



50 Shades of Grey: Whipping Your Application's UI into Shape

Doug Hennig

Stonefield Software Inc.

Email: dhennig@stonefield.com

Corporate Web sites: www.stonefieldquery.com

www.stonefieldsoftware.com

Personal Web site : www.DougHennig.com

Blog: DougHennig.BlogSpot.com

Twitter: [DougHennig](https://twitter.com/DougHennig)

There's no excuse for creating a boring looking VFP application. Using some of the controls available today, you can create a new, modern user interface for your forms that'll add years to the life of your applications. With a few days of effort, your apps can be as pretty as anything out there. This document looks at several new controls that allow you to freshen your user interface and wash out the grey.

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 up), 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://vfp.codeplex.com>) to provide modern-style menus, toolbars, and other graphical user interface elements in your applications.

This document focuses on controls that have a more modern interface than the versions you've probably used in the past. Some of them are VFPX projects while others are classes I or others created. Specifically, we're going to look at gauges, toolbar buttons, a tabbed document interface, and a library of controls that replace the VFP TextBox, EditBox, CheckBox, OptionGroup, CommandButton, PageFrame, and window title bar with more attractive alternatives.

Adding gauges to your applications

People like visual images. Most people would rather see a chart than columns of raw numbers because it's easier to see the relationships between items visually. Adding analysis tools like charts and gauges to your applications make them much more valuable to your users.

What is a gauge? A gauge is an image that shows how a single value compares to a maximum or goal value. The two values can be anything: current sales compared to budget, volume to date compared to the maximum allowable volume, and so on. For example, many charitable organizations show the current status of their fund raising campaigns as a thermometer. The top of the thermometer represents the value of the fund raising goal and the height of the bar inside the thermometer represents how much money has been raised so far.

The most common type of gauge looks like a speedometer in a car; see **Figure 1** for an example. The end of the gauge represents the maximum value and the position of the needle represents the current value. Color bands around the outside edge of the gauge show the ranges of certain categories. For example, in Figure 1, the red band indicates where sales are too low, the yellow band where they're acceptable but not great, and the green band where sales should be to make the boss happy.



Figure 1. It's easy to draw beautiful gauges using the Gauge class.

We recently added support for gauges to my company's flagship product, Stonefield Query (www.stonefieldquery.com). In this document, I'll show you how we did it.

The Gauge class

There's just a single class used to draw gauges: Gauge in Gauge.vcx (additional components are also required as I'll discuss later). It's a subclass of Custom so it has no visible appearance at runtime. How does the gauge appear then? After you call the DrawGauge method, the cImage property is set to the bytes for the gauge image. You can use FILETOSTR() to write the contents of cImage to a file, such as if you want to use the gauge in a report, or set the PictureVal property of an Image object if you want it to appear in a form. For example, SampleGauge.scx, one of the forms included in the samples for this document, uses this code to have the Gauge object referenced in the oGauge property draw a gauge in an Image named imgGauge:

```
with This
    .oGauge.nSize = .imgGauge.Width
    .oGauge.DrawGauge()
    .imgGauge.PictureVal = .oGauge.cImage
endwith
```

Table 1 lists the properties of the Gauge class. As you can see, there are quite a few of them. Most of them affect the appearance of the gauge.

Table 1. The properties of the Gauge class.

Property	Description
cDialText	The text for the dial
cDialTextFontName	The font for the dial text
cErrorMessage	The text of any error that occurs

Property	Description
cFormat	The format for the labels in .NET syntax: {0:#,##0} by default
cImage	The gauge image
cLabelFontName	The font for the labels
lAdjustLabelSize	.T. to adjust the label size based on the gauge size, .F. to use the size specified in nLabelFontSize
lDialTextFontBold	.T. if the dial text is bold
lDialTextFontItalic	.T. if the dial text is italics
lDisplayDigitalValue	.T. to display the value in a digital display
lLabelFontBold	.T. if the label text is bold
lLabelFontItalic	.T. if the label text is italics
lValuesAsPercentages	.T. to use percentages for values, .F. to use amounts for values
nBackColor	The background color or the starting color for a background gradient
nBackColor2	The end color for a background gradient; if it's the same as nBackColor, there is no gradient
nBackColorAlpha	The alpha for the background color
nBackGradientMode	The mode for a background gradient: 0 = left to right, 1 = top to bottom, 2 = from top left, 3 = from top right
nBand1Color	The color for band 1
nBand1End	The ending position for band 1
nBand2Color	The color for band 2
nBand2End	The ending position for band 2
nBand3Color	The color for band 3
nDialAlpha	The alpha for the dial color
nDialColor	The color to use for the dial
nDialTextColor	The color for the dial text
nDialTextFontSize	The font size for the dial text
nDigitsColor	The color for digital digits
nGlossiness	The glossiness value (0 - 100)
nGoalPosition	The position where the goal value appears on the gauge; defaults to 100
nLabelColor	The color to use for labels
nLabelDistance	The distance between labels and major tick marks
nLabelFactor	The factor to use for labels: 1, 1000, 10000, etc.
nLabelFontSize	The font size for the labels
nMajorTickColor	The color of major tick marks
nMajorTickCount	The number of major ticks
nMaxValue	The goal value for the gauge
nMinorTickColor	The color of minor tick marks
nMinorTickCount	The number of minor ticks
nSize	The height and width of the gauge (it's a square so they're the same)
nValue	The current value for the gauge
oBridge	A reference to a wwDotNetBridge object
oGauge	A reference to a .NET GaugeControl object

The best way to check out how the various properties work is by running SampleGauge.scx, shown in **Figure 2**. Each control has a tooltip specifying which property it controls. Changing any setting immediately redraws the gauge so you can instantly see the effect.

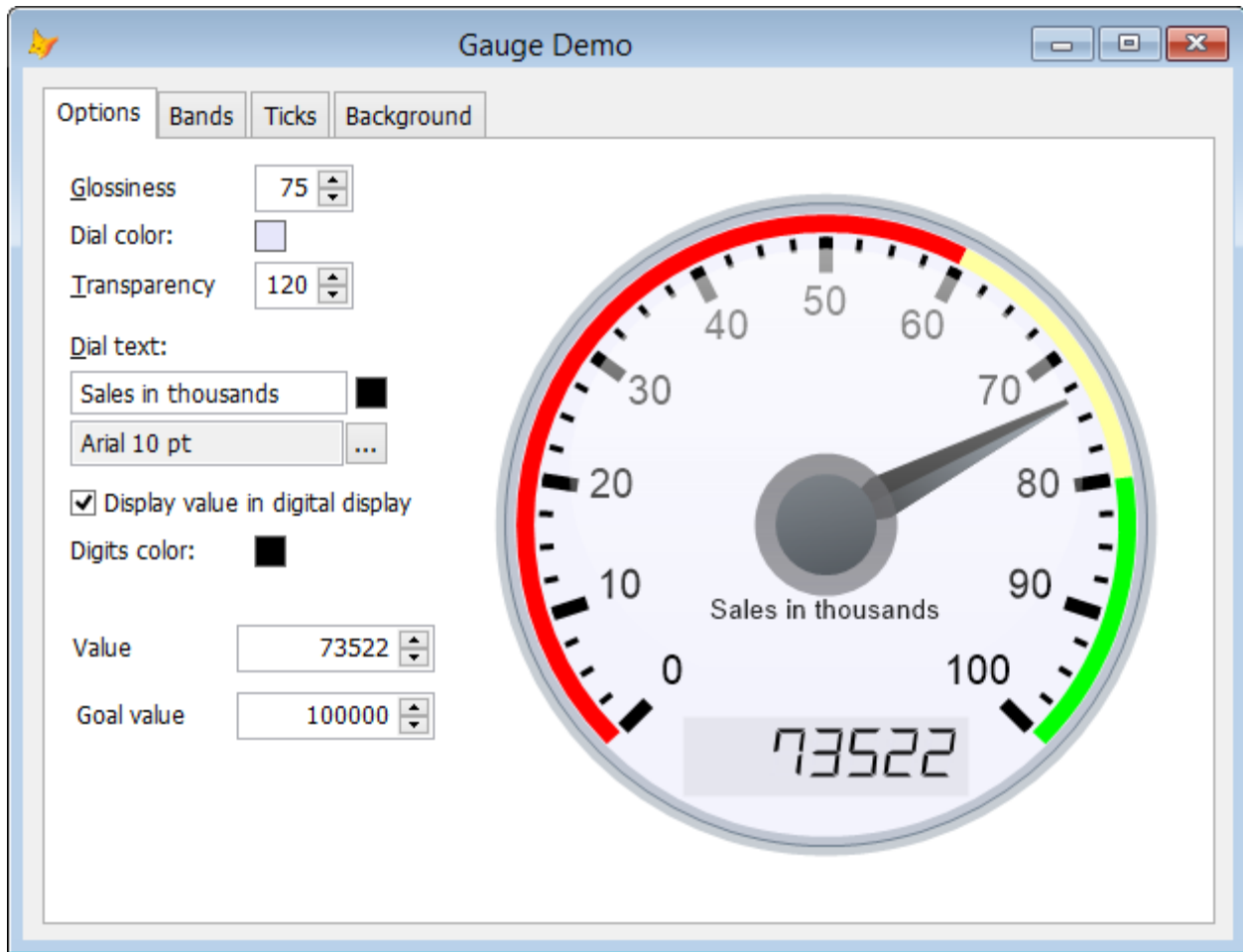


Figure 2. SampleGauge.scx allows you to experiment with the properties of the Gauge object.

