)
```

```
        loForm.ShortcutMenu(.oMenu, 'loForm')
      endif .lUseFormShortcutMenu ...
    endif .oMenu.nBarCount = 0

* Activate the menu if necessary.

    if .oMenu.nBarCount > 0
      .oMenu.ShowMenu()
    endif .oMenu.nBarCount > 0
  endif vartype(.oMenu) = 'O' ...
endwith
```

Note the use of PRIVATE, rather than LOCAL, variables. They define references to objects in case the action for a bar is to call a method of the object. You can't do this using code like "This.Method()" because "This" isn't applicable in a menu. Instead, the object reference is put into a variable and that variable is referenced; for example, "loObject.Method()".

The ShortcutMenu method is used to actually fill the shortcut menu object with menu bars. It accepts two parameters: an object reference to the menu object to populate and the name of the variable containing the object reference to this object. Why pass the menu reference when it's stored in the oMenu property? Because we might want to hook several classes together, populating the menu object of the first one. For example, you might want to have the shortcut menu for a textbox include not only items for the textbox, but for the form it's on as well. To do that, set the lUseFormShortcutMenu property of the textbox to .T. The ShowMenu method of the textbox will call the ShortcutMenu method of the form, passing a reference to the textbox's menu object and the name of the variable the textbox defined that references the form (loForm). That way, the form can add its items to the textbox's menu and everything will work.

In most classes, ShortcutMenu is an abstract method since I couldn't think of any useful menu choices that'd be used in every instance and subclass. Those that have a text component (SFComboBox, SFEditBox, SFSpinner, and SFTextBox), however, have menu bars for Cut, Copy, Paste, Clear, and Select All. Here's the code from SFTextBox.ShortcutMenu:

```
lparameters toMenu, ;
  tcObject
with toMenu
  .AddMenuBar('Cu\<t',   "sys(1500, '_MED_CUT', ;
    '_MEDIT')", , , , 'not ' + tcObject + ;
    '.Enabled or ' + tcObject + '.ReadOnly')
  .AddMenuBar('\<Copy',  "sys(1500, '_MED_COPY', ;
    '_MEDIT')")
  .AddMenuBar('\<Paste', "sys(1500, '_MED_PASTE', ;
    '_MEDIT')", , , , 'not ' + tcObject + ;
    '.Enabled or ' + tcObject + '.ReadOnly')
  .AddMenuBar('Cle\<ar', "sys(1500, '_MED_CLEAR', ;
    '_MEDIT')", , , , 'not ' + tcObject + ;
    '.Enabled or ' + tcObject + '.ReadOnly')
  .AddMenuSeparator()
  .AddMenuBar('Se\<lect All', "sys(1500, '_MED_SLCTA', ;
    '_MEDIT')")
endwith
```

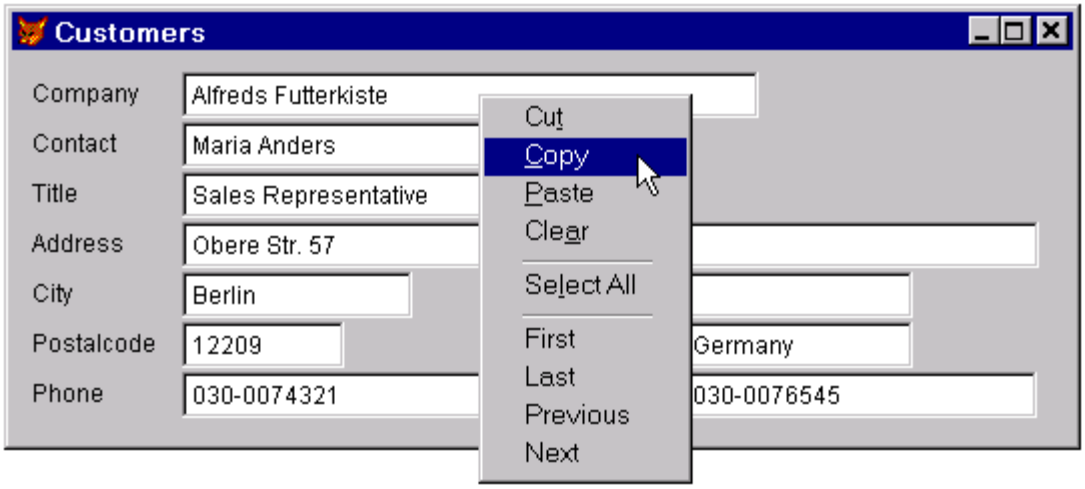
The sample CUSTOMERS form that accompanies this document has a shortcut menu with First, Last, Next, and Previous functions. To show how the form's menu can add to a control's menu, I set the lUseFormShortcutMenu property of txtCompany to .T. Right-click on that textbox and you'll see both the functions for textbox (Cut, Copy, Paste, Clear, and Select All) and those for

the form (First, Last, Next, and Previous) in the same menu. Here's the code for the ShortcutMenu method of this form:

```
lparameters toMenu, ;
  tcObject
with toMenu
  if .nBarCount > 0
    .AddMenuSeparator()
  endif .nBarCount > 0
  .AddMenuBar('First', tcObject + '.FirstRecord()')
  .AddMenuBar('Last', tcObject + '.LastRecord()')
  .AddMenuBar('Previous', tcObject + '.PreviousRecord()')
  .AddMenuBar('Next', tcObject + '.NextRecord()')
endwith
```

Notice that this code adds a separator bar to the menu if these functions are added to a control's menu, or not if the menu is strictly for the form.

Figure 1. The shortcut menu for the Company textbox includes functions for both the textbox and the form.



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 2, toolbar controls appear flat (for example, the command buttons have no outlines) until you move the mouse over them (see the third button from the left).

Figure 2. 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)
```

