A Deep Dive into the VFPX ThemedControls

Doug Hennig
Stonefield Software Inc.
2323 Broad Street
Regina, SK Canada S4P 1Y9
Email: dhennig@stonefield.com
Web sites: www.stonefield.com
www.stonefieldquery.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

One of the coolest projects in VFPX is ThemedControls by Emerson Santon Reed (the 2009 Southwest Fox Ceil Silver Ambassador). The ThemedControls project consists of eight controls—ThemedButton, ThemedContainer, ThemedExplorerBar, ThemedForm, ThemedOutlookNavBar, ThemedTitlePageFrame, ThemedToolbox, and ThemedZoomNavBar—that allow you to provide the modern interface users expect in today's applications. This document looks at these controls in detail and shows how to use them in your own applications.

Introduction

One of the reasons I hear that VFP developers are told to move to .Net is that .Net has controls that provide a newer, fresher look to applications. VFP apps look old, they say. Old fonts like Arial, old colors like the background grey, old icons for buttons, and old-style menus and toolbars. However, there's no reason for that. You can easily use newer, cleaner fonts like Segoe UI (the standard system font in Windows Vista and Windows 7), you can use modern, colorful 32-bit icons (there are hundreds or even thousands of web sites that provide free or paid icons), and you can use some of the projects on VFPX (http://vfpx.codeplex.com) to provide modern-style menus, toolbars, and other graphical user interface elements in your applications.

For example, the form shown in **Figure 1**certainly isn't a boring old VFP form, and yet it is a VFP form. This form uses a Microsoft Outlook-like control to display different categories of items and supports themes to display dynamic, interesting colors.

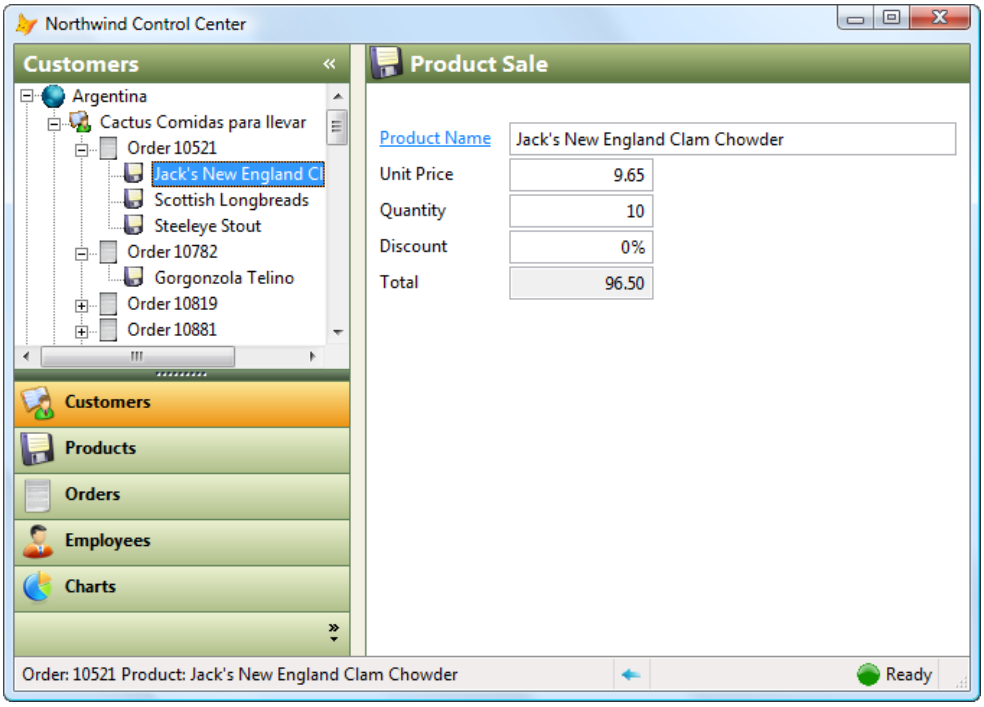


Figure 1. There's no reason you have to provide a boring user interface in your VFP applications.

VFPX is a community site for extensions for VFP developers. Amongst its many projects are several that provide more modern interfaces to VFP applications:

- GDIPlusX, which is also the basis for several other projects, provides access to GDI+, the same graphics engine .Net applications use to draw their user interfaces. That means anything you see in a Windows application can be recreated in a VFP application.
- Desktop Alerts gives you the ability to display "toaster" pages that appear when some event happens, similar to what Outlook displays when an email arrives.

- PopMenu provides owner-drawn menus that have a Microsoft Office-like appearance.
- TabMenu allows you to add a "ribbon" navigation system like Microsoft Office 2007 and later has.
- FoxCharts creates gorgeous 3-D charts you can host in your forms or reports without using any ActiveX controls.
- ThemedControls provides controls (including the Outlook control shown in **Figure 1**) that display in themed colors.

In this document, we'll focus on ThemedControls. This won't be a quick overview or demo; rather, it's a deep dive into these controls, covering how they work (in some cases) and showing specific details about how to add them to your applications.

Getting started: downloading ThemedControls

Like other VFPX projects, ThemedControls is available for download from http://vfpx.codeplex.com. From the home page, click the link for the ThemedControls project in the project list, then on the ThemedControls page, click the Latest Release of ThemedControls to navigate to the download page. (The direct link to that page is http://vfpx.codeplex.com/releases/view/7003.) Finally, click the download link to download the ZIP file to your system. The name of the ZIP for the current release as of this writing is ThemedControls_3-5-8.ZIP.

Unzip the ZIP file in a folder of your choice. This creates the following subdirectories:

- ThemedControls: contains sample files, System.APP (discussed later), and Themes.XML (also discussed later).
- ThemedControls\VCX: the source files for ThemedControls. The main set is ThemedControls.VCX/VCT, but as we'll see, there are numerous supporting files as well.
- ThemedControls\Images: image files used by the samples.

Adding ThemedControls to your project

Adding ThemedControls.VCX to your project actually adds several other files to the project when you build it:

- Buttons.VCX: contains the parent class of ThemedButton.
- Ctl32.VCX: classes created by Carlos Alloatti that are used by ThemedControls.
- ExplorerBar.VCX: contains the parent classes of the classes used by ThemedExplorerBar.

- OutlookNavBar.VCX: contains the parent classes of the classes used by ThemedOutlookNavBar.
- Toolbox.VCX: contains the parent classes of the classes used by ThemedToolbox.
- ZoomNavBar.VCX: contains the parent classes of the classes used by ThemedZoomNavBar.
- VFPX.VCX: supporting classes needed by both ThemedControls and Ctl32. Note: this VCX isn't added to your project automatically. You need to add this yourself or your users will have a problem when they try to run the application. You may not notice it because VFPX.VCX is on your system so the application can find it, but it won't be on the user's system.
- Ctl32.PRG, Ctl32_API.PRG, Ctl32_Classes.PRG, Ctl32_Functions.PRG, Ctl32_Structures.PRG, and Ctl32_VFP2C32.PRG: support programs needed by Ctl32.
- ThemedControls_API.PRG and ThemedControls_Structures.PRG: support programs needed by ThemedControls.

If you aren't using some of the ThemedControl classes, you can mark the VCX containing the parent classes as excluded. For example, if you don't use ThemedToolbox or ThemedZoomNavBar, you can mark Toolbox.VCX and ZoomNavBar.VCX as excluded.

In addition, ThemedControls uses GDIPlusX for GDI+ work, so you need to include System.APP in the files you distribute with your application.

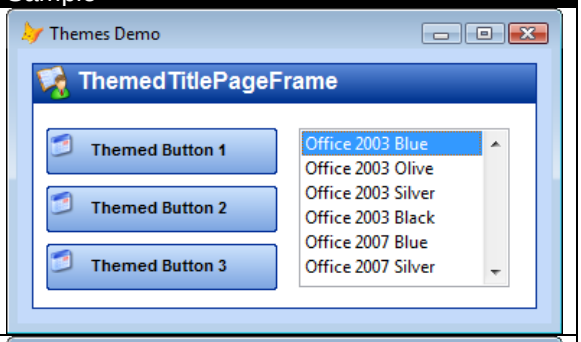
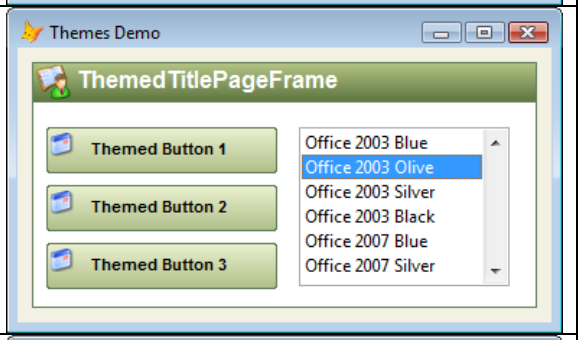
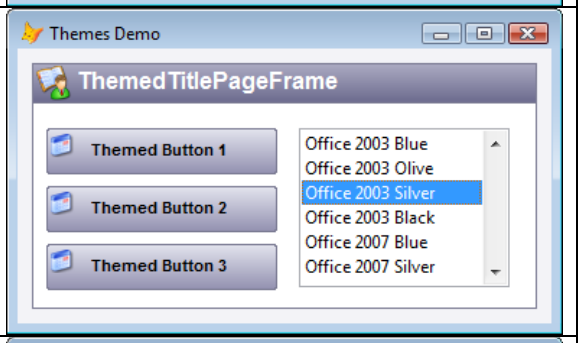
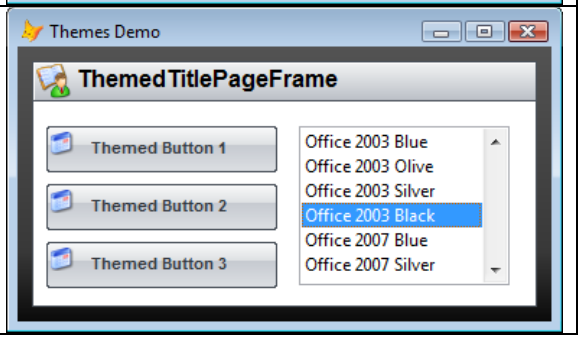
Without excluding any classes, an EXE containing nothing but ThemedControls and supporting files is 1.8 MB. System.APP is another 820 KB, so if you weren't already using GDIPlusX in your application, adding ThemedControls adds 2.6 MB to your application. Disk space is cheap, so that's a small price to pay for the cool features both VFPX projects can add to your applications.