Here are comments about some of the properties:

- `nGlossiness` controls how bright the “glossy” ellipse that appears at the top of the gauge is. This ellipse gives the illusion of reflected light, as if the gauge was made of glass or plastic.
- The gauge size is determined by the `nSize` property; since a gauge is drawn as a square, the height and width of the image are both set to `nSize`.
- By default, the labels indicating the values around the gauge are sized based on the size of the gauge: the larger the gauge, the bigger the labels. Set `lAdjustLabelSize` to `.F.` if you want to use the font size specified in `nLabelFontSize` instead.
- `cFormat` indicates how the labels are formatted. It has to use .NET syntax for number formats for reasons that will be obvious later. .NET syntax uses “#” as an optional digit placeholder, “0” as a digit placeholder that displays 0 if there is no digit, “.” for a decimal separator, and “,” to use a thousands separator (unlike VFP, only one is needed in the format even when numbers exceed one million). The format string is surrounded with “{0:” and “}”, so the default of “{0:#,##0}” specifies no leading zeros, no decimal places, and thousands separators.

- nBand1End indicates where on the outer rim of the gauge the first color band appears, such as the red band in **Figure 1**. Only the end value is needed, since the band starts at 0. Similarly, nBand2End and nBand3End indicate the ending positions of the second and third color bands, with the starting positions being the end of the previous band.
- By default, nBand1End, nBand2End, and nBand3End are assumed to be percentages, so a value of 35 indicates an ending position of 35% of the gauge arc. If you want to use amounts instead, such as 25,000, set lValuesAsPercentages to .F.
- Since the gauge can't go above 100%, the needle is pegged at the maximum value when nValue is greater than nMaxValue. Because you may want the maximum value to be a goal that could be exceeded (for example, a salesperson's monthly quota may be \$10,000 but they certainly could sell more than that), you can set nGoalPosition to a lower value than 100. For example, if you set it to 75, then the nMaxValue value appears at 75%.
- For larger numbers (over 1,000), the labels can overlap the major tick marks, so increase the value of nLabelDistance accordingly. Alternatively, you can set nLabelFactor to a value to divide the labels by so smaller numbers are shown. For example, if you set nLabelFactor to 10000, the 5,000 position on the gauge appears as "5."

Creating a dashboard

Dashboards are all the rage these days. A dashboard is a form displaying multiple panels of information, such as charts, reports, and of course gauges. Another sample form that comes with the samples for this document, Dashboard.scx (**Figure 3**), is a simple demo of how a dashboard might work.

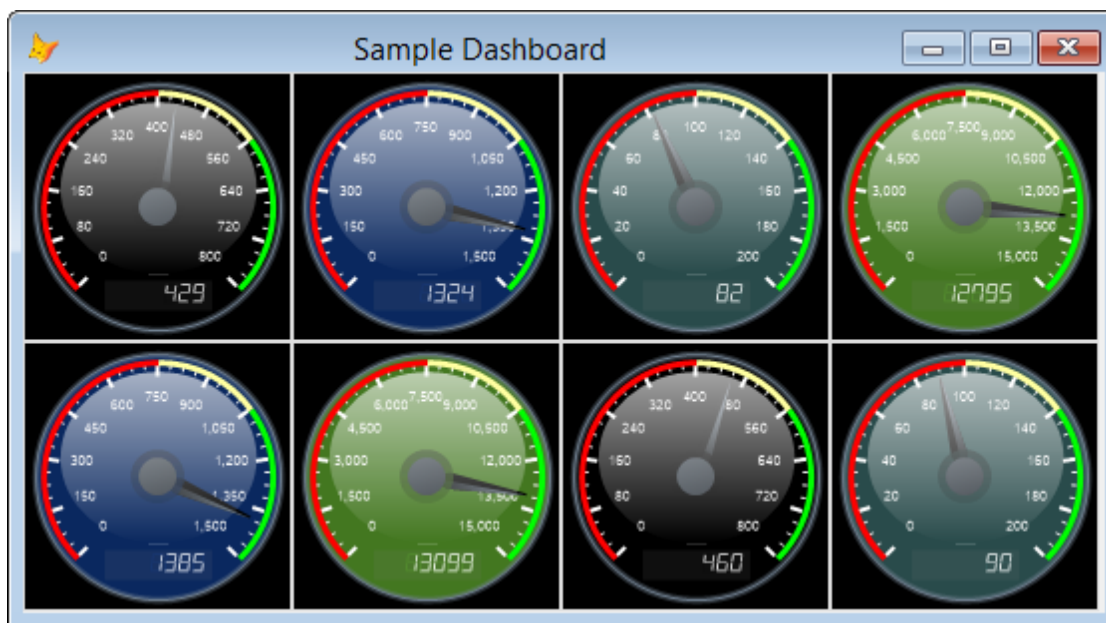


Figure 3. A dashboard consists of multiple gauges.

This form is actually quite simple. Its `Init` method adds eight `Image` controls and sizes and positions them so they take up two rows of four images. A `Timer` on the form calls the form's `DrawGauges` method, which uses a single `Gauge` control to do the drawing. `DrawGauges` sets the properties of the `Gauge` control to different values for each gauge (different dial colors and different current and maximum values), draws the gauge, and sets the `PictureVal` property of each `Image` control to the resulting image. Just for fun, the timer fires every 2 seconds and shows a random value so you can see the needles move.

How Gauge works

The `Gauge` class is actually a wrapper for a .NET DLL that does all the work. I'll discuss the .NET class later.

To avoid COM registration and other issues, I use Rick Strahl's `wwDotNetBridge` utility (<http://tinyurl.com/ce9trsm>). As you can see in **Listing 1**, the `Init` method of `Gauge` instantiates `wwDotNetBridge` into the `oBridge` property. Since you usually only want a single instance of `wwDotNetBridge` in an application, you can pass an existing instance to `Init` instead. `Init` also loads the `Gauge.dll` .NET assembly and instantiates the `Gauge.GaugeControl` class into the `oGauge` property.

Listing 1. The `Init` method sets up the helper objects needed by the class.

```
lparameters toBridge

* If we were passed a wwDotNetBridge object, use it. Otherwise, create one.

if vartype(toBridge) = '0'
    This.oBridge = toBridge
else
    This.oBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V2')
endif vartype(toBridge) = '0'
loBridge = This.oBridge

* Load the Gauge assembly: it must be in the current directory or path.

if not loBridge.LoadAssembly('Gauge.dll')
    This.cErrorMessage = 'Gauge.dll could not be loaded: ' + ;
    loBridge.cErrorMsg
    return
endif not loBridge.LoadAssembly('Gauge.dll')

* Instantiate a GaugeControl object.

This.oGauge = loBridge.CreateInstance('Gauge.GaugeControl')

* Set cFormat to a default we can't set in the Properties window.

This.cFormat = '{0:#,##0}'
```

Since the .NET DLL does all the work, all the `DrawGauge` method of the `Gauge` class has to do is populate the properties of the .NET object with the values of its own properties, call

the .NET object's DrawGauge method, and put the return value, which is the bytes of the gauge image, into clmage. The code for DrawGauge is shown in **Listing 2**.

Listing 2. The DrawGauge method uses the GaugeControl object to draw the gauge.

```
local lnEnd1, ;
    lnEnd2, ;
    lnMaxValue
with This

* Get the band positions.

    lnEnd1    = .nBand1End
    lnEnd2    = .nBand2End
    lnMaxValue = .nMaxValue/100

* If the band values were entered as amounts, convert to percentages.

do case
    case .lValuesAsPercentages
    case .nMaxValue = 100
        && max value hasn't been set
        lnEnd1 = 35
        lnEnd2 = 70
    otherwise
        lnEnd1 = int(lnEnd1/lnMaxValue)
        lnEnd2 = int(lnEnd2/lnMaxValue)
endcase
endwith
with This.oGauge

* Set the appearance properties.

.AdjustLabelSize    = This.lAdjustLabelSize
.BackColor           = This.GetColor(This.nBackColor, ;
    This.nBackColorAlpha)
.BackColor2         = This.GetColor(This.nBackColor2, ;
    This.nBackColorAlpha)
.BackGradientMode   = This.nBackGradientMode
.Band1Color         = This.GetColor(This.nBand1Color)
.Band1End           = lnEnd1
.Band2Color         = This.GetColor(This.nBand2Color)
.Band2End           = lnEnd2
.Band3Color         = This.GetColor(This.nBand3Color)
.DialColor          = This.GetColor(This.nDialColor, ;
    This.nDialAlpha)
.DialText           = This.cDialText
.DialTextColor      = This.GetColor(This.nDialTextColor)
.DialTextFontName   = This.cDialTextFontName
.DialTextFontSize   = This.nDialTextFontSize
.DialTextFontBold   = This.lDialTextFontBold
.DialTextFontItalic = This.lDialTextFontItalic
.DigitsColor        = This.GetColor(This.nDigitsColor)
.DisplayDigitalValue = This.lDisplayDigitalValue
```



```
.LabelFontBold      = This.lLabelFontBold
.LabelFontItalic    = This.lLabelFontItalic
.LabelFontName      = This.cLabelFontName
.LabelFontSize      = This.nLabelFontSize
.Format             = This.cFormat
.Glossiness         = This.nGlossiness
.Height            = This.nSize
.LabelColor         = This.GetColor(This.nLabelColor)
.LabelDistance     = This.nLabelDistance
.LabelFactor        = This.nLabelFactor
.MajorTickColor    = This.GetColor(This.nMajorTickColor)
.MajorTicks        = This.nMajorTickCount
.MinorTickColor    = This.GetColor(This.nMinorTickColor)
.MinorTicks        = This.nMinorTickCount
```

* Set the value properties.

```
.MaxValue = This.nMaxValue * 100/This.nGoalPosition
.Value     = This.nValue
```

* Draw the image and set our cImage property to the image bytes.

```
This.cImage = .DrawGauge()
endwith
```

The only complication in DrawGauge is that VFP color values don't match up with .NET color values: the .NET values have the red, green, and blue components reversed, and also support an alpha, or transparency, value. So, DrawGauge calls a helper method named GetColor (**Listing 3**), which pulls out the color components and puts them into the order needed for .NET.