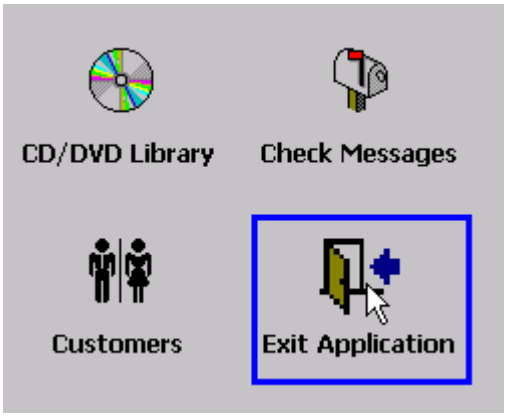
Since VERSION(5) was added in VFP 6, the TYPE() test and use of EVALUATE() in this statement ensure it will work even in VFP 5.

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.

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 (that's how the textbox in TESTHOTTRACKING.SCX does its thing). 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 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 3 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 3. The SwitchboardButton class shows an example of hot tracking using MouseEnter and MouseLeave.



Another example is the HyperlinkLabel class, also in MYCLASSES.VCX. The MouseEnter event for this class sets This.ForeColor to 16711680 (blue) and This.FontUnderline to .T., and MouseLeave restores them. This makes the label act like a hyperlink.

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, such as SFToolbarButton in SFBUTTON.VCX.

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):

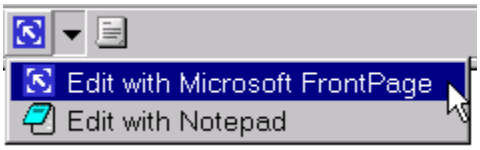
```
if clVFP7ORLATER
  This.Style = 1
endif clVFP7ORLATER
```

Figure 4 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 4. 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 5 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 5. 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, my 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 hot tracking of the buttons is 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 (discussed earlier), why reinvent the wheel? 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 5, 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 (once again, 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 document. ColorPicker is just a simple demonstration of this control; it allows the user to change the background color of the active form (or _SCREEN if there is no 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 or _SCREEN to the selected color:

```
local loObject
loObject = iif(type('_screen.ActiveForm.Name') = 'C', ;
```

```
  _screen.ActiveForm, _screen)
loObject.BackColor = getcolor(loObject.BackColor)
```

The ShortcutMenu method has the following code:

```
lparameters toMenu, ;
    tcObject
local lcObject
lcObject = iif(type('_screen.ActiveForm.Name') = 'C', ;
  '_screen.ActiveForm', '_screen')
toMenu.AddMenuBar('Red',   lcObject + ;
  '.BackColor = rgb(255, 0, 0)')
toMenu.AddMenuBar('Green', lcObject + ;
  '.BackColor = rgb(0, 255, 0)')
toMenu.AddMenuBar('Blue',  lcObject + ;
  '.BackColor = rgb(0, 0, 255)')
toMenu.AddMenuBar('Grey',  lcObject + ;
  '.BackColor = rgb(212, 208, 200)')
dodefault(toMenu, tcObject)
```

Application 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 shortcut menus, so let's talk about the main menu.

Menus haven't changed much in FoxPro since FoxPro 2.0. 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 (VFP calls inverted bars "MRU", although the meaning of that term here is different than you'd expect; in my opinion, they should've been called "adaptive" instead.) 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.

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 adaptive menu. However, later we'll look at a more useful, true MRU feature.

One change many people have been hoping for for a long time is an object-oriented menu system. There are several benefits for object-oriented menus. First, although it'd be extremely rare for a menu to be reusable from one application to another, what about parts of a menu? Individual bars or even entire pads could definitely be used in many applications. For example, the bars in the Edit pad (Undo, Redo, Cut, Copy, Paste, Clear, and Select All) are usually the same from application to application. Some bars in the Help pad (such as Help and About) and File pad (such as Print Setup and Exit) would likely call the same functions in every application. So, being able to reuse parts of a menu would be a time saver for many developers.

Another benefit of object-oriented menus is inheritance. You might have several bars that are similar (for example, they may have the same SKIP FOR clause or even call the same function, passing a parameter to indicate which one was chosen) but have different prompts. Wouldn't it be nice to create a new bar by subclassing an existing one and just changing the few things that make it unique?

VFP 7 still doesn't have object-oriented menus, so I created a set of classes that would provide them.

How Menus Work in VFP

Before we look at the classes that implement an object-oriented menu, let's review how menus are handled in VFP. A menu bar appears below the title bar of an application. Although you can define a menu bar using DEFINE MENU, most developers use the VFP system menu bar, _MSYSMENU, because it's easier and that's what code generated by the Menu Designer does.

A menu bar consists of pads. A typical application has File, Edit, and Help pads at a minimum, but Tools, View, and Window pads are also common. Pads are created with the DEFINE PAD command. A pad has a prompt, a hot key (usually an Alt-key combination), a SKIP FOR clause (which, when it evaluates to .T., disables the pad), and a message that appears in the status bar

when the pad is highlighted. Pads can have other properties, such as font and color, but these are rarely used. You can specify what should happen when a pad is selected, but the standard behavior is to display a popup.

As with pads, popups can have a number of properties specified with the DEFINE POPUP command, but the two that are typically used are MARGIN, indicating that extra space is available at the left margin of the popup so images or mark characters can be displayed, and RELATIVE, which means that items in the popup appear in the order in which they're defined.