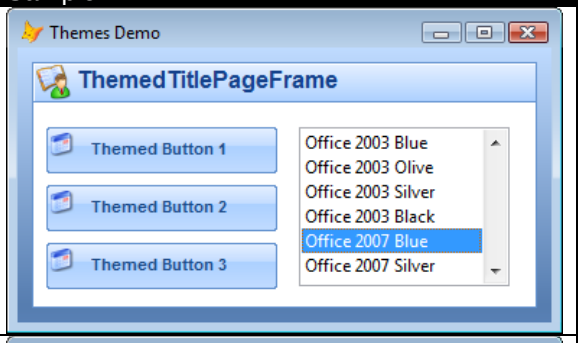
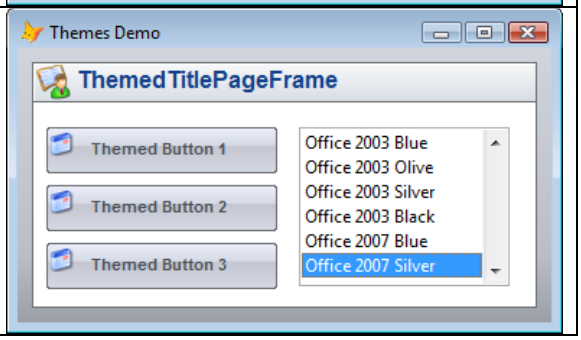
ThemesManager

We'll start our tour of the ThemedControls project with the ThemesManager class, as it's at the heart of the "themed" part of ThemedControls. ThemesManager, contained in ThemedControls.VCX, is a subclass of Custom that's responsible for managing a set of themes, controlling which theme is selected, and communicating to themed objects when the selected theme changes.

Themes are sets of colors and images, or "skins," for controls. ThemedControls comes with the themes shown in **Table 1**. (The sample images were taken by running ThemesDemo.SCX, included with the sample files accompanying this document.) Themes are defined in one or more XML files; Themes.XML, included with ThemedControls, provides the built-in themes. It's theoretically possible to create your own themes, but practically it's a little hard to do, as you can likely guess by looking at the partial listing of Themes.XML shown in **Listing 1**. Emerson is considering building a themes editor, which would make this job a lot easier.

Table 1. Themes included with ThemedControls.

ID	Name	Sample
1	Office 2003 Blue	
2	Office 2003 Olive	
3	Office 2003 Silver	
4	Office 2003 Black	

ID	Name	Sample
5	Office 2007 Blue	
6	Office 2007 Silver	

Listing 1. The XML that defines items in a theme consists of binary values (some of the text was omitted for brevity).

```
<ThemesDetails>
    <themeid>1</themeid>
    <membername>BUTTON_LEFT_FOCUSED</membername>
    <membertype>I</membertype>
    <membervalue>0</membervalue>
    <memberimage>Qk22BAAAAAAAADYEAAAoAAAAAgAAACAAAAABAAgAAAAAAAAAAAADEDgAAxA4AAABAAAA
AQAAli0A/1/C9/9kxPf/aMb4/23I+P9xyvj/ds34/3rP+f9+0fn/g9P5/4fV+f+f+M1/r/kNr6/5Xc+v+Z3vr/
neD7/6Li+/+m5Pv/q+f8/6/p/P+06/z/uO38/7zv/f/B8f3/xfT9/8r2/f/O+P7/0/r+/9f8/v/////AAAA
/wAAAP8AAAD/AAAA/wAAAP8AAAD/AAAA/wAAAP8AAAD/AAAA/wAAAP8AAAD/AAAA/wAA
AAAaAAAAGwAAABwAAAAAAAAdAAAA</memberimage>
    <imageext>BMP</imageext>
</ThemesDetails>
```

Let's dive into ThemesManager to see how it works. The Init method accepts a logical parameter which indicates whether you want to automatically initialize the manager (pass nothing or .F.) or manually call its Initialize method (pass .T.). There are a couple of reasons why you might want to use manual initialization:

- You want to use a different themes file than Themes.XML. The ThemesXMLFile property is .F. by default, which means the class uses Themes.XML. You can change that property and then call Initialize yourself to use a different file. However, you can also use multiple theme files by leaving ThemesXMLFile alone and setting AdditionalThemesXMLFiles to a comma-delimited list of files. As we'll see later, I like to set AdditionalThemesXMLFiles to ThemesSplitters.XML, which includes splitter skins in the six built-in themes.

- You want temporary files stored somewhere other than a Temp subdirectory of the current folder. ThemesManager caches the images it uses for the various components of each theme by writing them out to BMP files, such as Office 2003 Blue_BUTTON_LEFT_FOCUSED.BMP and Office 2003 Blue_BUTTON_LEFT_HOTTRACKING.BMP, in a temporary folder. By default, it uses a folder called Temp in the current directory. If it can't create that folder (for example, in Windows Vista or Windows 7 because the program folder is read-only), it then uses the user's temporary files folder. It uses these folders if the TempFolder property is empty, which it is by default. I actually like to use a subdirectory of the user's temporary files folder, so I set the TempFolder property to the desired folder before calling Initialize. Note that you're responsible for creating the directory yourself if you fill in TempFolder.

So, my code to instantiate and initialize the ThemesManager looks like this:

```
lcTempFolder = addbs(sys(2023)) + 'Themes'
if not directory(lcTempFolder)
   try
      md (lcTempFolder)
   catch
      lcTempFolder = ''
   endtry
endif not directory(lcTempFolder)
if type('_screen.ThemesManager') <> 'O'
   _screen.NewObject('ThemesManager', 'ThemesManager', ;
      'ThemedControls.vcx', '', .T.)
   with _screen.ThemesManager
      .AdditionalThemesXMLFiles = 'ThemesSplitters.xml'
      .TempFolder = lcTempFolder
      .Initialize()
   endwith
endif type('_screen.ThemesManager') <> 'O'
```

Notice that this adds a ThemesManager object to _screen. That's the preferred way of working with ThemesManager; as we'll see later, the various themed controls expect that. It also has the advantage of having a single, globally accessible ThemesManager object.

If you don't need to manually initialize ThemesManager for the reasons outlined above, you don't actually have to instantiate it at all. All of the ThemedControls classes check whether _screen.ThemesManager exists and if not, create it. Since ThemesManager.Init calls Initialize unless you pass .T., ThemesManager also initializes itself when invoked this way.

Initialize does a bunch of set up tasks, including loading and initializing GDIPlusX and Ctl32, opening some supporting procedure files, and loading the themes from the specified XML files by calling LoadThemes.

Once you've instantiated ThemesManager, you'll likely want to specify a theme. To do that, set ActiveTheme to the ID of the desired theme. To use Office 2003 Olive, for example, set ActiveTheme to 2. Changing ActiveTheme sets off a chain of events:

- ActiveTheme_Assign fires, which ensures the new value is reasonable (between 1 and ThemesCount, the number of loaded themes) and then calls ChangeTheme.
- ChangeTheme calls LoopThroughControls for _screen and for each of the open forms.
- LoopThroughControls goes through all of the controls in the specified form, including drilling down into containers, and calls the ChangeTheme method of each control if it exists.
- The ChangeTheme method of each control does something to change its appearance based on the new theme. Typically, it calls the GetMember method of _screen.ThemesManager to retrieve a specific image and set the Picture property of some Image control to that image. For example, ThemedForm contains an Image named imgBackground that fills the form so the form has a background skin. ThemedForm.ChangeTheme has this code:

```
with _Screen.ThemesManager
   This.imgBackground.Picture = .GetMember("Form.Background.Picture")
endwith
```

Thus, when you change ActiveTheme, any themed controls automatically refresh themselves with the appropriate image. This makes a nice effect: all of the forms in your application use the selected theme and all change at once. To see this in action, run ThemesDemo.SCX twice and change the theme in one of them. You should see the theme displayed in both forms change instantly.

If you want to use the current Windows theme, set the InheritWindowsTheme property to .T. It gets the current theme from Windows and sets ActiveTheme to the appropriate ID. However, it also sets up Windows event binding so it changes themes when you change the Windows theme.