Listing 3. The GetColor method converts a VFP color number to the .NET equivalent.

```
lparameters tnColor, ;
    tnAlpha
local lnRed, ;
    lnGreen, ;
    lnBlue, ;
    lnAlpha
lnRed   = mod(tnColor, 256)
lnGreen = mod(bitrshift(tnColor, 8), 256)
lnBlue  = mod(bitrshift(tnColor, 16), 256)
lnAlpha = iif(vartype(tnAlpha) = 'N', tnAlpha, 255)
return rgb(lnBlue, lnGreen, lnRed) + bitlshift(lnAlpha, 24)
```

The .NET component

GaugeControl.cs, included with the samples for this document, is the source code for the .NET gauge component in Gauge.dll. I started with code created by Ambalavanar Thirugnanam, available from <http://tinyurl.com/ppl44uy>, and made a number of changes to it:

- I modified it to be a simple .NET class that returns an image as a string rather than a Windows Forms User Control that displays the gauge. This allows the image to be written to a file or displayed in a VFP Image control without having to worry about registering the .NET control as an ActiveX control and adding it to a VFP form.
- I added properties for various colors, such as the background color, rather than using hard-coded values.
- I added support for a background gradient in addition to a solid color.
- Because it's difficult to create a .NET Font object in VFP, even with wwDotNetBridge, I added properties for the name, size, bold, and italics settings of fonts used for the dial text and labels. GaugeControl uses these properties to instantiate a Font object with the specified settings.

If you're interested in how the .NET component works, I recommend reading the article at <http://tinyurl.com/ppl44uy> as it discusses the logic and math involved in drawing the gauge. Then examine the C# source code in GaugeControl.cs to see how it's implemented.

If you build the Gauge solution that includes GaugeControl.cs, you'll find that it has a post-build event that copies the DLL to the parent folder of the solution, which is the same folder as Gauge.vcx is located. Note that if you've used the VFP Gauge control and VFP is still open, you have to close VFP before building the .NET solution because the .NET DLL is still open in VFP.

Since GaugeControl.cs uses GDI+ to do all of the drawing, why didn't I convert the C# code to VFP code using the VFPX GDIPlusX project? After all, that would give us a 100% VFP solution with no need for wwDotNetBridge or Gauge.dll. The reason I didn't is two-fold:

- Why reinvent the wheel? It would've taken several hours to convert the C# code into the equivalent VFP code and there'd be lots of debugging to make sure it works the same.
- I've run into some performance issues with GDIPlusX on some machines. In fact, this is what prompted me to look at this solution in the first place. I was using the VFPX FoxCharts project to draw gauges but found that on some systems, it was taking a minute or more to draw the gauge. In tracking the problem down, I found that on those systems, some of the GDI+ function calls were taking an order of magnitude longer to execute, and these functions were called thousands of times for each gauge. I don't know why some systems have this performance problem with GDIPlusX but the .NET component has no such problems on those systems.

Deploying Gauge

Deploying Gauge is straightforward:

- Add Gauge.vcx and wwDotNetBridge.prg to your project.
- Use the Gauge class as you see fit: to create image files for reports (cImage is in PNG format) or for the source of images in forms.

- Include `wwDotNetBridge.dll`, `ClrHost.dll`, and `Gauge.dll` in your installer or copy those files to the client's system. No registration is required for any of these components.

`Gauge.dll` requires version 2.0 of the .NET framework. Windows Vista and later come with .NET 2.0 so this is only an issue for Windows XP and earlier. If you use Inno Setup as your application installer, you can make your installer detect whether .NET 2.0 is missing and automatically download and install it by adding `#INCLUDE DotNet2Install.iss` to your Inno script file. `DotNet2Install.iss` is included in the samples for this document, as is `Isxdll.dll`, a component used by `DotNet2Install.iss`.

Toolbar buttons

VFP developers often use a toolbar instead of adding buttons directly to a form because controls in a toolbar don't get focus. However, toolbars aren't the easiest controls to work with:

- The user can move and close a toolbar, which often leads to support calls like "how do I get the toolbar back." They can even undock the toolbar with an inadvertent double-click.
- You can't create a toolbar instance; you have to create a separate class even if only one instance of the toolbar is used and it's never subclassed.
- Toolbars are odd controls to work with at design time. When you drop a control on one, VFP adds it to the left edge of the toolbar regardless of where you actually dropped the control. Separators are weird to work with too; I always have to reposition them a couple of times to get them to the correct place.
- You can't visually add a toolbar to a form; doing so creates a formset. Programmatically adding one to a form is odd too: you have to do it someplace like `Activate` rather than `Init` or it won't attach to the form.
- Toolbars take up some of the form height but you can't see that at design time so you have to pretend the toolbar is there when sizing the form and placing controls.

Carlos Alloatti specializes in developing libraries VFP developers can use to make the user interface of their applications more attractive, modern-looking, and easier to use. His `TBZ` library, available for download from <http://www.ctl32.com/tbz/tbz.html>, provides controls you add to a VFP form rather than a toolbar. Because they act like normal VFP controls at design time and don't receive focus at runtime, you avoid the issues discussed above while getting the benefits of a toolbar.

One other benefit you get from `TBZ` is that in addition to command buttons (the `_TBCommand` class), checkboxes (`_TBCheck` and `_TBToggle` for a graphical version), and radio buttons (`_TBRadio`), Carlos provides drop-down (`_TBDropDown` and `_TBDropDownVert`) and split button (`_TBSplitButton` and `_TBSplitButtonVert`) controls. These are buttons that include shortcut menus; in the case of a split button, you can either

click the button to take some action or the down arrow beside it to display the menu. See **Figure 4** for an example.

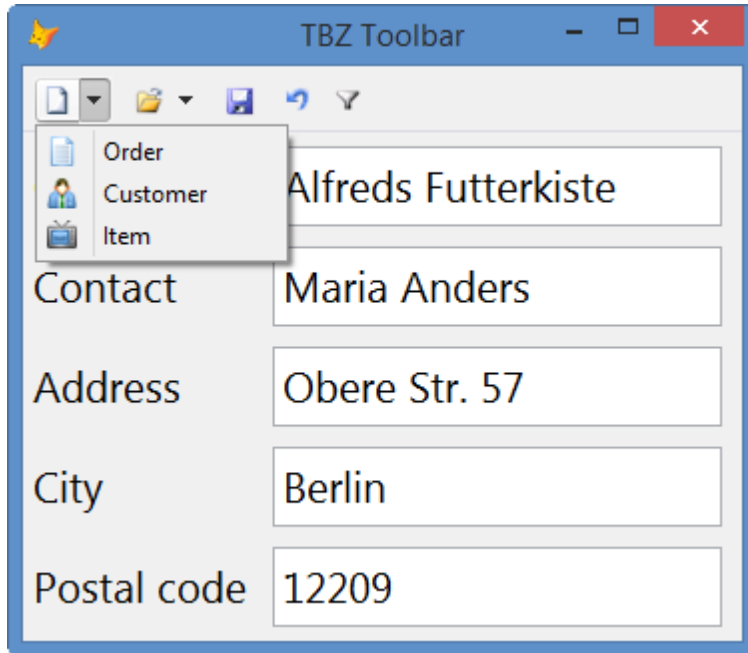


Figure 4. The TBZ split button provides both a command button and a shortcut menu.

Using TBZ

Start by adding `_TBZ.vcx` to your project. When you build the project, 74 PRGs, all starting with “_api,” are automatically added. These PRGs use a rather ingenious technique: each declares a Windows API function with an alias the same as the PRG name and then calls that function. The benefit of this approach is that there’s no need to declare functions before they’re used. For example, when the Windows API `CreateFont` function is needed, call `_APICreateFont`. The first time it’s called, the PRG is executed, which declares `CreateFont` as `_APICreateFont` and then executes it. The second time `_APICreateFont` is used, the Windows API function is called instead of the PRG because of the order in which VFP looks for names.

In addition to the PRGs, eight PNGs, `_TBZ01.png` through `_TBZ08.png`, are automatically added to the project. These PNGs contain images used for the various styles of the controls.

To create a toolbar in a form, drop instances of TBZ classes onto the form. For example, to create the sample `TBZToolbar.scx` included with the samples for this document, I dropped two `_TBSSplitButton`, two `_TBCommand`, and a `_TBToggle` onto the form and positioned them accordingly. I also dropped a `_TBLine` on the form to act as a separator line for the toolbar. The interesting thing about `_TBLine` is that it sizes itself automatically to the width of the form so you don’t have to worry about sizing it, setting `Anchor` to handle resizing, etc.

After adding the controls to the form, add code to the `Click` method of each. For `_TBDropDown` and `_TBSSplitButton`, add code to the `DropDownClick` method to define and

display a shortcut menu and take the necessary action. **Listing 4** shows the code that displays the menu shown in Figure 4. In a real application, the MESSAGEBOX statement would be replaced with the appropriate code.

Listing 4. This code displays a shortcut menu for the first split button in the form.

```
local lnResult
This.tbMenuItemAdd(1, 0, 0, 0, 0, '&Order', 'invoice.png')
This.tbMenuItemAdd(2, 0, 0, 0, 0, '&Customer', 'customer.png')
This.tbMenuItemAdd(3, 0, 0, 0, 0, '&Item', 'item.png')
lnResult = This.tbMenuShow()
do case
  case lnResult = 1
    messagebox('Create new order')
  case lnResult = 2
    messagebox('Create new customer')
  case lnResult = 3
    messagebox('Create new item')
endcase
```

Add items to the shortcut menu by calling the `tbMenuItemAdd` method of the button and display the menu by calling `tbMenuShow`. The parameters for `tbMenuItemAdd` are:

- The ID of the item. This is the value returned by `tbMenuShow` when the user selects an item.
- The parent ID of the item. To create a submenu, specify the ID of the item this item should be under. **Figure 5** shows an example of a submenu.
- The type of menu item. You can use constants defined in `_TBZ.h` for the values. The choices are 0 for a normal item, 1 for a separator, 2 to display a button as a checkmark, and 3 if the item contains a submenu.
- 1 if the item is checked (or displays a button if the third parameter is 2) or 0 if not.
- 0 if the item is enabled or 1 if not.
- The caption of the item. Use `&` to specify that the next character is the hot key for the item.
- An image file to use as a picture for the item.