Popups consist of bars, the items a user can select from the popup. A bar is created with the DEFINE BAR command, and has a number of properties you can specify, including the prompt, the hot key and the text representing the hot key that should appear to the right of the prompt, a SKIP FOR clause, and a message that appears in the status bar when the bar is highlighted. As with pads, other bar properties such as font and color are rarely used. VFP 7 adds several new bar properties: the name of a file containing the image to display to the left of the bar's prompt, the name of a system menu bar whose image should be used for the bar, whether the bar should appear inverted, and whether the bar should be an MRU bar. You specify what should happen when the user selects a bar with the ON SELECTION BAR command.

Now let's look at the classes that implement object-oriented menus. As I mentioned earlier, some of the options VFP provides for menus are rarely if ever used. I decided when I created these classes that I'd ignore options I never use or those that don't follow Window standards (many of these options are carry-overs from the DOS days). I also decided for simplicity to ignore cascading menus for now (a cascading menu has bars that, when selected, display a subpopup of choices). All the classes discussed in this article are located in SFMENU.VCX.

I'd like to give credit to my sources of inspiration (and, as usual, a bit of code <g>) for these classes: the Visual FoxPro 3 Codebook by Yair Alan Griver (Sybex, ISBN 0-7821-1648-5) and Visual FoxExpress from F1 Technologies (www.f1tech.com), written by Mike and Toni Feltman.

SFMenu

This class represents the menu bar. It's a subclass of SFCollection, a class defined in SFCOLLECTION.VCX that provides collection management (see my column in the July 1998 issue of FoxTalk for details on this SFCollection). SFMenu has only one custom property, cInstanceName, which contains the name of the variable this class is instantiated into (we'll see why we need this later). Although it's public, it has an assign method that makes it read-only to everything except methods of this class. When you instantiate SFMenu, pass the name of the variable as a parameter; the Init method sets cInstanceName to the parameter value. Here's an example:

```
loMenu = newobject('SFMenu', 'SFMenu.vcx', '', 'loMenu')
```

The AddPad method is used to add a new pad to the menu. Pass the class for the pad, the library the class is contained in, and the name to assign to the pad; I suggest using the prompt with "Pad" as the suffix for the name (for example, "FilePad"). The class is instantiated as a member

of SFMenu and is also added to the collection so pads can easily be enumerated. Here's an example that calls this method:

```
loMenu.AddPad('SFEditPad', 'SFMenu.vcx', 'EditPad')
```

You can now reference the new pad as loMenu.EditPad.

The Show method displays the menu. If no pads have been added to the menu, Show calls the DefineMenu method. That method is an abstract method in this class, but in a subclass, you could put a series of AddPad calls that add the desired pads to the menu; this makes the menu subclass self-contained (no outside code has to add pads to the menu). The code then calls the Show method of each pad in the menu and turns on menu handling. Here's the code for Show:

```
local lnI, ;
  loPad
with This
  set sysmenu to
  if .Count = 0
    .DefineMenu()
  endif .Count = 0
  for lnI = 1 to .Count
    loPad = .Item(lnI)
    loPad.Show()
  next lnI
  set sysmenu automatic
endwith
```

The Refresh method refreshes the menu by activating it again; this is useful after you've displayed the menu, and then changed some of the bars or pads in the menu. The ReleaseMembers method, called when the object is destroyed (this is actually defined in SFCustom, the parent class of SFCollection), restores the VFP system menu bar.

SFPad

SFPad is the parent class for all pads in a menu. It too is a subclass of SFCollection. Set the cCaption property to the prompt for the pad (such as "File"), cKey to the hot key (such as "Alt+F"), and cStatusBarText to the message to display in the status bar (such as "Performs file functions"). You can also set cSkipFor to an expression that, when it evaluates to .T., disables the pad, lVisible to .F. if the pad shouldn't currently be visible (you can later set it to .T. to display the pad), and lMRU to .T. if the pad should have an MRU bar in VFP 7 (this property is ignored in VFP 6). Wait a minute: isn't MRU a property of a bar instead of a pad? Yes, but if you look at Office 2000, you'll see that no pad's popup has more than one MRU bar in it, and it's always the last bar. That means that really, MRU status should be a property of the pad (or popup) rather than an individual bar. When lMRU is .T., SFPad will automatically create an MRU bar using the class specified in the cMRUBarClass and cMRUBarLibrary (by default, "SFMRUBar" and "SFMenu.vcx", respectively).

Notice that in VFP menus, there's no direct relationship between pads and bars. Instead, bars belong to a popup and a popup is associated with a pad. I decided to simplify this in my design; I really couldn't see the need to expose popups at all (if you think about it, you'll realize that's

how the VFP Menu Designer works too). So, bars will belong to pads in this design. The Init method automatically creates a popup and stores its name in the protected cPopupName property.

To add a bar to a pad, call the AddBar method, passing the class for the bar, the library the class is contained in, the name to assign to the bar (I suggest concatenating the pad prompt, the bar prompt, and "Bar" for the name, such as "FileExitBar"), and optionally the bar number (if you don't pass this, AddBar will automatically assign the next available bar number to it). The class is instantiated as a member of SFPad and is also added to the collection so bars can easily be enumerated. The Init method of the bar class is passed the name of the popup, the bar number, and .T. if the bar number was specified or .F. if AddBar assigned it. Here's an example that calls AddBar:

```
loMenu.FilePad.AddBar('SFBar', 'SFMenu.vcx', 'FileOpenBar')
```

You can now reference the new bar as loMenu.FilePad.FileOpenBar. To add a separator bar, call the AddSeparatorBar method (it doesn't accept any parameters).