You'll almost certainly want to allow the user to select their own theme. There are several ways you can do that:

- Have a control in some form (such as an Options dialog) that changes _screen.ThemesManager.ActiveTheme to the ID for the selected theme.
- Call _screen.ThemesManager.ShowPanel to display a dialog which allows the user to select a theme. You can pass the class and library containing the dialog to ShowPanel, or pass nothing, in which case it'll use the ThemesPanel class in ThemedControls.VCX, shown in **Figure 2**.

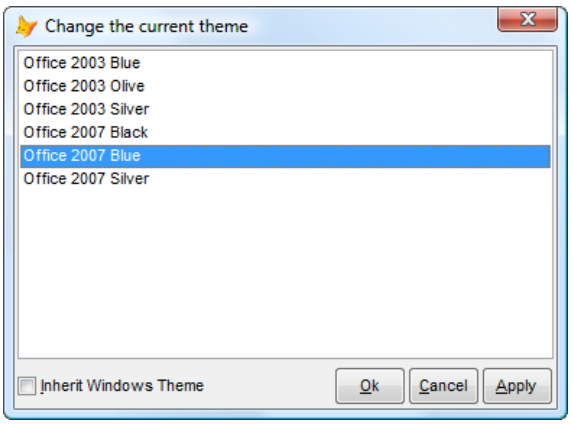


Figure 2. The ThemesPanel dialog allows the user to select a theme.

You'll need to save ActiveTheme when the user exits, such as to the Windows Registry or some preferences file, and restore it when they start the application.

Themed Controls

Before we look at specific ThemedControl classes, let's look at some common things all of these classes have. All ThemedControl classes contain the following two methods:

- InitThemedControl: this method, called from Init, creates a ThemesManager object if one doesn't already exist and then calls ChangeTheme.
- ChangeTheme: this method is called from both InitThemedControl so the control starts off using the current theme and from ThemesManager.LoopThroughControls when ActiveTheme changes. The code in this method is specific for the control; we saw the code in ThemedForm.ChangeTheme earlier. Typically, ChangeTheme calls _screen.ThemesManager.GetMember, passing it the name of a specific component, and sets the Picture property of an image or the color of some object to the return value.

One other thing themed controls have in common is an Image control that displays a specific image representing some component of the selected theme. For example, ThemedForm has an Image named imgBackGround that's sized to fill the form.

Now that you know the "secret" of how themes work, you can add themes support to your own existing classes rather than using ThemedControl classes if you wish. You don't actually need an InitThemedControl method; you can do those tasks anywhere, even directly in Init. However, you do need to add a ChangeTheme method, since that's what ThemesManager calls when the selected theme changes.

If you're not sure which version of ThemedControls you have, check the Version property of any control. The current version as of this writing is 3.5.8, released 07/15/2010.

One tip: make sure you SET LIBRARY TO at the end of your application. This isn't needed for runtime but if you don't do this and run your application in the VFP IDE, you'll get all kinds of errors the next time you run it because the VFP2C32 library isn't properly set up if it's open at startup.

ThemedForm

I've already discussed ThemedForm is some detail. You can use it for forms or as the base class for form classes when you want a form's background to display the current theme color. For example, ThemesDemo.SCX, shown in **Figure 3**, is based on ThemedForm. Of course, as I mentioned earlier, you don't have to use ThemedForm; if you'd rather use your own classes, simply replicate what ThemedForm does: add an Image control, size it so it fills the form in Init, and update its Picture property in a new ChangeTheme method.

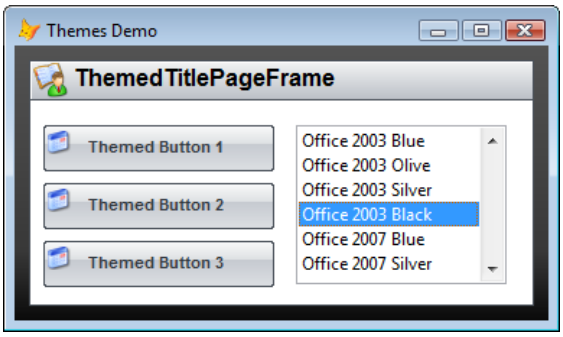


Figure 3. ThemedForm shows the current theme color in the background.

The only other special thing about ThemedForm is that it has a Loaded property and SetEnvironment method. Load calls SetEnvironment and sets Loaded to .T. SetEnvironment is a hook method you can add code to in an instance or subclass of ThemedForm to do any initialization before a ThemedOutlookNavBar or ThemedToolBox control that may be on the form initializes itself.

One thing I don't like about ThemedForm is that WindowState is set to 2-Maximized. I never create maximized forms but rather let the user control the window state, so I set it to the 0-Normal in instances or subclasses.

ThemedContainer

ThemedContainer provides a container whose border is colored according to the current theme. Its ChangeTheme method passes "Container.BorderColor" to GetMember and sets the BorderColor to the result. Since BorderWidth is only 1 by default, it's kind of subtle, but is more obvious with wider borders.

ThemedTitlePageFrame

ThemedTitlePageFrame is a themed pageframe with Tabs set to .F. so individual tabs don't appear. Each page has a title bar displaying an image and a title. The themed parts of the pageframe are its border and the page title bars. ThemesDemo.SCX includes a ThemedTitlePageFrame as you can see in **Figure 3**.

ThemedTitlePageFrame uses ThemedTitlePage as the class for pages (MemberClass and MemberClassLibrary contain ThemedTitlePage and ThemedControls.VCX, respectively). ThemedTitlePage is a subclass of Page with two controls added to it: TitleContainer, which is an instance of ThemedTitleContainer, and UserControls, an instance of Container. TitleContainer is the title at the top of the page and UserControls is where you add your own controls (although you can also add them directly to the page if you wish, adding them to UserControls makes sure they don't overlap with TitleContainer). Let's look at ThemedTitleContainer first.

ThemedTitleContainer

ThemedTitleContainer, shown in **Figure 4**, is a subclass of Control and contains several objects:

- imgBackground, an Image that provides the background for the title bar.
- imgTitle, an Image that displays the icon in the title bar.
- lblCaption, the caption for the title bar.
- linTitle, a Line under the title bar.

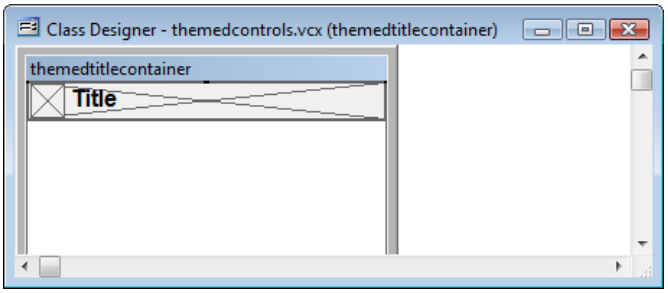


Figure 4. ThemedTitleContainer consists of several controls.

ThemedTitleContainer.ChangeTheme uses ThemeManager to change several things about the title bar:

- imgBackground.Picture uses the "Title.Background.Picture" member and Left uses "Title.Background.Left", meaning that the background may shift depending on the selected theme.
- lblCaption.ForeColor uses "Title.FontColor."
- BorderColor uses "Title.BorderColor."

ThemedTitleContainer has several custom properties with Assign methods so you don't have to drill down into the contained controls (which you actually can't do since this is based on Control) to set their properties:

- Caption: sets the caption of lblCaption.
- FontName: sets lblCaption.FontName.
- Picture24: sets imgTitle.Picture.

You can use ThemedTitleContainer by itself in forms if you want. However, since it's based on Control rather than Container, you can't add anything to it like you would a normal container, so it can only act as the title bar.

There's a bug in ThemedTitleContainer.Reposition in version 3.5.8 and earlier: it initially sizes imgBackground one pixel too narrow. Here's the fix:

```
With .imgBackground
   .Anchor = 0
*** DH 07/19/2010: width needs to be 1 pixel wider
***.Width = Max(This.Width - 3,0)
   .Width = Max(This.Width - 2,0)
   .Anchor = 11
Endwith
```

ThemedTitlePage

ThemedTitlePage has several properties you'll want to use:

- Caption: sets the caption for the page. Its Assign method sets TitleContainer.Caption as well, so the title bar of the page displays the desired caption.
- Picture24: its Assign method sets TitleContainer.Picture24 so the title bar displays the specified image. Because Emerson defined a property editor for Picture24, setting it is easy: simply double-click the property in the Properties window to display a GETPICT() dialog from which you can select the desired image. Note: there's a bug in the property editor script in version 3.5.8 and earlier: it references a non-existent Picture16 property instead. To fix this, open ThemedTitlePage in the Class Designer, bring up the MemberData Editor, select Picture24, change "Picture16" in the script to "Picture24," click OK to save the change, and save and close ThemedTitlePage.

ThemedTitlePage also has three properties, ClientAreaHeight, ClientAreaTop, and ClientAreaWidth, that aren't used for anything; they're referenced in ThemedTitlePageFrame.Resize but that code is commented out. I'm not sure if they're there for future use or some feature Emerson started to implement and then abandoned.

As I mentioned earlier, although you can add controls directly to the page, the intention is that you add them to the UserControls container on the page.

Using ThemedTitlePageFrame

Using ThemedTitlePageFrame is just like using a regular PageFrame with just a couple of differences:

- Drop a ThemedTitlePageFrame on a container of some kind, like a form or a class and size it as necessary.
- Set PageCount to the desired number of pages.
- For each page, set the Caption and Picture24 properties.
- Add controls to the UserControls container of each page.
- Provide some mechanism for the user to select a page; since Tabs is .F., they can't click a tab to select a page.