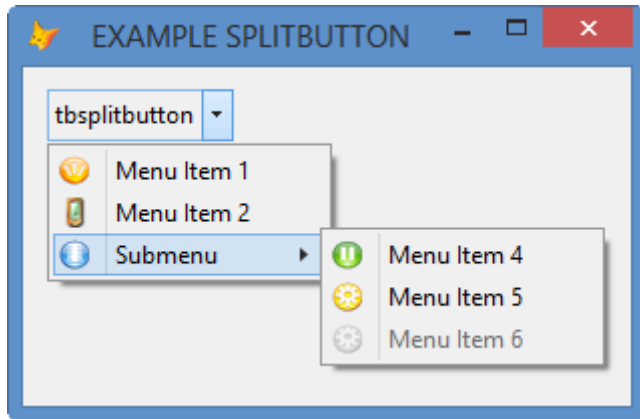


Figure 5. Submenus are easy to create using `tbMenuItemAdd`.

You may also want to set properties of the controls. There are a lot of properties that control the appearance of the controls, all documented on the TBZ web page, but the main ones are listed in **Table 2**.

Table 2. The most common properties of the TBZ controls.

Property	Description
Picture	The image to use for the button.
Caption	The caption for the button.
PicturePosition	Determines where the picture goes; only applicable when Caption isn't blank.
Alignment	Specifies the alignment of the caption.
ToolTipText	The tooltip to display for the control. TBZ uses a custom tooltip window rather than the VFP window.
tbAlign	Controls how automatic positioning works: 0 means no automatic positioning, 1 (the default) positions controls starting at the left edge of the form, and 2 positions them starting at the right edge.
tbGroupID	Radio controls with the same value act like a group.
tbMarginH	The space between the left or right edge of the form and the first control when <code>tbAlign</code> is 1 or 2.
tbMarginV	The space between the top or bottom of the form and the controls when <code>tbAlign</code> is 1 or 2.
tbStyle	Specifies which of the eight PNG files provides the visual style for the control; see the TBZ web page for screen shots of the various styles.
tbValue	The value of the radio button or checkbox; use this instead of <code>Value</code> .

Although it's just a sample form and doesn't do much, `TBZToolbar.scx` shows one use for split buttons: allowing the user to select one of several choices. For example, clicking `New` (the leftmost button) would add a new customer but clicking the drop-down part of the button allows the user to select what to create a new record of: `Order`, `Customer`, or `Item`. Similarly, clicking `Open` (the second button in the toolbar) would display a dialog of orders for the current customer but clicking the drop-down displays the most recent orders for the customer.

TBZ issues

There are a couple of things I couldn't get to work at first. One was that neither drop-down nor split buttons would display a shortcut menu. The cause turned out to be a case-sensitivity issue in the `_GetMouseStatus` method of `_TB`, the parent class for the TBZ controls. Replace the commented-out line in that method with the one following it:

```
*** If m.laMouse[1].Name = 'UISHAPER' Then
    If upper(m.laMouse[1].Name) = 'UISHAPER'
```

The second wasn't really a problem, just a difference in the way the controls worked from the documentation. The TBZ web page states that the controls are automatically laid out at runtime, similar to how a VFP Toolbar works. However, I found I had to manually position the controls where I wanted them to appear because they weren't automatically positioned. The cause turned out to be simple: the TBZ controls use `BINDEVENT` to bind their `_ArrangeControls` method, which does the positioning, to the `Init` method of the form, and my sample form didn't have code in `Init`, which caused the event binding to fail. Adding a comment to `Init` took care of that issue.

How it works

Carlos has a good description of how TBZ works on the TBZ web page. He includes information on how to create your own style PNG files if you want to use a different theme as well as GDI+ drawing issues he ran into and how he solved them.

Tabbed document interface

Some users live in their Internet browsers. They're used to a web page just being another tab within their browser window. Carlos' TDI library, available for download from <http://www.ctl32.com/tdi/tdi.html>, allows you to create a similar user interface for your application: each open form is just another tab within a main window. **Figure 6** shows an example, with three forms open as tabs.

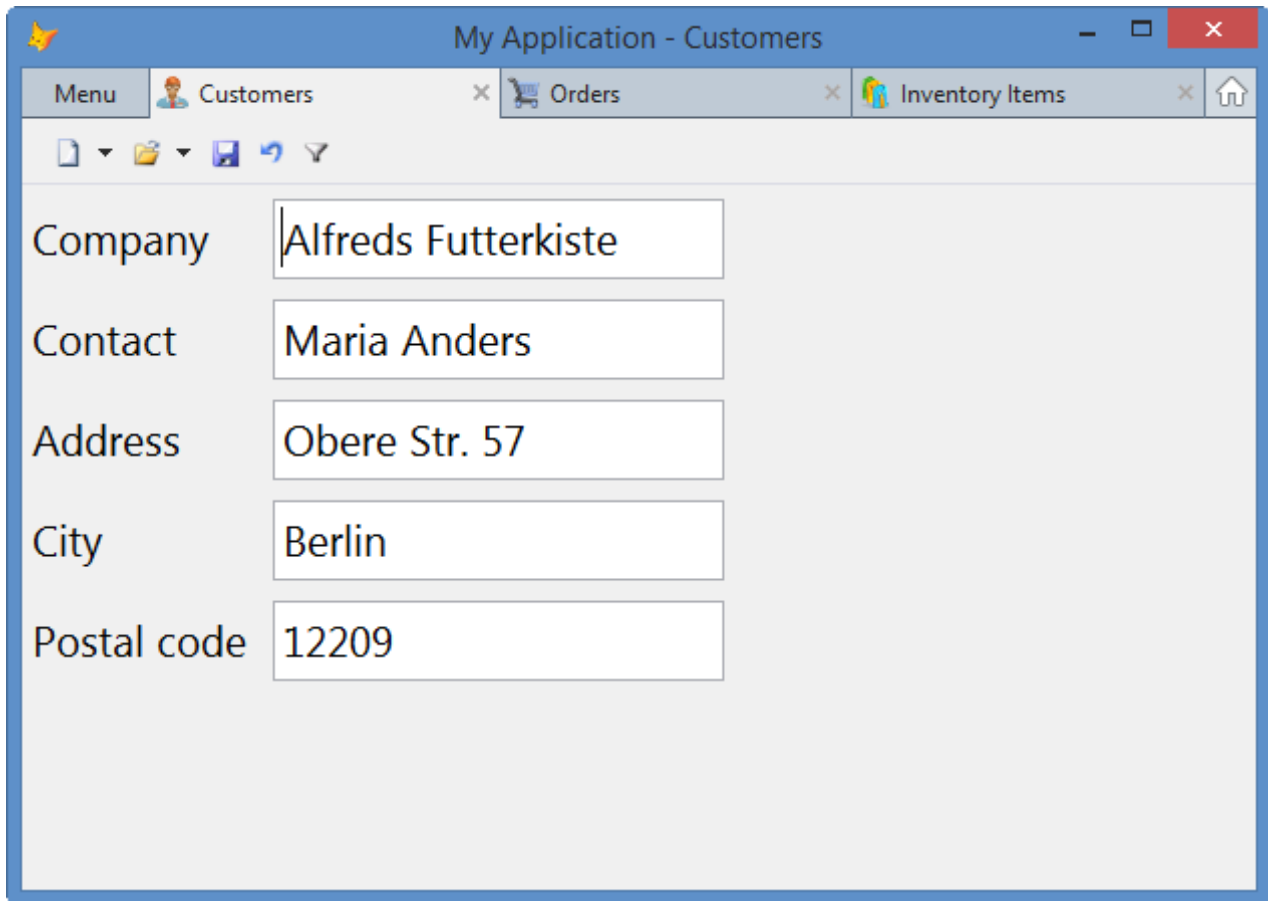


Figure 6. The TDI library makes it easy to create a tabbed document interface.

Typically, the first tab in such an interface is a menu of available forms or some type of home page for the application, such as that shown in **Figure 7**. When the user opens a form, it appears as a new tab. The tab may display an icon and includes an “X” to close the tab. The home button at the right edge of the main window selects the first tab. As more tabs are opened, the tab width automatically narrows as necessary to fit all of the open tabs in the window. Tabs can be rearranged by dragging them to the desired tab position. The first tab is special in that it doesn't display an icon and can't be closed or rearranged.