The Show method displays the pad, its popup, and the bars in the popup. If the pad hasn't been defined yet (the protected lDefined property is .F.), the Define method is called to define the pad and create an MRU bar if the lMRU property is .T. Define also calls AddBars. That method is an abstract method in this class, but in a subclass, you could put a series of AddBar calls that add the desired bars to the pad; this makes the pad subclass self-contained. The Show method then calls the either Show or Hide method of each bar in the pad (Hide if the bar is inverted, Show if not). Since this method is called from the Show method of SFMenu, you probably won't have to call this method directly unless you call the Hide method (discussed next). Here's the code for Show:

```
local lnI, ;
  loBar
with This
  if not .lDefined
    .Define()
  endif not .lDefined
  for lnI = 1 to .Count
    loBar = .Item(lnI)
    if loBar.lInvert
      loBar.Hide()
    else
      loBar.Show()
    endif loBar.lInvert
  next lnI
endwith
```

The Hide method releases the pad and popup that underlay this class, and sets the lVisible and lDefined properties to .F. You can call this method to make a pad disappear, and then later call Show to make it reappear. Alternatively, you can set lVisible to .F. and then later .T.; this property has an assign method that calls either Hide or Show, depending on the value you're setting it to.

The MRUSelected method is public only because it's called when the MRU bar belonging to the pad is selected. Selecting an MRU bar causes all inverted bars to be displayed, so this method starts by hiding the MRU bar and showing all inverted bars. It then reactivates the popup, and

after a selection has been made, redisplays the MRU bar and hides all inverted bars. Here's the code for this method:

```
local lnI, ;
  loBar, ;
  lcPopupName
with This

* Hide the MRU bar, then show all inverted bars.

  .MRUBar.Hide()
  for lnI = 1 to .Count
    loBar = .Item(lnI)
    if loBar.lInvert
      loBar.Show()
    endif loBar.lInvert
  next lnI

* Display the popup again.

  lcPopupName = .cPopupName
  activate popup &lcPopupName

* Now that the popup is closed, show the MRU bar and
* hide all inverted bars.

  .MRUBar.Show()
  for lnI = 1 to .Count
    loBar = .Item(lnI)
    if loBar.lInvert
      loBar.Hide()
    endif loBar.lInvert
  next lnI
endwith
```

Note that inverted bars are only displayed when the user selects the MRU bar, and are hidden again afterward. Other applications that use this technology, such as Word and the Start menu in Windows 2000, typically use a more complex behavior. For example, in Word, if you select an inverted bar, it stays visible until you close the application. If you select the bar again and again over time, it becomes more permanently visible. Bars that originally weren't inverted may become so if you don't use them over time. Obviously, this requires a lot more bar management than the classes I created will support!

The ReleaseMembers method, called when the object is destroyed, calls Hide to destroy the pad and popup.

SFEditPad is a subclass of SFPad. Its AddBars method adds bars for Undo, Redo, Cut, Copy, Paste, Clear, and Select All functions. I figured since every application has one of these pads, why bother reinventing the wheel each time?

SFBar

SFBar is the parent class for all bars in a menu. It's based on SFCustom (defined in SFCTRLS.VCX), a subclass of the Custom base class. Set the cCaption property to the prompt for the bar (such as "Open…"), cKey and cKeyText to the hot key and the text for it (such as "Alt+F" for both), and cStatusBarText to the message to display in the status bar (such as "Open a document"). If you want to use a VFP system bar, set cSystemBar to the name of the bar (for

example, the SFHelpTopicsBar subclass has cSystemBar set to "_MST_HPSCH" so it automatically has the functionality of that system bar). To conditionally disable the bar, you can either set cSkipFor to an expression or put code in the Allow method that returns .T. if the user is allowed to select the bar. You can also set lEnabled to .F. to unconditionally disable the bar. Set lVisible to .F. if the bar shouldn't currently be visible (you can later set it to .T. to display the bar). To display a mark (such as a checkmark) beside the bar, set lMarked to .T.

There are four ways you can define what happens when the user selects the bar: set cActiveFormMethod to the name of the method of _screen.ActiveForm to call (such as "Find"; you can specify parameters in parentheses if necessary), set cAppObjectMethod to the name of the method of oApp to call (if you use an application object and it's instantiated into a global variable called oApp), set cOnClickCommand to the VFP command to execute, or put the code to execute in the Click method of a subclass of SFBar.

In VFP 7, bars can have graphics; to use this feature, either set cPictureFile to the name of the graphic file to use or cPictureResource to the name of the VFP system bar whose graphic you want to use (for example, "_MFI_NEW" to use the image of the File New bar). VFP 7 also supports inverted bars, so set lInvert to .T. to have the bar only visible when the MRU bar is selected and to have it appear inverted. These properties are ignored in VFP 6.

You likely won't need to call any SFBar methods directly. Show displays the bar by first calling the protected FindBarPosition method if necessary (if the bar number was specified rather than automatically assigned or if the bar is inverted, it may need to be placed between other bars, so FindBarPosition figures out where it should go), and then calling the protected Define method to set up and execute the DEFINE BAR and ON SELECTION BAR commands. The Hide method releases the bar so it disappears from the menu.

By the way, the reason we need to know the name of the variable SFMenu is instantiated into is the ON SELECTION BAR command. We can't use code like "This.Click", because "This" isn't available in the context of a menu selection. Instead, we need to use something like "loMenu.FilePad.FileOpenBar.Click". We can get "FilePad" from This.Parent.Name and "FileOpenBar" from This.Name, but we can't get "loMenu" from any native property. Thus, we need to store the name of the variable the menu was instantiated into in SFMenu.cInstanceName. Then, the entire path to the command to execute can be determined with:

```
This.Parent.Parent.cInstanceName + '.' + ;
  This.Parent.Name + '.' + This.Name + '.Click()'
```

I created a few commonly used subclasses of SFBar. SFHelpTopicsBar (discussed earlier) provides a Help Topics bar. SFSeparatorBar is used when you call SFPad.AddSeparatorBar; it simply has cCaption set to "\-". SFMRUBar is the class used for an MRU bar when you set SFPad.lMRU to .T.