ThemedTitlePageFrameDemo.SCX, shown in **Figure 5**, provides an example of how to use ThemedTitlePageFrame in a form. The buttons, instances of ThemedButton we'll see next, provide the mechanism to select the desired page.

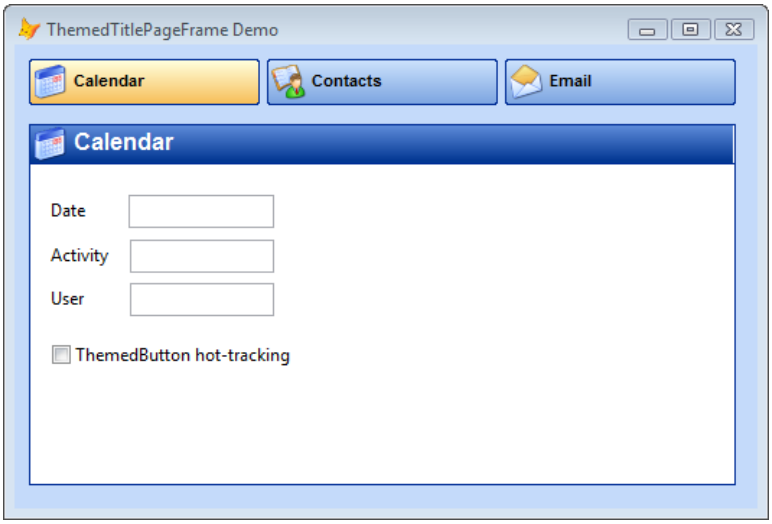


Figure 5. ThemedTitlePageFrameDemo.SCX shows how to use ThemedTitlePageFrame.

ThemedButton

ThemedButton provides a themed command button, including an image and label. It has a nice effect when you move the mouse over it: as you can see in **Figure 5** with the Calendar button, the background of the button changes, making it obvious which button you're over. The buttons in **Figure 3** and **Figure 5** are instances of ThemedButton. ThemedButtons provide an attractive alternative to boring CommandButtons.

If you look at the ThemedButton class, you'll notice that it's a subclass of Button, contained in Buttons.VCX. Button actually has all of the themed behavior, so I'm not sure why we need two classes. Instead, Button should be renamed to ThemedButton and moved to

ThemedControls.VCX. But I digress. For the rest of this section, when I refer to ThemedButton, I'm actually talking about Button.

ThemedButton, a subclass of Container, consists of the following controls:

- imgBackgroundLeft, imgBackgroundMiddle, and imgBackgroundRight: these Images are programmatically lined up across the width of the container (Resize does that; they're initially at left = 0, top = -20 so they're hidden at design time) and together form the background image. There are actually two images for each Image control, one for when the mouse is over the button and one when it isn't.
- cmdFocus: a CommandButton that, like the background images, starts up at top = -20 but is moved into position at runtime. The Click and KeyPress methods of cmdFocus pass control up to the same named method of the container.
- shpMouseHandler: this Shape also starts off at -20 and is moved into position. It provides the mouse handling effects: its MouseEnter and MouseLeave events call the container's SetImages method so the appropriate images are displayed.
- imgIcon: this Image contains the icon to display in the button
- lblCaption: this Label displays the caption for the button.

The Type property determines whether the button uses hot-tracking or not; 0 means normal and 1 means use hot-tracking. Hot-tracking means the button doesn't have a border and has a transparent background until you move the mouse over it. **Figure 6** shows this effect, with the mouse over the Calendar button.

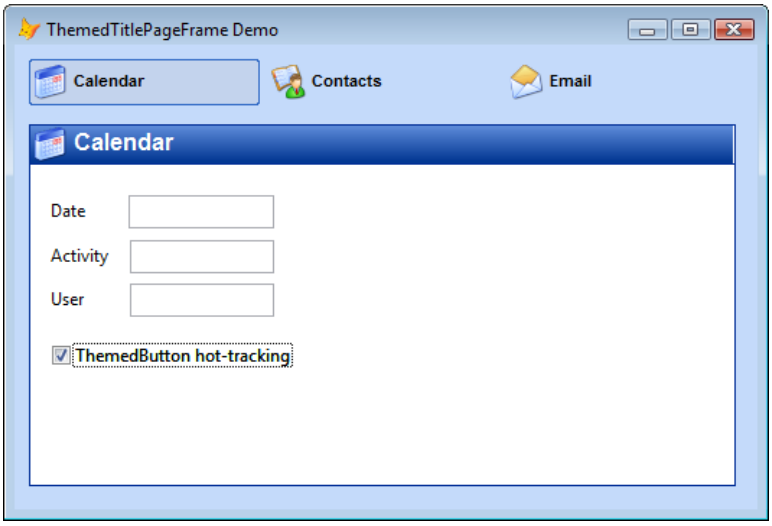


Figure 6. ThemedButtons with Type set to 1-Hot-Tracking have a different appearance.

We won't look at the themed elements used by ThemedButton; feel free to look at the code in ChangeTheme if you're interested.

Like shpMouseHandler.MouseEnter and MouseLeave, ChangeTheme calls SetImages to set the Visible and Picture property of the background image objects and the ForeColor property of lblCaption. If you decided to programmatically change Type, call SetImages to display the button correctly for the specified type. This could also be handled with an Assign method for Type.

To use ThemedButton, do the following:

- Drop a ThemedButton on a container of some kind, like a form or a class.
- Set lblCaption.Caption to the desired caption, imgIcon.Picture to the desired image (the default size is 24x24), and shpMouseHandler.ToolTipText to the desired tooltip. ThemedButton really should have Caption and Picture24 properties with Assign methods and an Assign method on ToolTipText that set lblCaption.Caption, imgIcon.Picture, and shpMouseHandler.ToolTipText so you don't have to drill down to access those properties. I created a subclass of ThemedButton called SFThemedButton (in SFThemedControls.VCX) that takes care of this. (It also has an Assign method for Type that calls SetImages.) Alternatively, you can use the ThemedButton builder that comes with ThemedControls (ButtonBuilder in ThemedControlsBuilders.VCX). To invoke the builder, double-click the CustomBuilder property in the Properties window. See **Figure 7**.
- Set Type to 1 if you want to use hot-tracking.
- Put the desired code into the Click method.

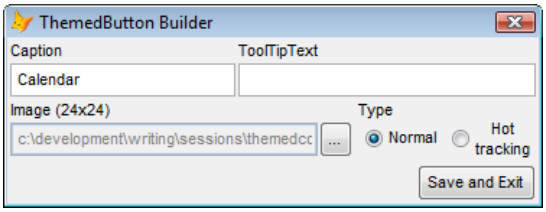


Figure 7. The ThemedButton Builder makes it easier to set the button's properties.

There's a bug in Button.SetImages in version 3.5.8 and earlier: if you change Type from 0 to 1 and then back to 0 again in code, calling SetImages after each change, the buttons won't appear properly because the imgBackground* images were made invisible when Type was 1. The code below shows the fix to make:

```
If This.Type==0
   If llFocus
      .imgBackgroundLeft.Picture = .ImgLeftFocused
      .imgBackgroundMiddle.Picture = .ImgMiddleFocused
      .imgBackgroundRight.Picture = .ImgRightFocused
   Else
      .imgBackgroundLeft.Picture = .ImgLeftNotFocused
      .imgBackgroundMiddle.Picture = .ImgMiddleNotFocused
      .imgBackgroundRight.Picture = .ImgRightNotFocused
   Endif
```

```
*** DH 07/17/2010: if Type is changed to 1 then back to 0, the images may not be
*** visible, so force it.
   Store .T. To ;
      .imgBackgroundLeft.Visible, ;
      .imgBackgroundMiddle.Visible, ;
      .imgBackgroundRight.Visible
*** DH 07/17/2010: end of new code
Else
   Store llFocus To ;
      .imgBackgroundLeft.Visible, ;
      .imgBackgroundMiddle.Visible, ;
      .imgBackgroundRight.Visible
Endif
```

ThemedExplorerBar

ThemedExplorerBar allows you to create an explorer bar, commonly seen in Windows XP Explorer windows but since abandoned in Windows Vista and later. Explorer bars are handy for displaying lists of grouped tasks, such as those shown in **Figure 8**.

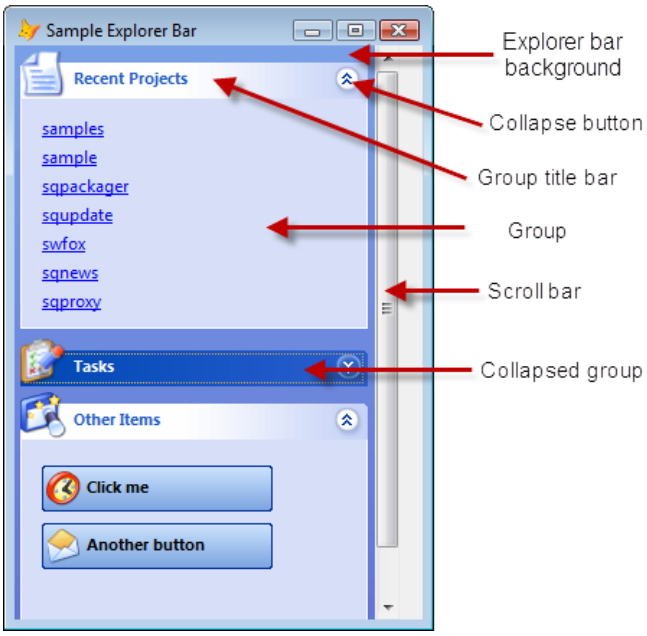


Figure 8. ThemedExplorerBar allows you to create explorer bar interfaces.