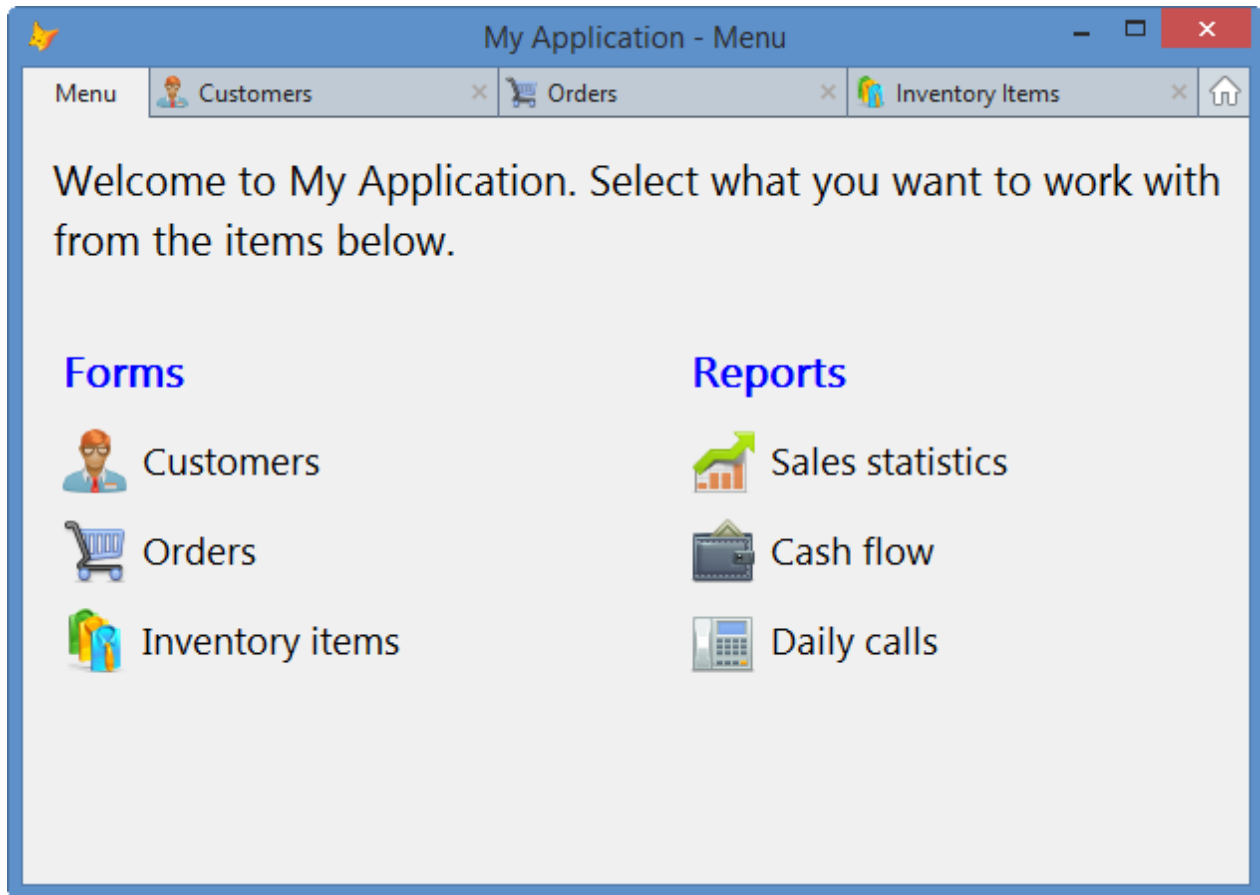


Figure 7. The first tab is usually a menu or home page.

Using TDI

Start by adding TDI.vcx to your project. When you build your project, 29 PRGs, all starting with “_api” and using the same technique the PRGs used by TBZ do, are automatically added. Only 10 of the PRGs are the same as those in TBZ so if you use both libraries in an application, you’ll have 93 of these PRGs in your project.

The next step is to create the main window for the application. This form hosts the forms in your application rather than _SCREEN so it’ll be a top-level form. To hide _SCREEN, create a CONFIG.FPW file containing at least SCREEN=OFF (you’ll likely want other lines as well, such as RESOURCE=OFF) and add it to the Text Files section of your project. Create a form and set ShowWindow to 2-As Top-Level form. Drop a TDIMain object on the form and add code to its TDIInit method to run another form, the one that’ll display the content for the first tab. For example, Main.scx (shown in **Figure 8**), the form that acts as the main window in the samples for this article, just has one line of code in TDIInit:

```
do form Menu
```

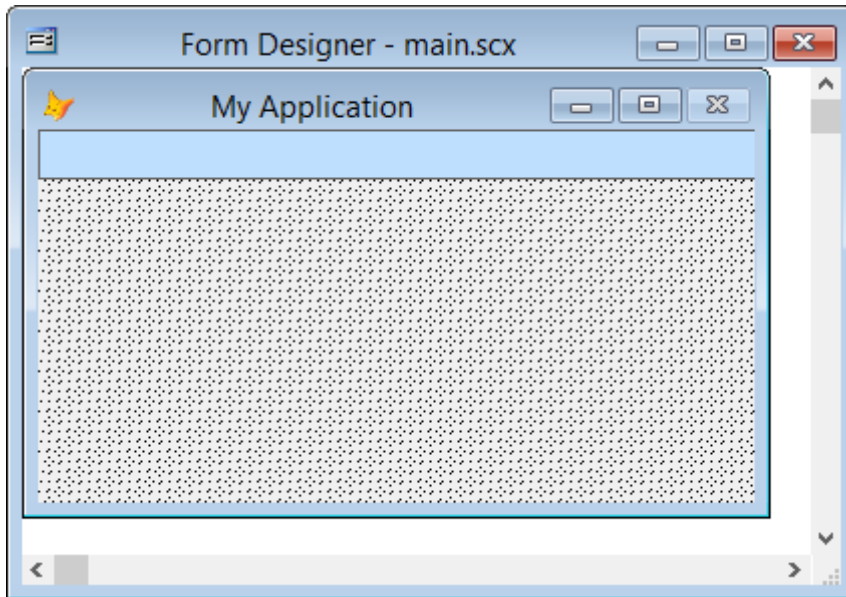


Figure 8. The form for the main window just consists of an instance of TDIMain.

There are only a few properties of TDIMain you may wish to change:

- **TabFirstWidth:** the width of the first tab. If you leave this at the default of 0, the tab is automatically sized.
- **TabMaxWidth:** the maximum width of a tab. The default is 200.
- **TDICenterDialogs:** set this to `.T.` (the default) to center system dialog windows, such as `MESSAGEBOX()`, on the form or `.F.` to center them on the Windows desktop.
- **TDIGoToFirstTabOnClose:** set this to `.T.` to automatically select the first tab when a tab is closed or `.F.` (the default) to select the next logical tab.

Now create a form that contains the contents of the first tab. Set `ShowWindow` to 1-In Top-Level Form and `Caption` to the text displayed on the first tab. Add whatever controls to the form you wish. In the case of `Menu.scx` (shown in **Figure 9**), which is the form used for the first tab in the samples for this article, I added some labels and some instances of a custom class consisting of an image and label to act as big buttons (the “Customers,” “Orders,” and other buttons shown in Figure 9). The `Click` method of each “button” runs another form using `DO FORM SomeFormName` or displays a message box.

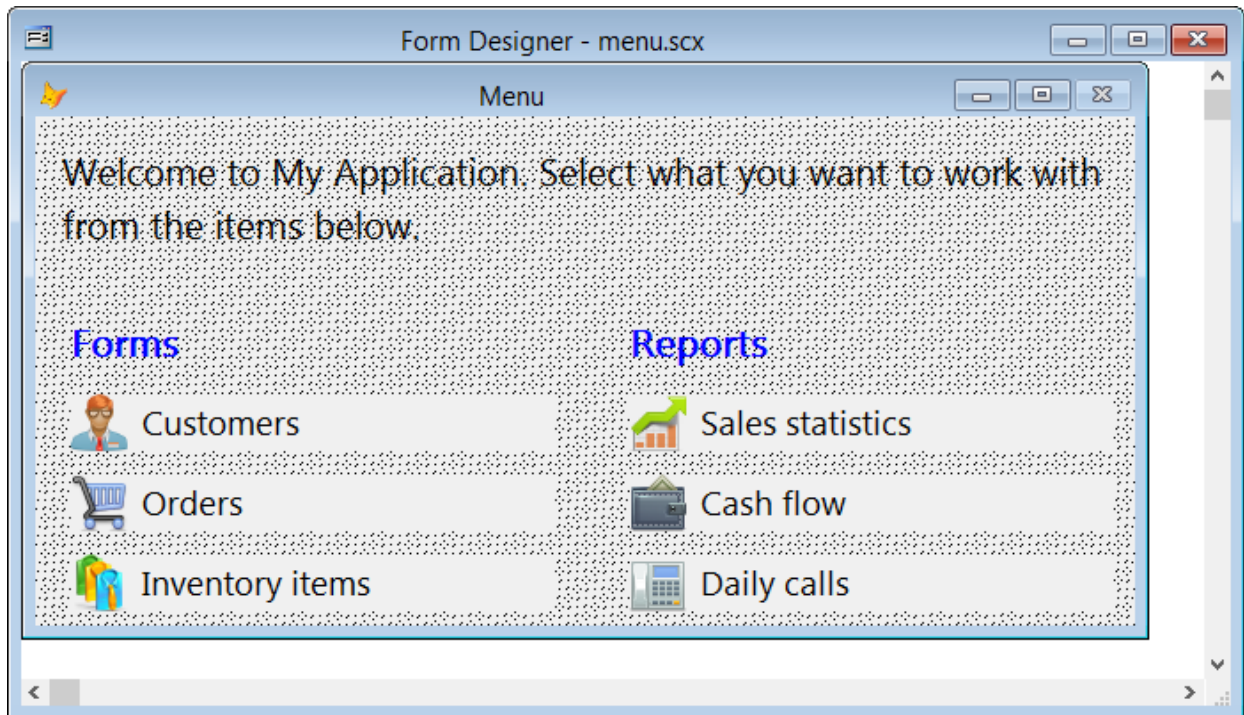


Figure 9. The form for the main tab usually contains buttons to launch other forms.

Finally, create the other forms used by the application, being sure to set `ShowWindow` to 1-In Top-Level Form, `Caption` to the text displayed on the tab, and `Icon` to the desired icon for the tab. One thing you'll find is that you might want to move the controls a little to the right and down from the top from where you might normally put them. For example, I usually start controls 10 pixels from the left and 10 from the top but that puts them too close to the edges when used with TDI. I suggest starting them at least 15 pixels from the left and top edges.

To display the main window for the application, use `DO FORM MainWindowName` or instantiate and call the `Show` method of the main window class you created.

How it works

Carlos has a brief description of how TDI does its magic on the TDI web page. Basically, one of his classes monitors VFP window creation and takes over how the form is displayed inside the top-level form.