Example: MAIN.PRG

MAIN.PRG is the startup program for a simple application that demonstrates the techniques discussed in this document and the menu enhancements in VFP 7. Part of its code defines a menu

programmatically using the OOP menu classes. SFMenu is instantiated into the oMenu variable, and it's given two pads: File and Edit. The File pad's lMRU property is set to .T. and an inverted bar, Print Setup, and an associated separator bar are added between the Invoices and Exit bars. When you run MAIN.PRG, you won't see the Print Setup bar initially. Click on the MRU bar at the bottom of the menu, and Print Setup appears. After you choose an item from the menu (other than Exit, of course), Print Setup disappears again. The Print Setup and Exit bars have graphics associated with them, via the values of the cPictureResource and cPictureFile properties. The Edit pad has the usual bars (it's an instance of the SFEditPad class).

Here's the relevant code from MAIN.PRG:

```
oMenu = newobject('SFMenu', 'SFMenu.vcx', '', 'oMenu')

* Create the File pad.

oMenu.AddPad('SFPad', 'SFMenu.vcx', 'FilePad')
with oMenu.FilePad
  .cCaption       = '\<File'
  .cKey           = 'ALT+F'
  .cStatusBarText = 'Performs file functions'
  .lMRU           = .T.

  .AddBar('SFBar', 'SFMenu.vcx', 'FileCustomers')
  with .FileCustomers
    .cCaption         = '\<Customers'
    .cStatusBarText   = 'Display the customers form'
    .cOnClickCommand  = 'do form Customers'
  endwith

  .AddBar('SFBar', 'SFMenu.vcx', 'FileInvoices')
  with .FileInvoices
    .cCaption         = '\<Invoices'
    .cStatusBarText   = 'Open a file'
    .cOnClickCommand  = [messagebox('You chose ] + ;
      [File, Invoices')]
  endwith

  .AddBar('SFBar', 'SFMenu.vcx', 'FilePrintSetup')
  with .FilePrintSetup
    .cCaption         = '\<Print Setup...'
    .cStatusBarText   = 'Change the printer settings'
    .cOnClickCommand  = 'sys(1037)'
    .lInvert          = .T.
    .cPictureResource = '_mfi_sysprint'
  endwith

  loBar = .AddSeparatorBar()
  loBar.lInvert = .T.
  loBar.cBarPosition = 'before ' + ;
    transform(.FilePrintSetup.nBarNumber)

  .AddSeparatorBar()

  .AddBar('SFBar', 'SFMenu.vcx', 'FileExit')
  with .FileExit
    .cCaption        = 'E\<xit'
    .cStatusBarText  = 'Exit this application'
    .cOnClickCommand = 'ExitApp()'
    .cPictureFile    = 'close.bmp'
  endwith
endwith

* Create the edit pad.
```

```
oMenu.AddPad('SFEditPad', 'SFMenu.vcx', 'EditPad')

* Display the menu.

oMenu.Show()
```

MRU Menus

A more useful version of an MRU feature is 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 document 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:

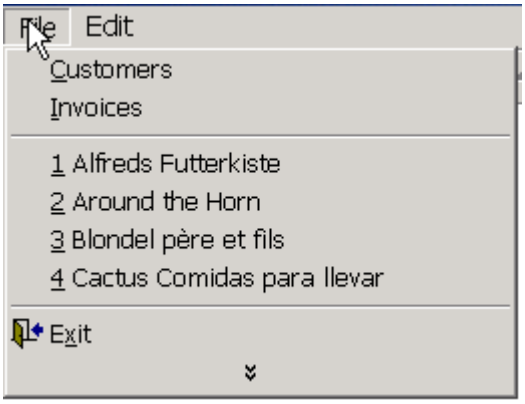
```
lparameters tcFunction, ;
  tcCaption
local lnRows, ;
  loSeparator, ;
  lcBar, ;
  loBar
```

```
* 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 6 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 6. 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.

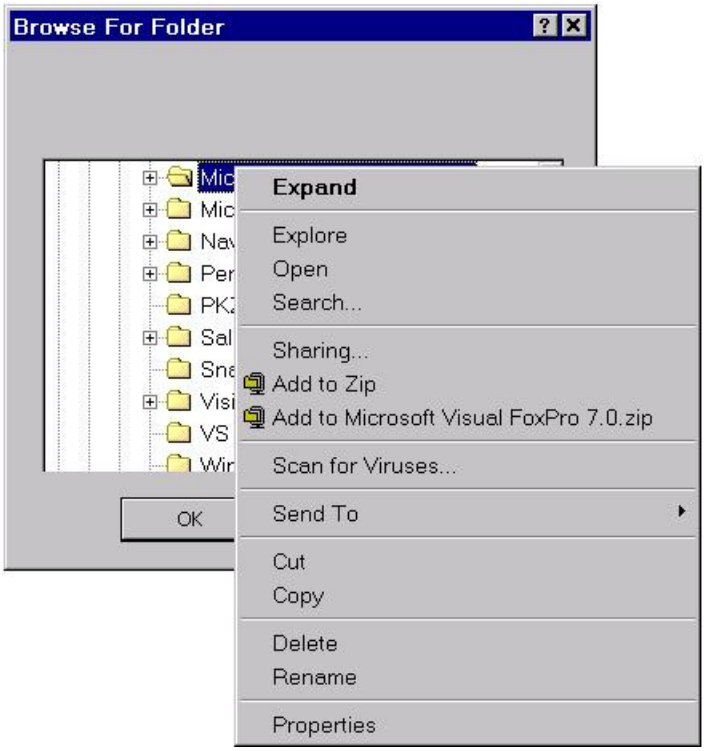
File and Folder Dialogs

GETDIR() displays a dialog in which the user can select a directory. This function isn't used often in a typical data entry application, but may be used, for example, in a configuration or preferences dialog to specify where certain files should go.

In VFP 7, GETDIR() has three new parameters: cCaption, the caption of the dialog (the default is "Browse for Folder"), nFlags to indicate the behavior of the dialog, and lRootOnly that allows you to treat the starting directory as the root so the user can't navigate above it. If any of the new parameters are passed, this function uses the SHBrowseForFolder Windows API function, so the dialog has the same user interface as other Windows applications.

The VFP Help topic for GETDIR() includes a list of some of the values for the nFlags parameter; the SHBrowseForFolder topic in the MSDN Help file has a complete list. Like other functions that support a similar parameter (such as the MESSAGEBOX() function), you can add together different values to provide the behavior you want. Figure 6 shows an example that illustrates Windows Explorer-like features by using flag value 64.

Figure 6. Adding 64 to the flags parameter of GETDIR() displays a dialog with features similar to Windows Explorer, such as context menus.



I normally call GETDIR() from a button in a form. This button sits beside a textbox where the user could type a directory path; the button makes it easier because they can simply navigate to the desired directory in a GETDIR() dialog and choose OK. To easily implement such an interface, I created SFGetDir (in SFBUTTON.VCX). SFGetDir has several custom properties, including cResult (which contains the name of the place to put the directory the user selected, such as the Value property of a textbox), cDefaultDir (the default directory; if it's left empty, the value of cResult is used as the default), cText (the text to display in the dialog), cCaption (the caption for the dialog), lEditBox (.T. to display an editbox), lIncludeFiles (.T. to include files in the list of folders), lUseNewUI (.T. to use Windows Explorer-like features), and cAfterDone (an expression to evaluate after the dialog is closed; it can be used for validation, directory name processing, etc.). The Click method of SFGetDir has the following code:

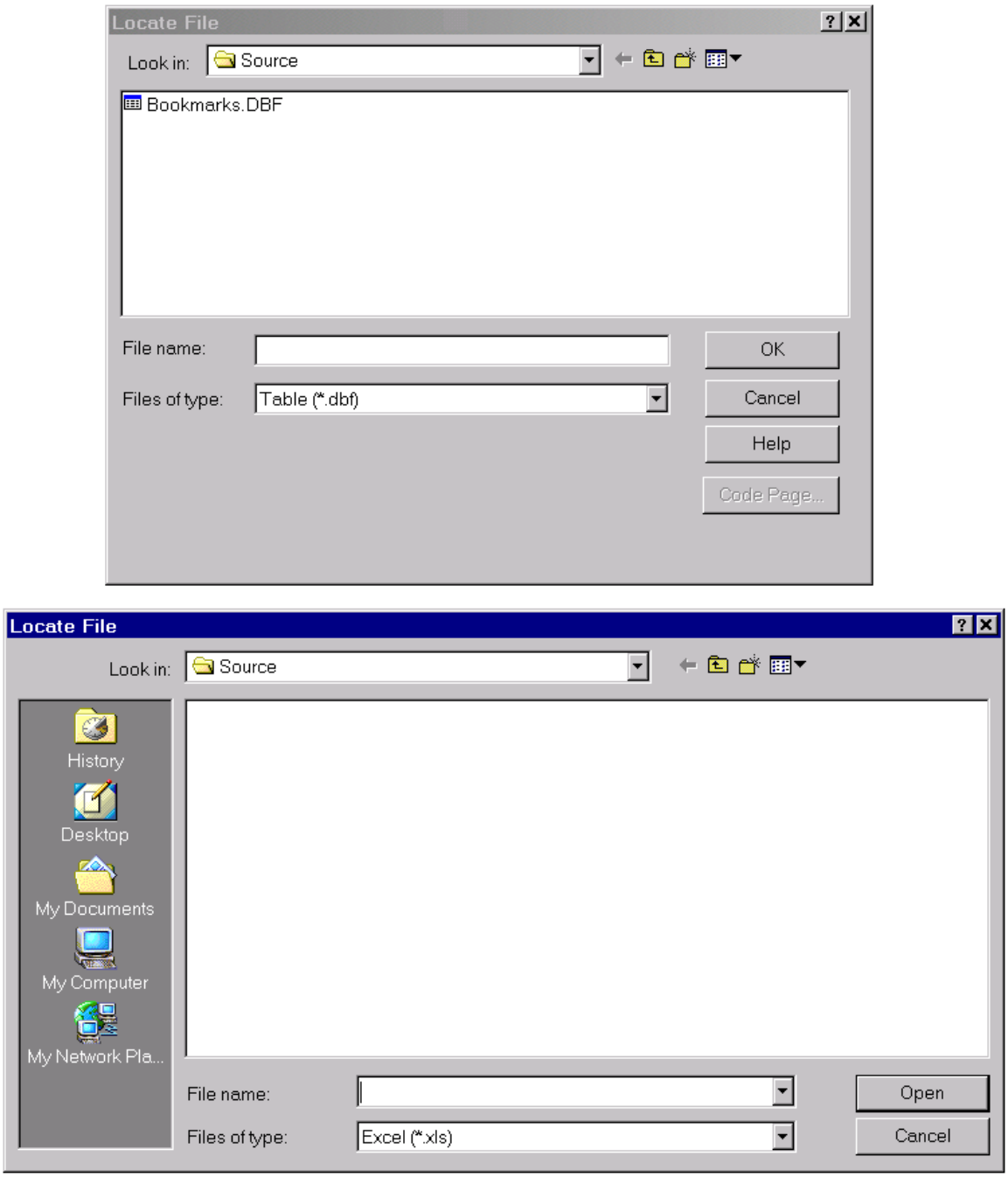
```
local lcResult, ;
  lcOptions, ;
  lnFlags, ;
  lcDir
with This
  assert not empty(.cResult) ;
    message 'SFGetDir: the result container was not ' + ;
    'defined.'