There are actually two classes involved in an explorer bar: ThemedExplorerBar, which is the overall bar, and ThemedExplorerGroup, which is an individual group. ThemedExplorerBar is really just a container for ThemedExplorerGroup controls, but provides features like a scroll bar, resize behavior, and so on.

Like ThemedButton, ThemedExplorerBar and ThemedExplorerGroup are subclasses of ExplorerBar and ExplorerGroup in ExplorerBar.VCX, which has all of the functionality,

including theme support. As before, when I discuss ThemedExplorerBar and ThemedExplorerGroup, I'm actually referring to ExplorerBar and ExplorerGroup.

An explorer bar has the following features:

- If the explorer bar isn't tall enough to display all groups, a vertical scroll bar appears.
- A group consists of a title bar, with icon, caption, and collapse button, and a user area where you can add any type of control. For example, in **Figure 8**, the Recent Projects group consists of hyperlinks (just VFP Label objects with FontUnderline set to .T. and ForeColor set to RGB(0, 0, 255)) while the Other Items group has a couple of ThemedButtons.
- Clicking the title bar or the collapse button of a group toggles between collapsed and expanded states. In **Figure 8**, the Tasks group is collapsed while the other two are expanded. The State property of a group indicates whether it's expanded (0) or collapsed (1). You can also set this property in the Properties window and it'll be respected when the explorer bar runs. Changing it programmatically, however, has no effect. Instead, call the Collapse or Expand methods.
- For a group, the color of the title bar, the color of the caption, the color and style of the collapse button, and the color of the background and borders reflect the current theme. For the explorer bar, the only themed component is the background color.
- The effect when you move the mouse over the title bar of a group depends on the theme. For the Office 2003 themes, the caption and collapse button change color. For the Office 2007 themes, those items also change color but the entire title bar becomes highlighted as well. Also, the collapse button looks different between the two sets of themes.
- The Type property of ThemedExplorerGroup can either be 0-Normal or 1-Special. A special group has a specially colored title bar. In **Figure 8**, the Tasks group is a special group, so it appears in dark blue when the Office 2003 Blue theme is used. If you change Type programmatically, you have to call ChangeTheme to redraw the title bar because there's no Assign method on that property.
- ThemedExplorerBar.Anchor is set to 7, which means the explorer bar expands vertically as the form is resized.

Like ThemedButton, ThemedExplorerGroup has a builder, ExplorerGroupBuilder in ThemedControlsBuilders.VCX, you can invoke by double-clicking the CustomBuilder property (**Figure 9**). Also like ThemedButton, ThemedExplorerGroup should have Caption and Picture24 properties with Assign methods so you can set one up programmatically without having to drill down to the contained components. My subclass, SFThemedExplorerGroup in SFThemedControls.VCX, implements that.

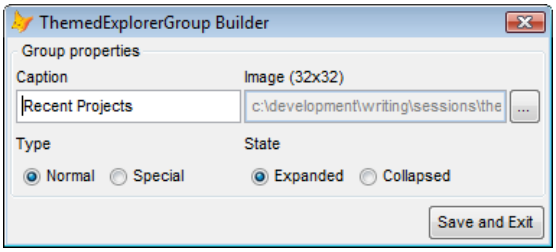


Figure 9. The ThemedExplorerGroup builder makes changing properties of a group easier.

To create an explorer bar, do the following:

- Drop a ThemedExplorerBar on a container of some kind, like a form or a class, or create a subclass.
- Drop a ThemedExplorerGroup on the bar, double-click the CustomBuilder property to invoke the builder, and set the properties of the group as desired. Alternatively, if you're using SFThemedExplorerGroup, set the Caption, Picture24, Type, and State properties.
- Right-click the group and choose Edit, click on the cntUserControls object (the large shape below the title bar), right-click and choose Edit. Drop any controls you want into cntUserControls.
- Add more groups as necessary.

You can, of course, do these steps programmatically, so you could use a data-driven explorer bar rather than a hard-coded one.

StartPageExplorerBar in Samples.VCX is a subclass of SFThemedExplorerBar (a subclass of ThemedExplorerBar that currently doesn't have any changes). It has three instances of SFThemedExplorerGroup, as you can see in **Figure 8**. The Tasks and Other Items groups were filled by dropping controls on them, but Recent Projects is filled programmatically using the code, taken from StartPageExplorerBar.GetRecentProjects, which is called from Init:

```
local llLockScreen, ;
   loContainer, ;
   lnSelect, ;
   laProjects[1], ;
   lnProjects, ;
   lnI, ;
   lcPath, ;
   lcName, ;
   lcLabel

* Lock the screen.

llLockScreen = Thisform.LockScreen
Thisform.LockScreen = .T.
```

```
* If we haven't already done so, add links to recent projects to the recent
* projects group.

loContainer = This.oRecentProjects.cntUserControls
if loContainer.ControlCount = 0
   lnSelect = select()
   select ;
      DATA ;
      from (set('RESOURCE', 1)) ;
      where TYPE = 'PREFW' and ID = 'MRUL' ;
      into cursor _PROJECTS
   lnProjects = alines(laProjects, DATA, 2, chr(0))
   for lnI = 2 to min(lnProjects, 8)
      lcPath  = laProjects[lnI]
      lcName  = juststem(lcPath)
      lcLabel = 'Project' + transform(lnI)
      loContainer.NewObject(lcLabel, 'ProjectLink', 'Samples.vcx')
      with loContainer.&lcLabel
         .Top          = (lnI - 2) * 20 + 13
         .Left         = 15
         .Caption      = lcName
         .cProjectPath = lcPath
         .ToolTipText  = lcName + ' (' + lcPath + ')'
         .Visible      = .T.
      endwith
   next lnI
   use
   use in FoxUser
   select (lnSelect)
endif loContainer.ControlCount = 0

* If there aren't any recent projects, collapse the group.

if loContainer.ControlCount = 0
   This.oRecentProjects.Collapse()
endif loContainer.ControlCount = 0

* Restore the screen.

Thisform.LockScreen = llLockScreen
```

This code opens your VFP resource file, finds the record maintaining the list of most recently used projects, and adds a ProjectLink object (a subclass of Label, also contained in Samples.VCX) for each project to oRecentProjects.cntUserControls. Given the spacing used between links, there's only enough room for seven links, so the code only processes that many projects. Thus, the group lists the seven most recently opened projects. The Click method of the ProjectLink objects raises the OpenProject event of StartPageExplorerBar, which doesn't contain any code in this class.

To check out StartPageExplorerBar, drop it on a form and put the following code into the Init method of the form:

```
bindevent(This.oBar, 'OpenProject', This, 'OpenProject')
```

Create a method called OpenProject with the following code:

```
lparameters tcProjectPath
messagebox(tcProjectPath)
```

Run the form and click on a project in the Recent Project group. You should see a message box displaying the path to the project.

There's a bug in ExplorerBar version 3.5.8 and earlier: imgBackground.Height isn't set properly until the form is resized for the first time. The fix is to make the following change to Init:

```
This.InitThemedControl()
*** DH 07/19/2010: call Resize or imgBackground isn't sized correctly
This.Resize()
*** DH 07/19/2010: end of new code
This.Reposition(.T.)
```

ThemedOutlookNavBar

ThemedOutlookNavBar allows you to implement a Microsoft Outlook-like control in your applications. **Figure 10** shows an example, taken from BasicOutlookBarDemo.SCX.

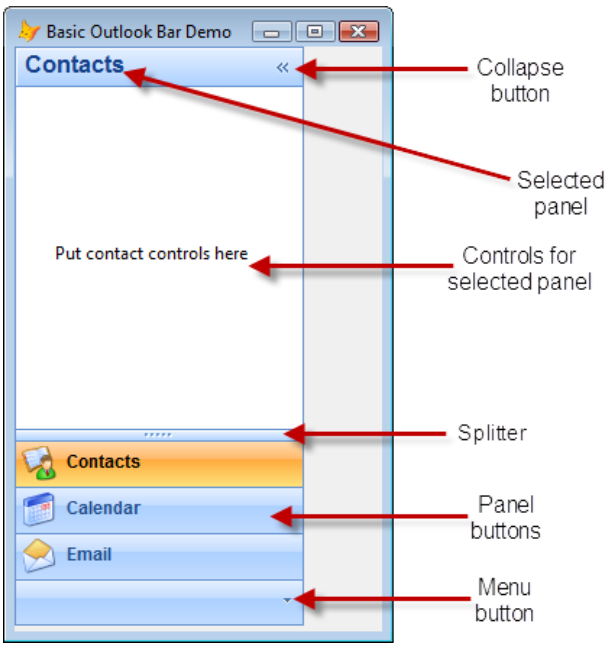


Figure 10. The ThemedOutlookNavBar control has an Outlook-like appearance.

Here's how the Outlook bar works:

- At the top of the control is the title bar for the selected panel. It shows the panel caption.

