

This paper was originally presented at the Southwest Fox conference in Gilbert, Arizona in October, 2013. <http://www.swfox.net>



"Modernizing" your VFP Apps

Rick Borup
Information Technology Associates, LLC
701 Devonshire Dr, Suite 127
Champaign, IL 61820
Phone: (217) 359-0918
Email: rborup@ita-software.com
Twitter: @rickborup

Is it Metro or is it retro? Regardless of how you feel about it, the Windows 8 "Modern" user interface introduced an entirely new paradigm, and I predict it's here to stay. Microsoft shelved the whole concept of the glossy 'aero' interface—with its rich graphics, alpha transparencies, and other eye candy—and replaced it with solid colors, flat icons, blocky tiles, and other major changes to the user experience. Like the ribbon interface introduced with Office 2007, these changes were at first reviled, but history shows they will soon be accepted by users as commonplace, becoming the de facto standard and making everything else look old-fashioned. What does this mean for our VFP apps? How can we redesign our apps' user interface to keep up with the times? Come to this session for some ideas on modernizing your legacy VFP apps to keep them looking fresh and up-to-date!

Table of Contents

Introduction	3
Evolution of the user interface	5
Character-based user interface.....	5
Graphical user interface.....	6
Skew-oh-what??	7
Design Concepts	10
Embrace space.....	10
Use color	10
Flat design	12
Modernizing your VFP app.....	14
Fonts.....	15
Forms.....	17
The original form.....	17
The first iteration	21
The second iteration.....	24
The third iteration.....	28
The fourth iteration	31
Other things to consider	35
Menus.....	38
Menu icons	38
Menu fonts.....	38
Main menu alternative	41
System dialogs.....	42
Summary	44
Resources	45
References.....	45
Software.....	45
Icon sources.....	45
Appendix A.....	47
Appendix B	49

Introduction

Let me begin by saying what this session is *not* about. It’s not about Windows 8, or at least not about Windows 8 specifically. It is not about Windows Phone apps or Windows Store apps or touch screen interfaces, although the design concepts found in all of those things are places where modern design principles have been applied. It’s also not about using an HTML interface to simulate Web design in Visual FoxPro – I’m talking about pure VFP solutions here.

What this session *is* about is the new, clean user interface design paradigm you see popping up all over these days – the “modern UI” – and how you can apply it to improve the appearance of your Visual FoxPro apps.

The guiding principles for modern user interface design are **do more with less** and **put content before chrome**.¹ The objective is to help the user focus on the primary content by removing extraneous, albeit attractive, elements from the user interface – in other words, to improve the user experience by reducing clutter.

*The guiding principles are “do more with less”
and “put content before chrome.”*

These design concepts can be seen not only in Windows 8 apps and Windows Store apps but also in a wide variety of web sites, HTML email, and desktop apps from many different vendors and sources. Microsoft, sometimes perceived as a follower when it comes to design, seems to be leading the way here. One example is the Microsoft web site itself, which has been wholly revamped and now employs a clean design with larger fonts and flat icons against a white background.

The MSDN site in particular got a great makeover a couple of years ago. Figure 1 shows what it looked like circa June of 2013. Note the clean, uncluttered appearance accomplished with an appropriate use of whitespace, muted colors on a white background, and flat yet colorful icons with text labels prominently featuring the major content areas. Sadly, many of these improvements were lost in the site’s July 2013 update.²

¹ See page 7 of the *Windows 8 User Experience Guidelines*. In this context, I believe “chrome” refers to flashy but extraneous design elements like the shiny chrome bumpers on a ’57 Caddy.

² To me, the site looks busier and less concisely organized after the July 2013 update. The links I consider to be the most important from a software developer’s point of view have been relegated to footnote status. Many developers feel the site’s content suffered, too – see visualstudiomagazine.com/articles/2013/08/07/devs-angry-over-msdn-redesign.aspx.

msdn


United States (English) | Sign in


[Home](#) [Library](#) [Learn](#) [Samples](#) [Downloads](#) [Support](#) [Community](#) [Forums](#)

Search MSDN with Bing



Microsoft Developer Network

**Visual Studio**
Amazing apps start with Visual Studio
Sign in to download the Visual Studio 2013 Preview



MSDN Subscription
An MSDN Subscription is the best way to get exactly what you need for your next challenging project.
[Buy an MSDN Subscription](#)
New Subscriber? [Get Started](#)
Existing Subscriber? [Access benefits](#)

Development platforms & tools

**VISUAL STUDIO**
No matter the size of your team, or the complexity of your project, Visual Studio can help turn your ideas into software.
[Download trial](#)
[Get the SDK](#)
[Learn Visual Studio](#)
[Visit the dev center](#)