Themed title bar

Forms that just have the usual grey appearance can be a little boring. Even worse, in Windows 10, by default all windows have a white title bar. **Figure 10** shows a typical VFP form with `Desktop` set to `.T.`

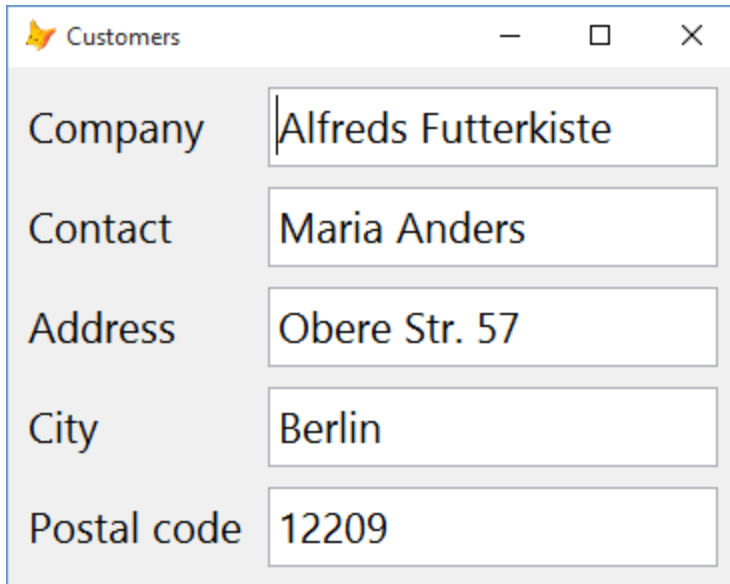


Figure 10. Desktop windows in Windows 10 look even more boring with a white title bar.

If you want to add a little color and pizzazz to your forms, use Markus Winhard's ThemedTitleBar VFPX project (<http://tinyurl.com/o6lvong>). **Figure 11** shows an example of the same form shown in Figure 10 but with a more attractive title bar thanks to Markus.

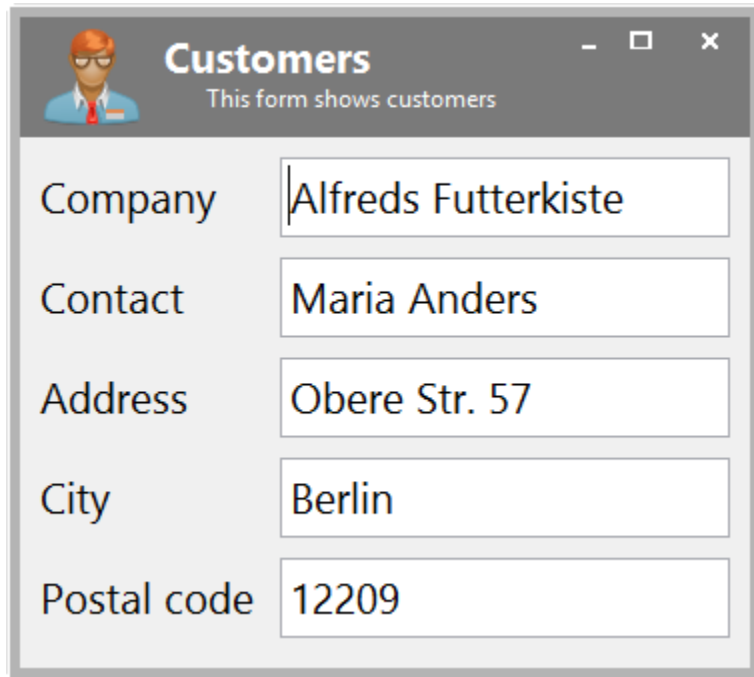


Figure 11. ThemedTitleBar allows you to add an attractive element to your forms.

ThemedTitleBar has the following appearance and behavior:

- Dragging the title bar moves the form.

- Double-clicking anywhere in the title bar toggles between maximizing the form and restoring it to its original size.
- The minimize, maximize, and close buttons work as expected. They respect the `MinButton`, `MaxButton`, and `Closable` properties of the form.
- The large label in the control is the title of the form. The smaller label below it is an optional description. The control can also optionally display an image.

`ThemedTitleBar` consists of two classes: `ThemedTitleBarBase` and `ThemedTitleBarSettingsBase`, both in `ThemedTitleBarBase.vcx`. Neither of these classes can be instantiated—you get an error if you try—so you'll actually use subclasses instead. Markus provides a set—`ThemedTitleBar` and `ThemedTitleBarSettings`, both in `ThemedTitleBar.vcx`—but you can create your own subclasses as we'll see later.

To use `ThemedTitleBar`, add both `ThemedTitleBarBase.vcx` and the class library containing the classes you'll use to your project, then drop an instance of the `ThemedTitleBarBase` subclass on a form. If you do nothing else, your form looks like Figure 11 but without the image or description. To display an image and/or description, add `ThemedTitleBarImage` and `ThemedTitleBarDescription` properties to the form and set them to the file name for the image and the text for the description, respectively. You can, of course, add these properties to your form base class. You can also use different names for these properties if you wish; in that case, change the values of the `cDescriptionProperty` and `cImageProperty` properties of the `ThemedTitleBarBase` subclass.

Other properties you may wish to set are:

- `IAutoMoveFormControls`: `ThemedTitleBarBase` automatically moves itself to the top of the form at runtime. Since you may have dropped the control somewhere other its final location, the other controls on the form have to move down to accommodate it. `IAutoMoveFormControls` determines whether that happens or it. Its default setting is `.T.`, meaning the controls are moved. However, if you place the `ThemedTitleBarBase` subclass at the top of the form and adjust the other controls yourself (which I prefer, so the form looks the same at design time and runtime), set this property to `.F.`
- `cSettingsClass` and `cSettingsClassLib`: since you have to use a subclass of `ThemedTitleBarSettings`, set these properties to the name of the subclass and the VCX that contains it.
- `cTitleProperty`: by default, the title label comes from the form's `Caption` property. If you want to use another property instead, set `cTitleProperty` to the name of that property.
- `IFixedHeight`: if you use a resizer control on your form and don't want it to change the `ThemedTitleBar`'s height, set `IFixedHeight` to `.T.`

`ThemedTitleBarSettings` has four properties you may want to set:

- `cCaptionFontFamily`: this is a comma-delimited list of fonts to use for the caption. The first one that exists is used. The default is "Segoe UI, Tahoma, Arial."
- `cDescriptionFontFamily`: like `cCaptionFontFamily`, but determines the font for the description.
- `nTitleBarBackColor`: the background color of the title bar.
- `nTitleBarForeColor`: the foreground color of the caption and description labels.

The `Init` method of `ThemedTitleBarBase` gets an instance of the `ThemedTitleBarSettingsBase` subclass to use from the following locations:

- If a property of `_SCREEN` named `ThemedTitleBarSettings` exists and contains an object, it's used.
- If the form uses the default data session, it instantiates the class specified in `cSettingsClass` and `cSettingsClassLib` and adds it to `_SCREEN` as a member named `ThemedTitleBarSettings`. Thus, after the first form is run, any other forms using `ThemedTitleBar` use the existing `_SCREEN.ThemedTitleBarSettings` instance.
- If the form has a private data session, it instantiates the class specified in `cSettingsClass` and `cSettingsClassLib` but doesn't put it anywhere; it's only used in `Init`.

The first and second points above bring up something interesting: since all forms using `ThemedTitleBar` use the same instance of `ThemedTitleBarSettings`, they all display the same colors; hence the name "themed." However, one thing I found was that changing `nTitleBarBackColor` of the `ThemedTitleBarSettings` object didn't change the title bar color of any forms that were already open. So, I created a subclass of `ThemedTitleBarBase` named `SFThemedTitleBar` (in `SFThemedTitleBar.vcx`) with the following code in `Init`:

```
dodefault()  
if type('_screen.ThemedTitleBarSettings') = '0'  
    bindevent(_screen.ThemedTitleBarSettings, 'nTitleBarBackColor', ;  
        This, 'RefreshColors', 1)  
    bindevent(_screen.ThemedTitleBarSettings, 'nTitleBarForeColor', ;  
        This, 'RefreshColors', 1)  
endif type('_screen.ThemedTitleBarSettings') = '0'
```

This code binds the `RefreshColors` method of this subclass to the `nTitleBarBackColor` and `nTitleBarForeColor` properties of the `ThemedTitleBarSettings` object, so changes to those properties cause this code to execute:

```
with _screen.ThemedTitleBarSettings  
    This.BackColor = .nTitleBarBackColor  
    This.lblCaption.ForeColor = .nTitleBarForeColor  
    This.edtDescription.ForeColor = .nTitleBarForeColor  
endwith
```

I also created a subclass of `ThemedTitleBarSettingsBase` named `SFThemedTitleBarSettings` and set the `cSettingsClass` and `cSettingsClassLib` properties of `SFThemedTitleBar` to use that class.

Now, all I have to do is change the values of the `nTitleBarBackColor` and `nTitleBarForeColor` properties of `_SCREEN.ThemedTitleBarSettings`, such as with a “themes settings” dialog, and any open forms change automatically.

There's only one glitch with `ThemedTitleBar`: if the form's `Desktop` property is `.T.` (which I use for all my forms), the form has a small white band at the top in Windows 10 as you can see in **Figure 12**. This is true of any form that has `TitleBar` set to `0-Off`, not just forms using `ThemedTitleBar`, but it looks even odder when `ThemedTitleBar` is used. I'm sure there's some Windows API function you can call to turn this off but I haven't come across it yet.