- Below the title bar is the selected panel itself. You can place any controls you want to into the panel since essentially it's a page in a pageframe.
- Below the selected panel is the panel buttons area. This has one button per defined panel. The selected panels button has a different appearance than the others (the difference depends on the current theme).
- To select a panel, click its button. That panel then appears at the top of the control. If the panel has a hotkey letter assigned to it, you can also press the letter to select the panel. You can also click the menu button and select the desired panel from the menu.
- Between the selected panel and the panel buttons area is a splitter. To change the height of the selected panel, drag the splitter up or down. Dragging it down increases the height of the selected panel and reduces the number of panel buttons while dragging up does the opposite. Another way to do this is to click the menu button and select Show More to show more panel bars or Show Less to show fewer panel bars.
- Panel bars that were removed because you increased the height of the selected panel are still available: they are displayed as icons in the overflow area at the bottom of the control and in the menu (see **Figure 11**). To select a panel, click its icon or click the menu button and select the desired panel from the menu.

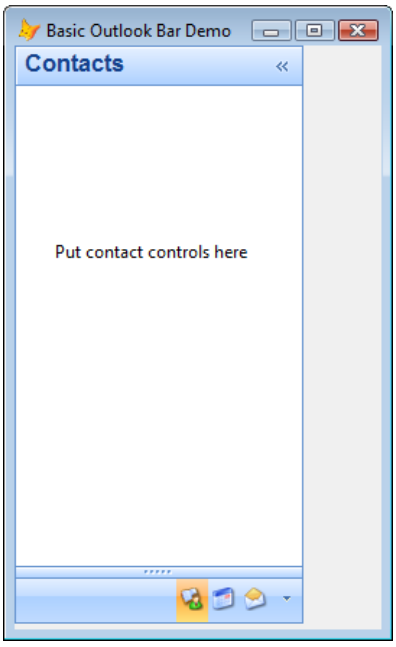


Figure 11. Increasing the height of the selected panel using the splitter causes fewer panel buttons to be displayed.

- The menu also lists the available themes and an "Inherit Windows Theme" function so you can change the theme.

- You can collapse the Outlook bar by clicking the collapse button at the top of the control (**Figure 12**). Click it again to expand the panel control.

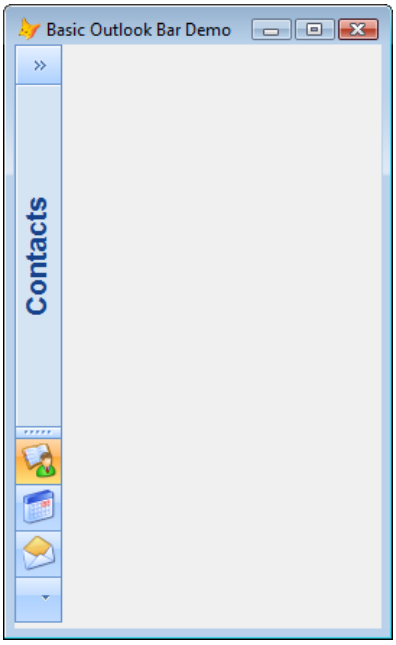


Figure 12. Click the collapse button to toggle between collapsed and expanded states.

- ThemedOutlookNavBar automatically changes its form's MinHeight property so the form can't be resized shorter than the minimum height the bar needs to display properly. Anchor is set so the bar expands vertically as the form is resized.

Like ThemedButton, ThemedOutlookNavBar is a subclass of OutlookNavBar in OutlookNavBar.VCX, which has all of the functionality, including theme support. As before, when I discuss ThemedOutlookNavBar, I'm actually referring to OutlookNavBar.

As you can probably guess, ThemedOutlookNavBar is actually a pretty complex control. However, it's quite easy to work with. For the most part, you add panels, either visually or programmatically, and add controls to those panels. A panel is actually a page in a pageframe, so you can add any controls you wish. The rest of the Outlook bar is taken care of by the class.

As with ThemedExplorerBar, I'm not going to go through the theme elements and which components they apply to because it's fairly complex and not really all that important to know for working with the control.

To create an Outlook bar, do the following:

- Drop a ThemedOutlookNavBar on a container of some kind, like a form or a class, or create a subclass. Let's call it oOutlookBar for the purposes of this document.

- Normally, the Outlook bar is fills the height of the form, so set oOutlookBar.Height to match that of the form. Anchoring is already set so it expands vertically when you resize the form.
- Set oOutlookBar.Panes.PageCount to the number of panels you want.
- For each panel (that is, each page in Panes), set the Caption, Picture16, and Picture24 properties to the caption, the image that appears in the overflow panel and menu, and the image that appears in the panel button for the panel. You can also set Hotkey to the letter that's the hotkey for the panel; note that Hotkey is case-sensitive, so if the caption is "Test" and "e" should be the hotkey, use "e" rather than "E."
- Add any desired controls to each page. Note that the controls shouldn't be taller or wider than oOutlook.Panes; for example, if you add a container of controls, be sure to size it so Height <= oOutlook.Panes.Height and Width < oOutlook.Panes.Width.

You can, of course, do these steps programmatically, so you could use a data-driven Outlook bar rather than a hard-coded one. Use the AddButton method to add a new panel to the Outlook control. For example, suppose you have a table that has one record per panel. It contains CAPTION, PICTURE16, PICTURE24, and HOTKEY fields that contain the values for the appropriate properties, and CLASS and LIBRARY fields for the name of a container class to add to the panel. This code would fill the Outlook bar programmatically:

```
with This.oOutlook
   scan
      .AddButton(CAPTION, PICTURE16, PICTURE24, HOTKEY)
      loPane = .Panes.Pages[.Panes.PageCount]
      loPane.NewObject('oControl', trim(CLASS), trim(LIBRARY))
      loPane.oControl.Visible = .T.
   endscan
endwith
```

I've created a subclass of ThemedOutlookNavBar called SFThemedOutlookNavBar (in SFThemedControls.VCX) that overrides AddButton to return the newly added page, so this code can be simplified a bit:

```
with This.oOutlook
   scan
      loPane = .AddButton(CAPTION, PICTURE16, PICTURE24, HOTKEY)
      loPane.NewObject('oControl', trim(CLASS), trim(LIBRARY))
      loPane.oControl.Visible = .T.
   endscan
endwith
```

Besides Panes.PageCount and AddButton, there are several other properties and methods you might find useful:

- SelectedButton contains the number of the selected panel (1 being the first panel, 2 the second, and so on). Changing SelectedButton changes the selected panel, but thanks to its Assign method, also fires some events you might want to tie into:
  - BeforeChangeSelectedButton fires before the value of SelectedButton changes. It's passed the current value of SelectedButton and the value it's about to change to. If you return .F. from BeforeChangeSelectedButton, the selected panel doesn't change. This might be a good place to put some validation code that prevents the user from changing panels until they've finished what they're doing with the current panel.
  - ButtonClicked fires after the selected panel has changed. It's passed the new value of SelectedButton as well as the Caption and Picture24 values of the selected panel. You can use this event to refresh the form, such as displaying properties at the right for the selected panel.

  You might want to save and restore the value of SelectedButton so the next time the user opens the form, the same panel is selected as last time.

- Calling ShowLess is a programmatic way of moving the splitter bar down, showing one less button. Similarly, ShowMore moves the splitter bar up and displays one more button.
- ChangeViewMode toggles between collapsed and expanded states. It fires the ViewModeChanged event, passing it .T. if the bar appears collapsed, which allows you to perform some additional actions. For example, you might want to change the Left and Width properties of controls to the right of the Outlook bar so they fill the space left when the bar collapses; after all, that's really the reason why the user would click the collapse button in the first place. Another example is if you include a ThemedExplorerBar in a panel of ThemedOutlookNavBar; in that case, you need to hide the explorer bar's scroll bar when the Outlook bar is collapsed. In the following code, an instance of a ThemedExplorerBar named oControl is in the first panel:

  ```
  lparameters tlShrunk
  if This.SelectedButton = 1 and tlShrunk
     This.Panes.Pages[1].oControl.Ctl32_Scrollbar.ctlVisible = .F.
  endif This.SelectedButton = 1 ...
  ```

- ShowedButtons contains the number of panel buttons being displayed. As with SelectedButton, you may wish to save and restore its value so the position of the splitter is the same as it was last time. Saving is easy, but restoring is a little more complicated because changing the value of ShowedButtons doesn't do anything. In that case, you need to do something like this:

  ```
  do while .oNavBar.ShowedButtons > ValueToRestoreTo
     .oNavBar.ShowLess()
  enddo while .oNavBar.ShowedButtons > ValueToRestoreTo
  ```

- MaxShowedButtons shows the maximum number of panels that can be displayed; the default is 5. Additional panels beyond this value appear in the overflow area. Set the property as desired.

- UpdatePane changes the properties of the specified pane. Pass it the pane number and the values for Caption, Picture16, Picture24, and Hotkey.

A related class is ThemedOutlookNavBarTbr, a Toolbar with a ThemedOutlookNavBar inside it, which you can use if you want a toolbar-based Outlook control.

There's a bug in the property editor script for the Picture24 property of the Pane class in version 3.5.8 and earlier: it incorrectly references Picture16. To fix this, open Pane in the Class Designer, bring up the MemberData Editor, select Picture24, change "Picture16" in the script to "Picture24," and click OK to save the change. One other change: reset the Picture16 and Picture24 properties to default or the property editor still doesn't seem to have any effect.