**WINDOWS**
Windows is reinvented with an all-new touch interface, Windows Store, and dev platform that lets you sell apps across the globe.
[Download the tools](#)
[Design apps](#)
[Develop your first app](#)
[Visit the dev center](#)

**WINDOWS PHONE**
Windows Phone is a different kind of phone. Get everything you need to design and build great apps and sell them in the new Store.
[Download the SDK](#)
[Build apps](#)
[Join the program](#)
[Visit the dev center](#)

**WINDOWS AZURE**
Windows Azure is an open and flexible cloud platform that enables you to quickly build, deploy and manage applications.
[Get the free trial](#)
[Get the tools](#)
[Explore the features](#)
[Visit the dev center](#)

**OFFICE**
Office is reinvented. Build a new class of apps for Office and SharePoint using familiar languages, tools, and hosting services.
[Download the tools](#)
[Learn to build apps](#)
[Sign up](#)
[Visit the dev center](#)

Figure 1: The MSDN home page, circa June 2013.

Other examples can be found in the world of mobile devices. Earlier this year I acquired an HTC 8X smartphone, which runs the Windows Phone 8 operating system. I found the user interface to be attractive, intuitive, and easy to use from the first day. Apple's iPhone, which of course came out well before the current generation of Windows phones, set the de facto standard for mobile user interface design, but even Apple is joining the modern trend with plans for a "smoother, cleaner, flatter" user interface design for the upcoming iOS 7. It may be debatable who was actually first with the modern design trend among these and other vendors, but it's clearly being adopted by many vendors on many types of devices.

So how does any of this apply to us as Visual FoxPro developers? The typical VFP app has been in use for several years and is probably beginning to look a little out of date. But the user interfaces we are able to create in VFP desktop apps are largely constrained by the tools provided by VFP – forms, controls, menus, and dialogs. When used without much customization, these tools tend to produce apps with a uniform gray appearance. So how do we "break out of the gray box?"

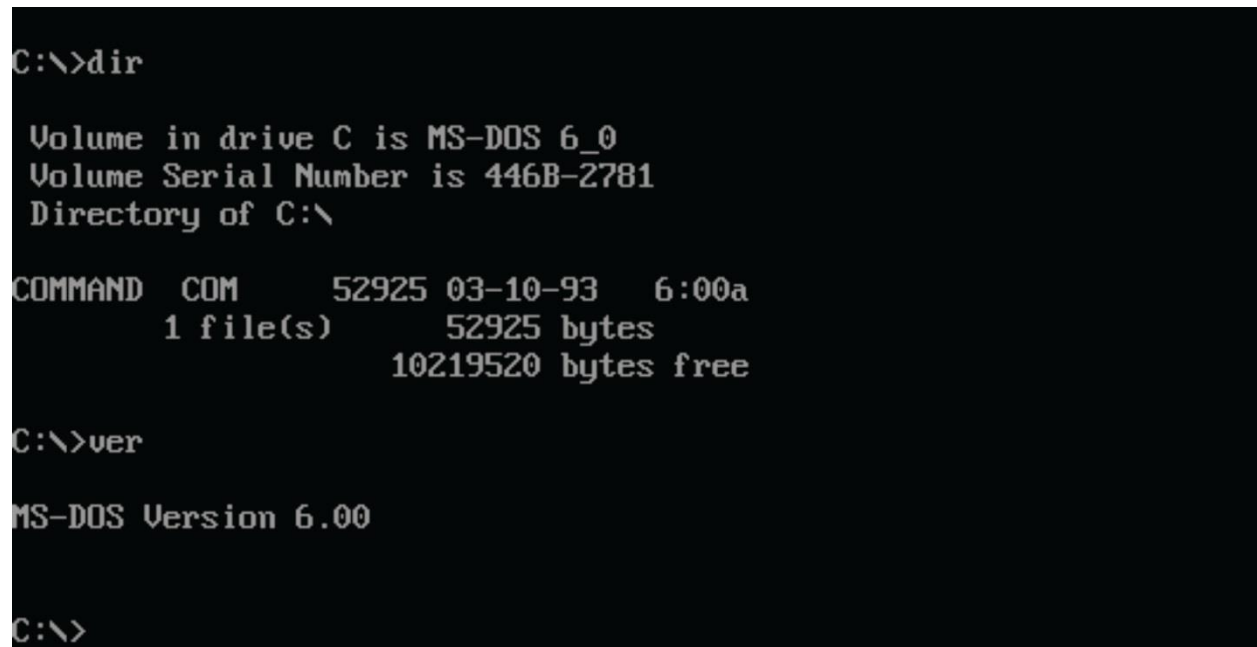
Fortunately, the VFP user interface tools are a lot more flexible than we may realize. Our challenge as VFP developers is to learn how to apply the concepts of "modern UI" design to give our apps a facelift, improving their appearance and appeal for years to come.