* Build a string of parameters from the properties of
* this object.

  lcResult = .cResult
  do case
    case empty(.cDefaultDir)
```

```
      lcOptions = evaluate(lcResult)
    case left(.cDefaultDir, 1) = '='
      lcOptions = '[' + ;
        evaluate(alltrim(substr(.cDefaultDir, 2))) + ']'
    otherwise
      lcOptions = alltrim(.cDefaultDir)
  endcase
  lcOptions = iif(empty(.cText), lcOptions, lcOptions + ;
    iif(empty(lcOptions), '[]', '') + ',[' + .cText + ;
    ']')
  do case
    case empty(.cCaption)
    case empty(lcOptions)
      lcOptions = '[],[],[' + .cCaption + ']'
    case not ',' $ lcOptions
      lcOptions = lcOptions + ',[],[' + .cCaption + ']'
    otherwise
      lcOptions = lcOptions + ',[' + .cCaption + ']'
  endcase

* In VFP 7 or later, support new options.

  if clVFP7ORLATER
    #define BIF_RETURNONLYFSDIRS       1
    #define BIF_EDITBOX               16
    #define BIF_VALIDATE              32
    #define BIF_USENEWUI              64
    #define BIF_BROWSEINCLUDEFILES 16384
    lnFlags = 0
    if .lEditBox
      lnFlags = lnFlags + BIF_EDITBOX + BIF_VALIDATE
    endif .lEditBox
    if .lIncludeFiles
      lnFlags = lnFlags + BIF_BROWSEINCLUDEFILES
    endif .lIncludeFiles
    if .lUseNewUI
      lnFlags = lnFlags + BIF_USENEWUI
    endif .lUseNewUI
    if lnFlags > 0
      lnFlags = lnFlags + BIF_RETURNONLYFSDIRS
    endif lnFlags > 0
    do case
      case lnFlags = 0
      case empty(lcOptions)
        lcOptions = '[],[],[],' + transform(lnFlags)
      case not ',' $ lcOptions
        lcOptions = lcOptions + ',[],[],' + ;
          transform(lnFlags)
      case occurs(',', lcOptions) = 1
        lcOptions = lcOptions + ',[],' + ;
          transform(lnFlags)
      otherwise
        lcOptions = lcOptions + ',' + ;
          transform(lnFlags)
      endcase
  endif clVFP7ORLATER

* Use the GETDIR() function, and if a directory was
* selected, store the result in the specified location.

  lcDir = getdir(&lcOptions)
  if not empty(lcDir)
    store lcDir to (lcResult)
  endif not empty(lcDir)

* If a method or function was specified to execute after
* GETDIR(), do it.

  if not empty(.cAfterDone)
    evaluate(.cAfterDone)
  endif not empty(.cAfterDone)
```

```
endwith
```

There's nothing new in VFP 7 for the GETFILE() or PUTFILE() commands, but a new _ComDlg class is included in the FFC (in _SYSTEM.VCX). Many people, including me, like to use the Common Dialogs ActiveX control rather than GETFILE() and PUTFILE() because it has a lot more control over the appearance and behavior of the dialogs (for example, you can specify a default directory and suppress the Help and Codepage buttons if you wish) and it appears as a more modern-looking dialog (see Figure 7). Now, you don't have to include COMDLG32.OCX (the file containing the Common Dialogs control) in the list of files installed on the user's system or worry about version issues and DLL hell. Instead, use _ComDlg. Rather than working with the Common Dialogs control, it directly uses the Windows API functions that control is simply a front for. See the "Common Dialog Box Foundation Class" topic in the VFP 7 help file for details on using this class. Note that although this class comes with VFP 7, it works just fine in VFP 6 as well.

Figure 7. The Common Dialogs control (bottom) looks more modern and provides more programmatic control than the GETFILE() dialog (top).



As with GETDIR(), I created SFGetFile and SFPutFile classes (also in SFBUTTON.VCX) that I can simply drop on a form and set some properties. Since these classes are similar, we'll just look at SFGetFile. Like SFGetDir, SFGetFile has cResult, cAfterDone, and cCaption properties to define where the name of the selected file should be placed, what to do after a file has been selected, and the caption of the dialog. Set the lUseCommonDialog property to .T. to use the _ComDlg class or .F. to use GETFILE(). Put a semicolon-delimited list of file extensions (along

with their descriptions) into cExtensions. If lUseCommonDialog is .T., put the default directory into cDefault. If not, put the desired text for the dialog into cText, the caption for the Open button into cOpenButton, and the proper value into nButtonType (see the VFP help topic for GETFILE() for a list of values). The Click method of SFGetFile has the following code:

```
local lcResult, ;
  loDialog, ;
  lcExt, ;
  laTypes[1], ;
  lnTypes, ;
  lnI, ;
  laExt[1], ;
  lcFile, ;
  lcCurDir, ;
  lcOptions
with This
  assert not empty(.cResult) ;
    message 'SFGetFile: the result container was ' + ;
    'not defined.'