One bug I haven't found a fix for (but have reported to Emerson) is that moving the mouse over the splitter bar sometimes moves the splitter bar down even though the mouse button isn't pressed. One way to see this effect is to bring a top-level form up when a form with a ThemedOutlookNavBar is already open, then close it. Moving the mouse over the splitter bar now exhibits this effect. I suspect it's a Windows event binding issue but haven't managed to track it down.

ThemedToolbox

The ThemedToolbox control has a UI similar to the VFP Toolbox or the Toolbox in older versions of Visual Studio. Its purpose is very similar to ThemedOutlookNavBar: providing panels of different categories and controls inside those panels. However, as you can see in **Figure 13** (taken from ThemedToolboxDemo.SCX), the UI is more spartan: panel buttons with a "+" or "-" to indicate whether it's expanded or not. Another difference is that clicking a panel button doesn't move it up to the top; rather, it collapses the previously selected panel and expands the current one, with collapsed panel buttons stacked on top of each other. Also, unlike ThemedOutlookNavBar, ThemedToolbox expands horizontally as well as vertically when the form is resized.

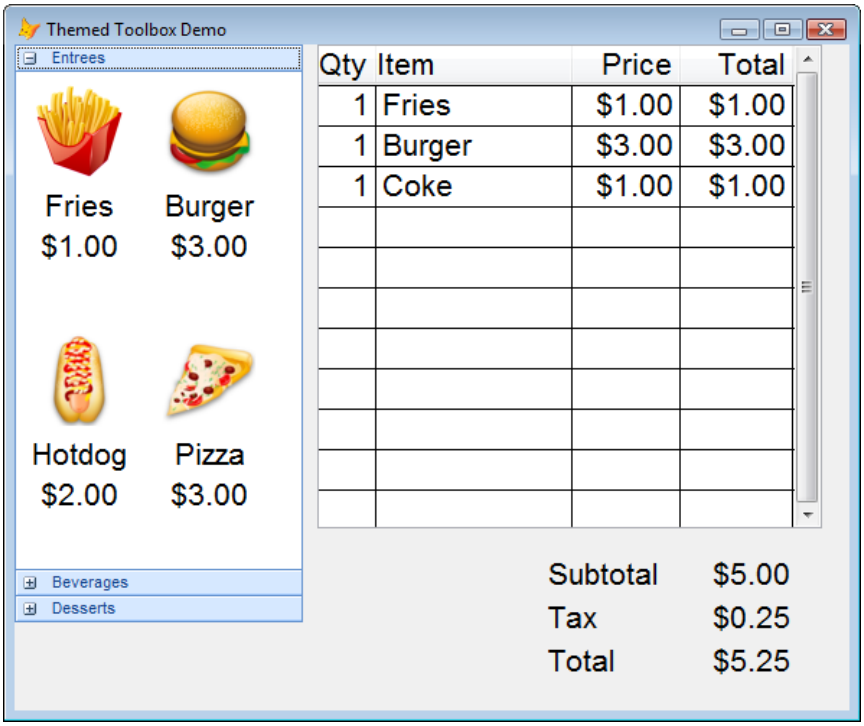


Figure 13. ThemedToolbox is useful for selecting from categories of items.

As we've seen with other classes, ThemedToolbox is a subclass without any extra features. In this case, it's a subclass of Toolbox in Toolbox.VCX. There are several other supporting classes in Toolbar.VCX: Panes, a PageFrame subclass (like ThemedOutlookBar, ThemedToolbox is essentially a fancy pageframe); Pane, a subclass of Page used as the pages in the pageframe; and Title, which provides the panel buttons.

A related class is ThemedToolboxTbr, a Toolbar with a ThemedToolbox inside it, which you can use if you want a toolbar-based toolbox.

Using ThemedToolbox is easier than ThemedOutlookNavBar:

- Drop a ThemedToolbox on a container of some kind, like a form or a class, or create a subclass.
- Set Panes.PageCount to the desired number of panels.
- For each panel, set the Caption of the Pane (page) object and add any controls necessary.

There's really only one property and method of interest. Wrap determines whether focus goes to the top panel (.T.) or the previous one (.F.) when the bottom panel is collapsed; the default is .T. TitleClicked fires when the user clicks a panel button.

Given the choice, I'd choose to use ThemedOutlookNavBar over ThemedToolbox because of its richer UI. However, ThemedToolbox might be appropriate for simpler developer tools.

ThemedZoomNavBar

ThemedZoomNavBar, shown in **Figure 14** (taken from ThemedZoomNavBarDemo.SCX), provides a UI similar to the Mac OS X Leopard desktop. I think this UI provides an attractive and dynamic alternative to toolbars. The large buttons are easy to click, even with a finger on a touch screen.

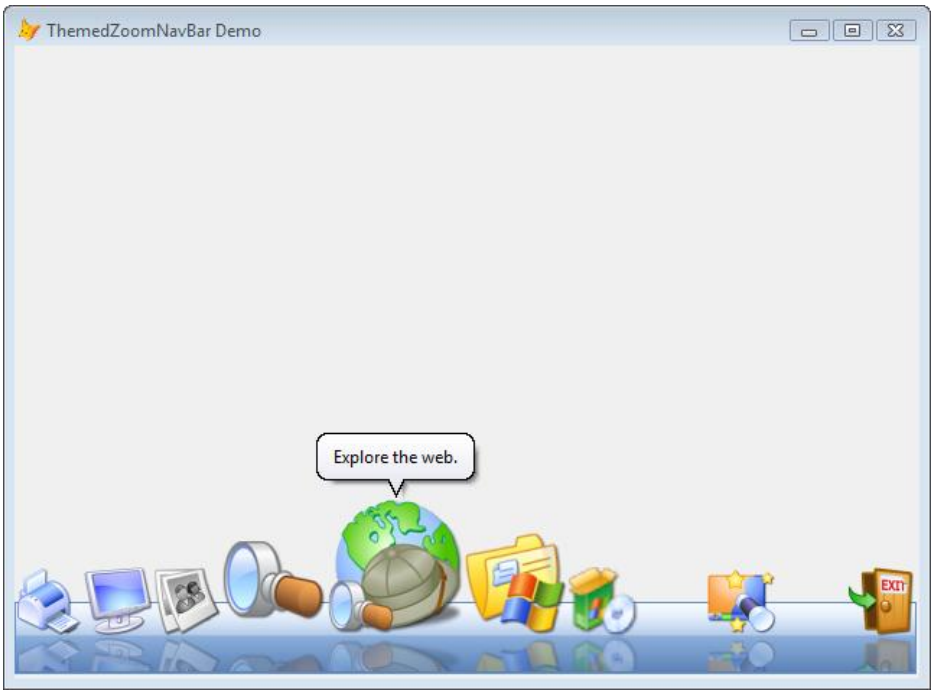


Figure 14. ThemedZoomNavBar can be used for attractive toolbars.

ThemedZoomNavBar has the following features:

- The control appears as a set of images sitting on top of a strip. The strip has two parts: the top part sits under the images and the bottom part appears in a darker color of the current theme and displays reflections of the images.
- Images initially appear at 48x48 pixels. When you move the mouse over an image, it zooms up to 96x96 (hence the name of the class) and images on either side of it zoom to 72x72. Because of this, you'll want to use 96x96 or bigger images or else they'll appear grainy when zoomed up. You can see this if you move your mouse over the image of the software box with the CD.
- Images are actually buttons. Clicking one fires the ButtonClicked event for the ThemedZoomNavBar control, passing it the name of the button. Typically, you'll use a CASE statement to determine what action to take based on the button name.
- An image can have a tooltip. However, as you can see in **Figure 14**, it's not a normal tooltip but a Ctl32_BalloonTip tooltip. Ctl32_BalloonTip, one of the classes in Ctl32.VCX that's required by the ThemedControls, provides much more capable

tooltips than VFP does. Notice, for example, the balloon shape of the tooltip window. For details on Ctl32_BalloonTip, see http://tinyurl.com/23dfgna.

- Separator buttons provide space between buttons, such the space to the left of the Exit button in **Figure 14**.
- You can place the control at the top or bottom of a form. It can also sit in a toolbar. In fact, ThemedZoomNavBarTbr is a Toolbar subclass that contains a ThemedZoomNavBar, so you can subclass it if desired.
- In addition to buttons across the control, you can also have "stacked" buttons. **Figure 15** shows an example. Stacked buttons are associated with a regular button; clicking that button displays the stacked buttons. Stacked buttons do not zoom as you move the mouse over them, but clicking them fires the StackButtonClicked event, passing it the name of the stacked button. As with ButtonClicked, you'll likely use a CASE statement in that event. Note that if you use stacked buttons, you have to put the ThemedZoomNavBar at the bottom of the form or else there isn't room for the stacked buttons to appear.