Evolution of the user interface

To know where we're going with user interface design, and why, it's helpful to spend a few minutes reviewing how we got where we are today.

Character-based user interface

Although by no means the first or the only character-based electronic user interface, MS-DOS ushered in the era of personal computing for most people. We all learned how to use the command line, how to CD to a specific folder, how to type DIR to see what was in there, and how to launch a program by typing its file name. This was simple stuff for most users.

A screenshot of an MS-DOS command prompt window with a black background and white text. The prompt is 'C:\>'. The user has entered 'dir', and the output shows volume information for drive C (MS-DOS 6.0, serial 446B-2781) and a directory listing for 'C:\'. The listing shows a single file 'COMMAND.COM' with a size of 52925 bytes. Below this, it shows '1 file(s)' and '52925 bytes' used, with '10219520 bytes free' available. The user then enters 'ver', and the output is 'MS-DOS Version 6.00'. The prompt returns to 'C:\>'.

```
C:\>dir

Volume in drive C is MS-DOS 6_0
Volume Serial Number is 446B-2781
Directory of C:\

COMMAND  COM      52925  03-10-93   6:00a
          1 file(s)      52925 bytes
                        10219520 bytes free

C:\>ver

MS-DOS Version 6.00

C:\>
```

Figure 2: In the beginning, there was DOS.

Think about how you interact with the command line: you use a keyboard. The keyboard was already familiar to anyone who'd ever used a typewriter, so the learning curve for the command line interface was not very steep.



Figure 3: The computer keyboard was familiar to anybody who'd ever used a typewriter.

Graphical user interface

MS-DOS and its counterparts from other vendors had a good, long run, but then along came a whole new paradigm called the Graphical User Interface in the form of Windows and others. What made these graphical user interfaces so different from the command-line? Suddenly, users were confronted with an array of objects on the screen and not a command line in sight. How were you supposed to interact with this? Where were you supposed to type anything? The keyboard became a second-class citizen, and everybody had to learn to use this new thing called a mouse.

The mouse was unfamiliar to most people because it didn't correlate with anything they already knew. People had to learn how to move it, how to point, how to click.



Figure 4: The mouse was unfamiliar because it didn't correlate with anything people already knew.

Do you remember the first time you used a mouse? Was it easy? Was it obvious how you were supposed to interact with the things you saw on the screen? And what *were* those things on the screen, anyway? Which ones did something and which were merely decorations?

In order to help make the new GUI interfaces seem more familiar and easier for people to use, designers worked to made the virtual things on the screen look like familiar things from the real world. This was the beginning of *skeuomorphism* in user interface design.

Skeu-oh-what??

One way to make something new seem more familiar is through the use of skeuomorphism. A *skeuomorph* is a ten-dollar word for "an element of design or structure that serves little or no purpose in the artifact fashioned from the new material but was essential to the object made from the original material".³ An example from the world of tangible objects is the simulated woodgrain decals applied to the exterior door panels and fenders of some automobiles from the 60's and later, which was meant to resemble the appearance of the real hardwood construction used in some car models in the 50's and earlier.



Figure 5: A 1940's era station wagon with real hardwood construction, alongside a skeuomorphic decal on a 1966 Buick Roadmaster Woody station wagon. First image from www.seriouswheels.com, second image from www.americandreamcars.com.

In the software world, skeuomorphism evolved as a way of helping people make a mental connection between the objects they saw on this new thing called a graphical user interface and familiar objects from the real world.

Think of the early Windows calculator, which was designed to look like a physical one (and still does, for that matter). Once people learned that clicking a button with the mouse on the Windows calculator was the equivalent of pressing a button with their finger on a real calculator, the mental connection was made and using the GUI started to become intuitive.