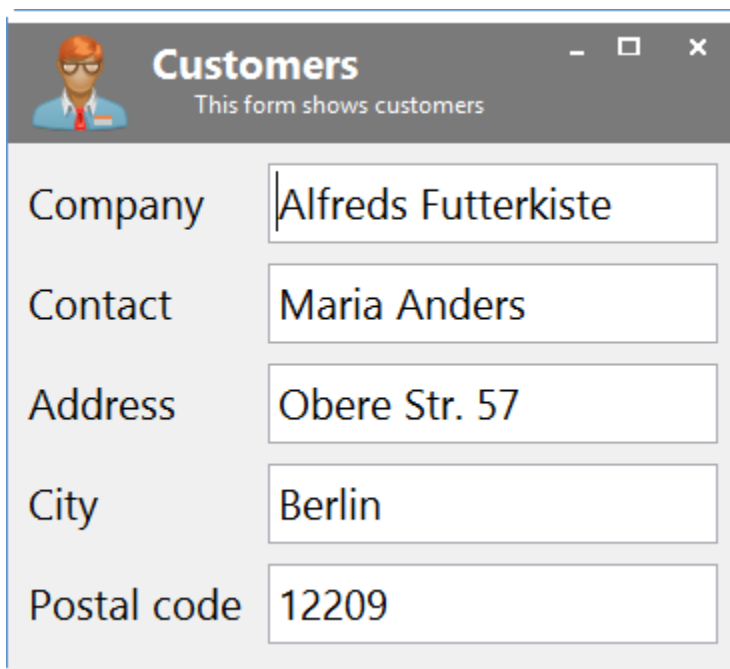


Figure 12. In Windows 10, forms with `Desktop .T.` have a white band at the top.

SSClasses

`SSClasses` is a VFPX project by Jun Tangunan that provides several controls with more colorful and interesting interfaces. You can download `SSClasses` from <https://vfp.codeplex.com/releases/view/99045>.

SSTextBox

`SSTextBox` is a container class (discussed on Jun's blog at <http://tinyurl.com/pdscl3b>) that at runtime looks like a textbox, but with some additional features, as you can see in **Figure 13**:

- The textbox has rounded corners. This is controlled through the Curvature property; set it to 0 for square corners.
- Setting focus to the control changes the border color, controlled with the BorderColor property.
- Double-clicking the textbox clears its value.
- The _Marker property determines whether a checkmark appears at the right edge of the control. Set it to 1 (the default) for a green checkbox, 2 for a blue one, and 0 for no checkbox.
- Set _NoBackspace to .T. if you want to prevent the backspace key from setting focus to the previous control (the default behavior of the VFP Textbox is one my customers have complained about).

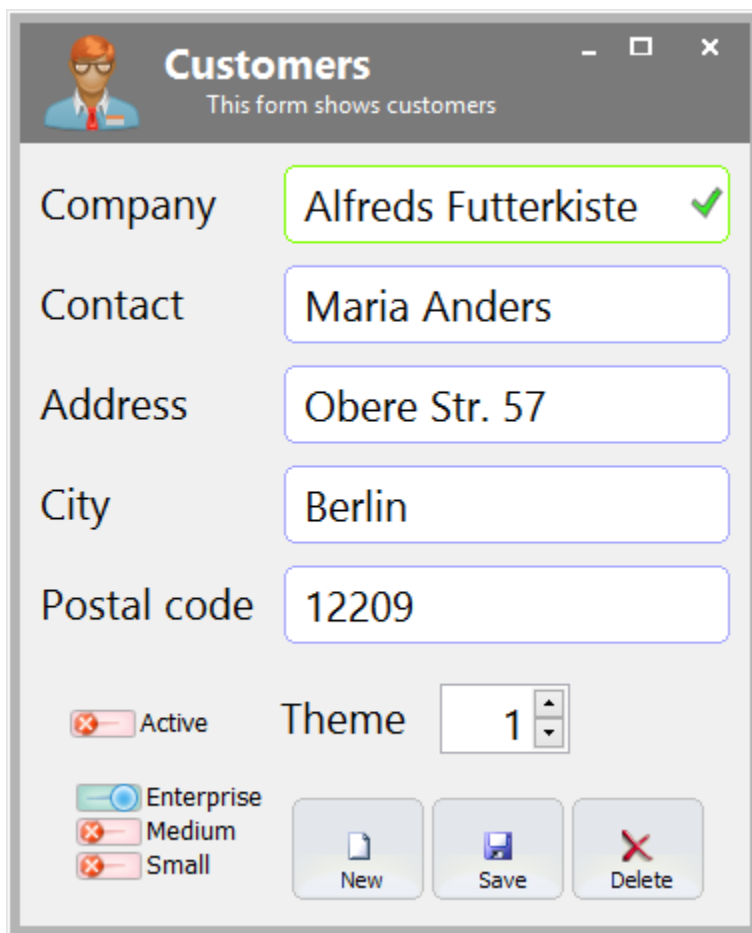




Figure 13. SSClasses provides several attractive, colorful controls.

I subclassed `SSTextBox` to create `SFSSTextBox` in `SFSSClasses.vcx`. It's a little bit easier to use; instead of setting the `ControlSource`, `FontName`, and `FontSize` properties of the textbox inside the container, you simply set those properties of the class itself. Also, it ensures the checkbox is vertically centered within the control and adds an `IClearOnDoubleClick` property that determines whether double clicking clears the value (the default is `.F.`).

SSSwitch

SSSwitch is a replacement for the VFP Checkbox that has a much more colorful appearance as you can see in Figure 13. It’s discussed on Jun’s blog at <http://tinyurl.com/nmbhgwo>. It has 18 different themes (see **Table 3**) that affect both the off and on state of the checkbox.

Table 3. The theme of SSSwitch affects its appearance.

Theme	Off	On
1	 Active	 Active
2	 Active	 Active
3	 Active	 Active
4	 Active	 Active
5	 Active	 Active
6	 Active	 Active
7	 Active	 Active
8	 Active	 Active
9	 Active	 Active
10	 Active	 Active
11	 Active	 Active
12	 Active	 Active
13	 Active	 Active
14	 Active	 Active
15	 Active	 Active
16	 Active	 Active
17	 Active	 Active
18	 Active	 Active

To set up SSSwitch, call the `_Settings` method from `Init`:

```
This._settings(Caption, Value, Theme number (1-18), .T. for no bold, ForeColor, ;  
    .T. for no border)
```

The value for the second parameter can be numeric or logical, depending on what type of value you want the control to contain. For example, pass `.T.` to use a logical value with it initially turned on. If you don't pass `.T.` for the fourth parameter, the caption of the checkbox is bolded when the value is `.T.` If you don't pass `.T.` for the last parameter, the control has a dashed line border when it has focus. Here's an example:

```
This._settings('Active', .T., 3, .T., , .T.)
```

It has a `ControlSource` property so you can bind it to any numeric or logical field or property. To get the current value of the control, use the `Value` property.

I subclassed `SSSwitch` to create `SFSSSwitch` in `SFSSClasses.vcx`. It's a little bit easier to use; instead of calling `_Settings`, set the `_Caption`, `Value`, `_nTheme`, `_nNoBold`, `ForeColor`, and `_Border` properties as desired. They each have an `Assign` method that calls a new `_Redraw` method that calls `_Settings`.

I also fixed a bug in `SSSwitch`: if you set `ControlSource` to an object property instead of a field in a cursor, you get an error when you change the value of the control. Here's the change I made in `SSSwitch.shpFocus.Click`:

```
*** DH 2015-09-11: handle control source being an object property  
*   If !Empty(This.Parent.ControlSource)  
*       Replace &lcField With .Value In &lcTable  
*   Endif  
*   do case  
*       case empty(This.Parent.ControlSource)  
*       case used(lcTable)  
*           replace &lcField with .Value in &lcTable  
*       otherwise  
*           store .Value to (This.Parent.ControlSource)  
*   endcase  
*** DH 2015-09-11: end of new code
```

SSOptSwitch

`SSOptSwitch` is a replacement for the `VFP OptionGroup` that uses the same 18 different themes as `SSSwitch`; see Figure 13.

To set up `SSOptSwitch`, call the `_Settings` method from `Init`:

```
This._settings(Value, Theme number (1-18), Captions, ForeColor, .T. for no border)
```

The value for the third parameter determines the number of buttons and what their captions are; specify the captions separated with “|” characters. If you don't pass `.T.` for the last parameter, the button with focus has a dashed line border. Here's an example:

```
This._settings(1, 1, 'Enterprise|Medium|Small')
```

Like SSSwitch, it has a ControlSource property so you can bind it to any numeric field or property. To get the current value of the control, use the Value property.

SSOptSwitch is a container class containing 12 buttons (only those that have the caption set are visible). If you want the buttons laid out horizontally instead of vertically, edit the instance and arrange the buttons as necessary.

SSButton, SSButton3_, and SSButton4

There are three classes in SSClasses that are replacements for the VFP CommandButton: SSButton, SSButton3_, and SSButton4 (no, I don’t know why there’s an underscore after SSButton3). All are more colorful than CommandButton, have built-in hover effects, and display their caption in bold when clicked. The differences between them are:

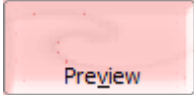
- SSButton supports themes which give it unusual color effects while SSButton3_ and SSButton4 just support highlight colors, although they also appear with a “glow” area.
- SSButton3_ appears as a grey button (although you can change the button color) while SSButton4 uses a variant of its hover color as its normal background color.
- SSButton has square corners while SSButton3_ and SSButton4 have rounded corners and a Curvature property that controls that appearance.


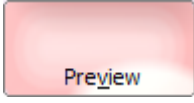

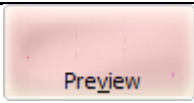

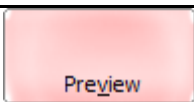

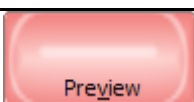
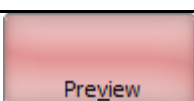
All three button types are initialized by calling _Settings in Init. For SSButton, it’s called like this:

```
This._settings(Caption, Theme number (1-10), Button color (1-6), Icon filename, ;
  Icon position (1-9), Tool tip, .T. to center caption, .T. to wordwrap caption, ;
  Font name, Font size, Font color, .T. for bold, .T. for italics, ;
  Special effect setting, Folder name for images)
```

The color and appearance of the button are affected by the second and third parameters. **Table 4** shows the ten themes that are available; the images were taken when the button color is 1 (red). For the button color parameter, use 1 for red, 2 for green, 3 for yellow, 4 for orange, 5 for grey, or 6 for blue. For the icon position parameter, use 1 for top left, 2 for top center, 3 for top right, 4 for middle left, 5 for middle center, 6 for middle right, 7 for bottom left, 8 for bottom center, or 9 for bottom right.