* If we're using the CommonDialog control, instantiate
* the _ComDlg class, set its properties, and call the
* ShowDialog method.

  lcResult = .cResult
  if .lUseCommonDialog
    loDialog = MakeObject('_ComDlg', ;
      home() + 'FFC\_System.vcx')
    if not empty(.cExtensions)
      lcExt    = strtran(evaluate(.cExtensions), ';', ;
        chr(13))
      lnTypes  = alines(laTypes, lcExt)
      loDialog.ClearFilters()
      for lnI = 1 to lnTypes
        lcExt = laTypes[lnI]
        if not ',' $ lcExt
          lcExt = lcExt + ',*.' + lcExt
        endif not ',' $ lcExt
        lcExt = strtran(lcExt, ',', chr(13))
        alines(laExt, lcExt)
        loDialog.AddFilter(laExt[1], laExt[2])
      next lnI
    endif not empty(.cExtensions)
    loDialog.cTitleBarText = .cCaption
    if not empty(.cDefault)
      loDialog.cFileName = alltrim(evaluate(.cDefault))
    endif not empty(.cDefault)
    lcCurDir = sys(5) + curdir()
    loDialog.lSaveDialog = .F.
    loDialog.ShowDialog()
    cd (lcCurDir)
    lcFile = addbs(loDialog.cFilePath) + ;
      loDialog.cFileTitle
  else

* We're using GETFILE(), so build a string of parameters
* from the properties of this object and call GETFILE().

    lcOptions = iif(empty(.cExtensions), "''", ;
        .cExtensions) + ',' + ;
      iif(empty(.cText), "''", "'" + .cText + "'") + ;
        ',' + ;
      iif(empty(.cOpenButton), "''", "'" + ;
        .cOpenButton + "'") + ',' + ;
      iif(empty(.nButtonType), '0', ;
        ltrim(str(.nButtonType))) + ;
      iif(empty(.cCaption), '', ",'" + .cCaption + "'")
    lcFile = getfile(&lcOptions)
```

```
  endif .lUseCommonDialog

* If the user chose a file, store the result in the
* specified location.

  if not empty(lcFile)
    store lcFile to (lcResult)

* If a method or function was specified to execute after
* file selection, do it.

    if not empty(.cAfterDone)
      evaluate(.cAfterDone)
    endif not empty(.cAfterDone)
  endif not empty(lcFile)
endwith
```

To see an example of SFGetDir and SFGetFile, run TESTDIALOGS.SCX.

Browser Forms

Much to the chagrin of the U.S. Justice Department <g>, Internet Explorer (IE) is a ubiquitous application. However, the majority of its functionality isn't in IE.EXE, but rather in the Microsoft Web Browser ActiveX control. By dropping this control on a VFP form, you can display HTML documents in VFP without worrying about all the baggage (or lack of programmatic control) that comes with IE. You have to use a trick to get this to work properly in VFP: put NODEFAULT in the Refresh method of the control. You can also use the _WebBrowser4 or _WebForm classes that come with VFP (in GALLERY\WEBVIEW.VCX).

An obvious use of this control is to display reports or other information from HTML your application generates. However, another use is as a navigation device, such as a switchboard form. An advantage for using HTML for such a device is that is can easily be customized. For example, you might have an Options dialog that allows your user to customize what functions should appear in the switchboard. All you have to do is generate a new HTML file based on the user's choices and you're done. You could even have a more advanced user or system administrator update the HTML file directly (if you're that brave <g>).

SWITCHBOARD.SCX is an example of such a form. It has a Microsoft Web Browser control named oWebControl on it. The Init method sizes the control so it's slightly larger than the form so borders don't show and automatically displays MENU.HTML:

```
with This.oWebControl
  .Top    = -4
  .Left   = -4
  .Height = This.Height + 6
  .Width  = This.Width  + 24
  .Navigate2('file://' + fullpath('Menu.html'))
endwith
```

MENU.HTML was created in FrontPage to display a switchboard form. It contains several hyperlinks that display VFP forms. Wait a minute: how can the Web Browser control display a VFP form? It can't. Instead, when the user clicks on a link, the desired form is run. Here's an example of such a link in MENU.HTML:

```
<a href="vfps://oMenu.FilePad.FileCustomers.Click()">
Customers form</a>
```

That looks like a normal hyperlink except it uses "VFPS://" and specifies what looks like VFP code. How can the Web Browser deal with that? Actually, it can't, but it doesn't have to: we'll grab that before the Web Browser has a chance to do anything with it. The BeforeNavigate2 event of the control fires when a link is selected but before the control actually navigates to it. Here's the code I placed in that event:

```
LPARAMETERS pdisp, url, flags, targetframename, ;
  postdata, headers, cancel
if upper(URL) = 'VFPS://'
  lcAction = upper(strextract(URL, 'VFPS://', '/', 1, 3))
  Cancel = .T.
  &lcAction
endif upper(URL) = 'VFPS://'
```

This code checks to see if "VFPS://" exists in the URL to navigate to, and if so, extracts the VFP command from the rest of the URL (notice it uses the new VFP 7 STREXTRACT() function, which saves several lines of ugly string processing code), sets the Cancel parameter to .T. (since it's passed by reference rather than by value, setting it to .T. cancels the navigation request), and then macro expands the command. In the case of the Customers Form link, the command is to execute the same code that choosing the Customers function in the File pad does (doing it this way rather than something like DO FORM CUSTOMERS means I don't have to duplicate code in several places, and then change all of those places when I need to change the code).

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 SWITCHBOARD2 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. The form also shows the use of shortcut menus.

To use the SWITCHBOARD form, which uses the Web Browser instead of SwitchboardButton mechanism, change MAIN.PRG to DO FORM SWITCHBOARD instead of SWITCHBOARD2.

Summary

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.

Biography

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).