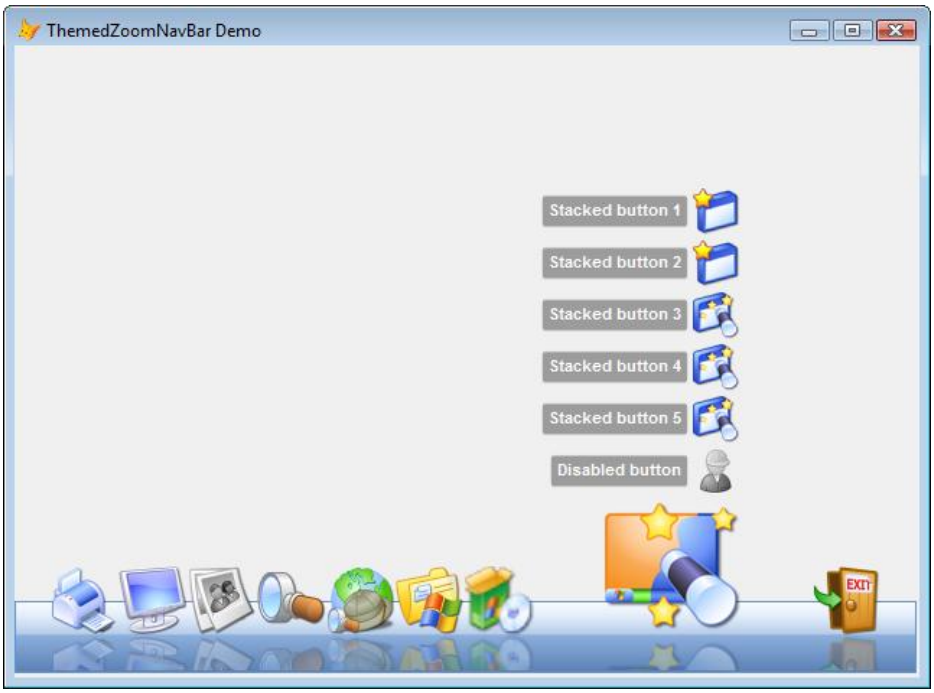


Figure 15. ThemedZoomNavBar buttons can include stacked buttons.

- Buttons and stacked buttons can be disabled, either by passing a parameter to the AddButton or AddStackButton methods (which I'll discuss later) or by calling SetButtonState(cButtonName, lEnabled) or SetStackButtonState(cParentButtonName, cStackButtonName, lEnabled).
- Normally, the control respects the setting of Width, so it's only as wide as you make it. However, if you set the Stretch property to .T., the ThemedZoomNavBar automatically fills the width of its form.

Like some of the other controls we've seen, ThemedZoomNavBar is a subclass without any additional behavior. Its parent is ZoomNavBar in ZoomNavBar.VCX. That class library contains several supporting classes as well: ZoomImage, which is used for buttons, StackContainer and StackButton, which are used for stacked buttons, ReflectedImage, used for reflected images, and SeparatorImage, used for separator buttons.

To use ThemedZoomNavBar:

- Drop a ThemedZoomNavBar on a container of some kind, like a form or a class, or create a subclass.
- Add buttons using the builder: double-click the CustomBuilder property in the Properties window to invoke the builder (ZoomBarBarBuilder in ThemedControlsBuilder.VCX) shown in **Figure 16**.

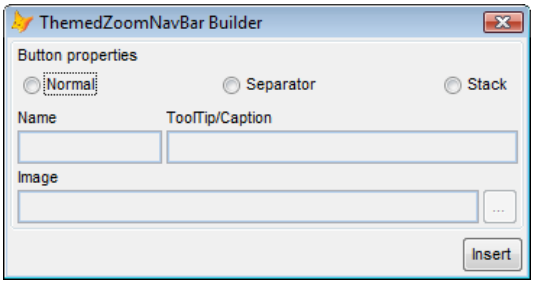


Figure 16. ThemedZoomNavBar comes with a builder to add buttons.

- Alternatively, you can call the AddButton method to programmatically add a button. Pass it the button name, which must be unique, the text to use for the tooltip, the name of the image file to use, the name of the reflected image to use (optional; it you don't pass it, ThemedZoomNavBar used GDIPlusX to create a temporary image), the name of the disabled version of the image (again, optional), the name of the disabled version of the reflected image (also optional), and .F. if the button should be disabled.
- To programmatically create stacked buttons, call AddStackButton, passing it the name of the parent button, the name of the stacked button, the caption that appears at the left, the image file to use, the disabled image to use (optional), and .F. if the button should be disabled.
- Put code into the ButtonClicked and StackButtonClicked events to handle clicks on buttons and stacked buttons. Both of these methods are passed the name of the clicked button.

Here's the code from the Init method of the ThemedZoomNavBar control in ThemedZoomNavBarDemo.SCX; this code was ripped off, er, adapted from one of Emerson's sample forms.

```
local lcDisableMessage, ;
   lcImageMessage
```

```
dodefault()
lcDisableMessage = 'You can disable/enable a button either by passing a ' + ;
   'parameter to the AddButton and AddStackButton methods or using the ' + ;
   'SetButtonState and SetStackButtonState methods.'
lcImageMessage   = 'This shows why you should use 96x96 images. Otherwise, ' + ;
   'when the image is resized to 96x96, it looks grainy.'
with This
   .AddButton('Printers', 'Manage printers.',         'Printer96.png')
   .AddButton('Display',  'Change display settings.', 'Monitor96.png')
   .AddButton('Photo',    lcDisableMessage,           'Photo96.png', , , , .F.)
   .AddButton('Search',   'Search for files.',        'Search96.png')
   .AddButton('Browser',  'Explore the web.',         'Explorer96.png')
   .AddButton('Explorer', 'Show files and folders.',  'Folder96.png')
   .AddButton('Software', lcImageMessage,             'Software32.png')

   .AddButton('Separator')

   .AddButton('Samples', 'Click here to show stacked buttons', 'Wizard96.png')
   .AddStackButton('Samples', 'Button1', 'Stacked button 1', 'NewWindow32.png')
   .AddStackButton('Samples', 'Button2', 'Stacked button 2', 'NewWindow32.png')
   .AddStackButton('Samples', 'Button3', 'Stacked button 3', 'Extensions32.png')
   .AddStackButton('Samples', 'Button4', 'Stacked button 4', 'Extensions32.png')
   .AddStackButton('Samples', 'Button5', 'Stacked button 5', 'Extensions32.png')
   .AddStackButton('Samples', 'Button6', 'Disabled button',  'Worker32.png', , .F.)

   .AddButton('Separator')

   .AddButton('Exit', 'Click here to exit.', 'OpenedDoor96.png')
endwith
```

This is the code in the ButtonClicked event; in a real form, this code would probably use a CASE statement to decide what action to take based on the name of the clicked button.

```
lparameters tcName
if tcName = 'Exit'
   Thisform.Release()
else
   messagebox('You clicked ' + tcName)
endif tcName = 'Exit'
```

One downside of ThemedZoomNavBar is that it takes up a fair bit of room. It's 125 pixels tall and as wide as it needs to be for the number of large buttons displayed. However, on a navigation-type form, that wouldn't be a problem.

Ribbon

Ribbon.VCX provides a ribbon navigation control similar to that in Microsoft Office 2007 and later. Ribbon is still in development, so we'll take a cursory rather than in-depth look at it.

A ribbon consists of several parts as you can see in **Figure 17**, taken from Microsoft Word. The application button is actually a menu, similar to the Windows Start button. The tabs

and group are essentially a pageframe with a slightly different UI. The dialog button displays a tooltip when you hover the mouse button over it and brings up a dialog when you click it.

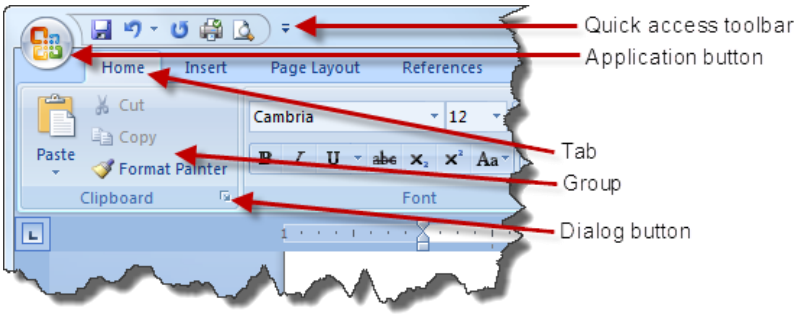


Figure 17. The Microsoft Office ribbon is a complex navigation control.

As you can see in **Figure 18**, the ThemedControls Ribbon matches the Office ribbon fairly closely.

Figure 18. The ThemedControls Ribbon control provides an interface like the Microsoft Office ribbon.

The ribbon actually consists of several controls:

- CntForm: provides the appearance of a form: a custom title bar, including custom minimize, maximize, and close buttons, and a themed background.
- QuickAccessToolbar: as its name implies, it provides the quick access toolbar, including the curved shapes in the border.
- ApplicationButton: again, its name makes it obvious what this class is for.
- Ribbon: the tabs and groups part of the ribbon.
- Group: provides a group container, including dialog button.
- RibbonButton: used for buttons in the ribbon.
- Spacer: provides a vertical space between controls, such as that between the Pen and Email buttons in Group Two in **Figure 18**.

I'm not going to discuss how to use the ribbon control because, as it's still in development, the details might change completely.

ThemedControls includes a sample form, Ribbon.SCX, which demonstrates the ribbon control UI.

Summary

There's no excuse for creating a boring looking VFP application. Using several of the VFPX projects, including ThemedControls, you can create a new, modern user interface for your forms that'll add years to the life of your applications. Your forms can resemble Microsoft Office applications so your users will feel more at home using your applications. With a few days of effort, your apps can be as pretty as any .Net application. Get started today!

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).