Table 4. The theme of SSButton affects its appearance.

Theme	Appearance
1	

2	
3	
4	
5	
6	
7	
8	
9	
10	

Regardless of the theme and button color, the hover color for the button is always a shade of blue, although the theme affects the exact shade used.

For `SSButton3_`, call `_Settings` like this:

```
This._settings(Caption, Highlight color (1-7), Icon filename, Icon position, ;  
    Tool tip, .T. to center caption, Special effect setting, Curvature, ;  
    .T. to wordwrap caption, BackColor, Font name, Font size, Font color, ;  
    .T. for bold, .T. for italics)
```

For the highlight color parameter, use 1 for red, 2 for green, 3 for yellow, 4 for orange, 5 for brown, 6 for blue, or 7 for “other” (a yellowish color). For the `BackColor` parameter, use 1 for `RGB(227, 230, 234)` (grey), 2 for `RGB(237, 231, 231)` (light grey), or 3 for `RGB(225, 250, 180)` (green).

SSTab is similar except it has no BackColor parameter:

```
This._settings(Caption, Color (1-7), Icon filename, Icon position, Tool tip, ;  
  .T. to center caption, Special effect setting, Curvature, ;  
  .T. to wordwrap caption, Font name, Font size, Font color, ;  
  .T. for bold, .T. for italics)
```

SSTab

SSTab provides colorful tabs for pageframes (**Figure 14**). This class has the following features:

- Because it's a separate control, you can place the tabs anywhere. In Figure 14, they are indented from the left edge of the pageframe.
- You can specify one of four colors for the tabs.
- The tabs can display an icon, a tooltip, and a folded corner for the active tab.

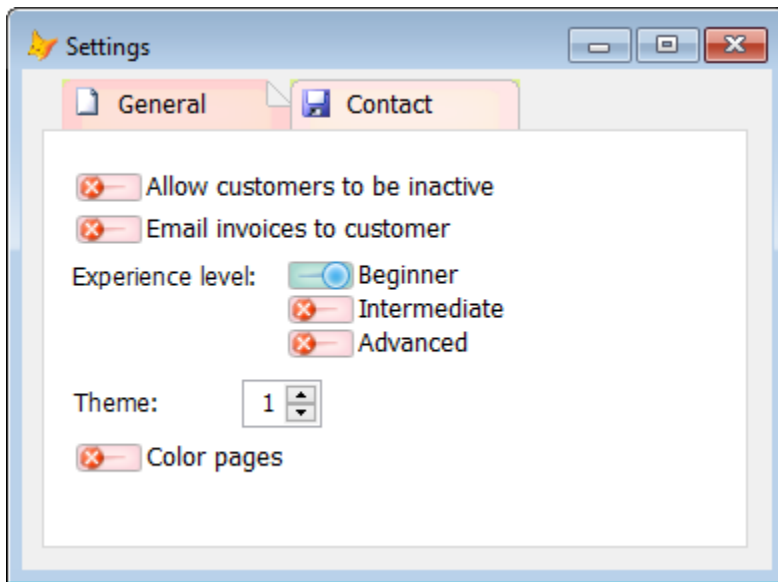


Figure 14. SSTab creates attractive tabs for pageframes.

To use SSTab, add a pageframe to a form and set Tabs to .F. so it doesn't display its own tabs. Add as many SSTab objects to the form as there are pages in the pageframe and in the Init method of each put code like this:

```
This._settings(Caption, Tool tip, Tab number, Name of pageframe, Color (1-4), ;  
  BackStyle, Icon filename, Icon stretch, .T. to show a fold, ;  
  .T. if this is the initially selected page, Inside color setting, ;  
  .T. to turn off PageFrame.Themes, Inside color to use)
```

If you pass 1 for the inside color setting, the pageframe is filled with a darker version of the color used for the tab. Use 2 for a lighter color or 0 for no color. Note that once you've specified a color, you can't change it back to uncolored; that could be fixed in

SSTab._Settings if desired. You can also specify a different color to use for the pageframe with the last parameter.

Jun notes that if a form contains more than one pageframe, you should put the pageframes and the associated SSTab objects into containers so the SSTab objects don't get confused about the effects.

SSEditBox

Figure 15 shows what SSEditBox provides: a replacement for the VFP EditBox that displays only a single line in the form but can drop down to a larger editing control. The advantage is that it doesn't take up much vertical space unless needed.

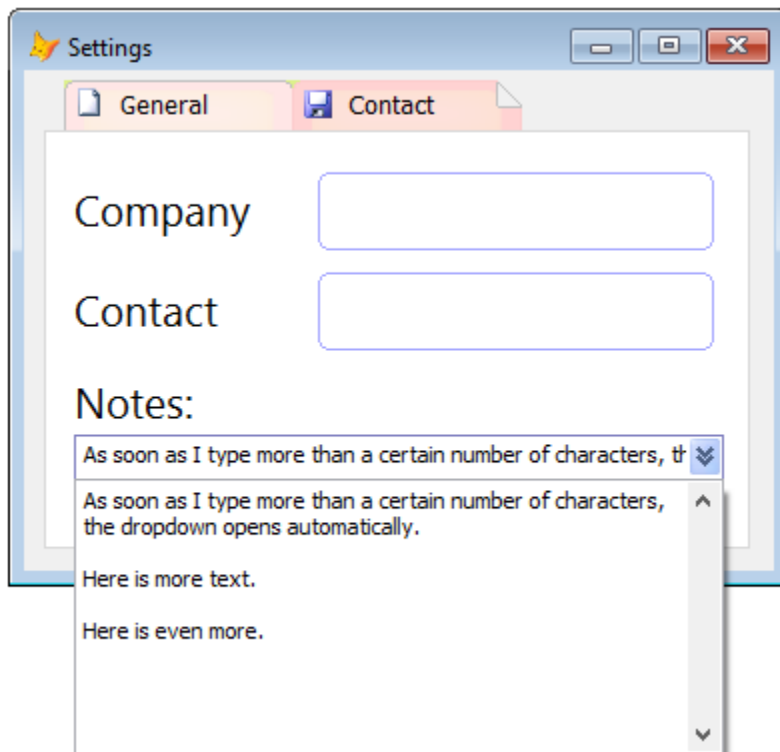


Figure 15. SSEditBox provides a dropdown edit window.

To use SSEditBox, drop it on a form and set ControlSource or handle the Value property as necessary. If you want the dropdown area (which is actually a form with no frame) to have a shadow, set _lShadow to .T. (the default). If you want it to be semi-transparent so you can see the controls it covers in behind, set _lTransparent to .T.

As you type in the control, the dropdown area automatically appears as soon as you've typed more characters than specified in _nMaxChars (the default is 15); set this property as desired. The dropdown also appears if you click the arrow at the right edge of the control or press the Down Arrow key. To close the dropdown, click the arrow again (see below), press Esc or Tab, or click outside the dropdown.

I fixed a couple of issues in SSEditBox. First, if you drop one on a page in a pageframe, you'll get an error in Init because Page doesn't have a Resize event. Here's the change I made in Init along with a comment about this:

```
*** DH 2015-09-18: this causes an error if the control is in a page of a
*** pageframe. Also, it's better to set Anchor than bind to Resize, so let's
*** just bind to our own Resize.
*BINDEVENT(this.Parent,"resize",this,"_resize")
BINDEVENT(this,"resize",this,"_resize")
```

I made a couple of changes in _dropedit. First, because it used CREATEOBJECT() to instantiate another class, that meant you'd get an error if you didn't SET CLASSLIB first. I changed CREATEOBJECT() to NEWOBJECT() so the class library is specified and SET CLASSLIB isn't needed. The second change is knowing when the edit form is open and doing nothing if so. Otherwise, clicking the arrow when the form is open, thinking that it'll close the form, causes the form to close and immediately reopen. Here's the entire method with the changes as indicated (I also added an lOpen property):

```
LPARAMETERS lRemoveLast
LOCAL lcValue
*** DH 2015-09-18: if the form is open, do nothing
if This.lOpen
    This.lOpen = .F.
    return
endif This.lOpen
*** DH 2015-09-18: end of new code

lcValue = ALLTRIM(this.txtedit.Value)
IF m.lRemoveLast
    this.txtedit.value = LEFT(m.lcValue,LEN(m.lcValue)-1)
ENDIF

*** DH 2015-09-01: changed to use NEWOBJECT so don't need SET CLASSLIB, and set
*** open flag
*This._editpop = Createobject("editdrop", Thisform,This.txtedit, m.lcValue)
This._editpop = newobject("editdrop", This.ClassLibrary, '', Thisform,This.txtedit, ;
    m.lcValue)
This.lOpen = .T.

If !IsNull(This._editpop)
    This._editpop._ShowForm(Objtclient(This, 2), Objtclient(This,1) + ;
        This.Height - 1, this._lshadow, this._ltransparent )
Endif
```

Summary

There's no excuse for creating a boring looking VFP application. Using the controls discussed in this document, and others available on VFPX or other sites, you can create a new, modern user interface for your forms that'll add years to the life of 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 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 *VFPX: Open Source Treasure for the VFP Developer, Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *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 and Southwest Xbase++ conferences (<http://www.swfox.net>). He is one of the administrators for the VFPX VFP community extensions Web site (<http://vfp.codeplex.com>). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (<http://tinyurl.com/ygnk73h>).