³ en.wikipedia.org/wiki/skeuomorph

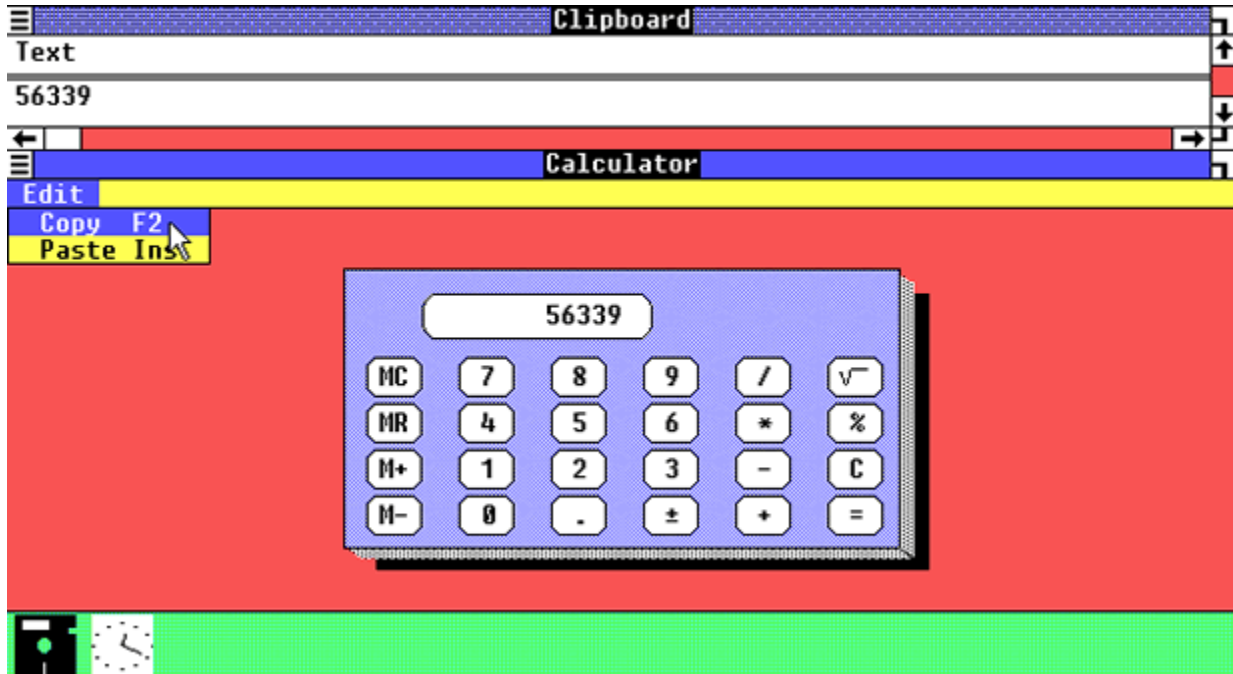


Figure 6: The calculator from an early version of Windows was a crude skeuomorphic representation of a physical calculator.

In the early days, skeuomorphism sometimes ran wild. The Microsoft Bob operating system interface took the concept to the extreme. Although it was a marketplace failure, it still stands as a prime example of this type of design.



Figure 7: The Microsoft Bob user interface represents skeuomorphism taken to the extreme in software design.

Current examples of skeuomorphism can be found in the Bookshelf and Notepad apps on Apple iPhones, iPods, and iPads. The user interface for both of these apps is carefully designed to look like their real-world counterparts.

For years these skeuomorphic design concepts carried over to the design of nearly every element used in software user interfaces, including things like 3-D borders to give a sense of depth or relief to certain shapes along with command buttons designed to look like real buttons you could depress with your finger if they existed on a physical control panel.

Over time, skeuomorphs became less and less necessary as users became more and more familiar with the basics of interacting with a GUI. Many people have now grown up with these interfaces since childhood and need few if any visual reality clues. *Of course* I know I'm supposed to click here – I don't need it to look like a real button. *Of course* I know this area on my screen isn't really raised or sunken – I don't need it to look that way any more.

And so, software skeuomorphs began to go the way of the dinosaur. Today, we see modern design concepts replacing the old ones as user interfaces trend toward a flat design with a more symbolic, cleaner, and less cluttered appearance.

Design Concepts

Before jumping into what can be done to modernize the user interface in VFP apps, it's helpful to keep a few fundamental design concepts in mind.

Embrace space

The appropriate use of negative space, also referred to as whitespace, is fundamental to the overall design of any visually pleasing graphic. The forms and screens we design for our VFP apps are no exception.

In the old days, when we had to work with a maximum design area of 800 x 600 (or, worse, 640 x 480), we concentrated on cramming as much information as possible into the viewable area on the screen. This was a natural and completely appropriate approach to forms design at that time.

Those constraints have long been lifted. The ever-increasing physical size of computer screens has brought with it a corresponding increase in available screen real estate, but, as developers, our sense of appropriate form design may not have kept pace. We may still tend to think of whitespace as wasted space, but we need to realize that it can in fact be essential to an aesthetically pleasing design.

When designing a web page or a Windows form, it can be helpful to think in terms of *content density* as measured along a scale of loose vs dense. Less whitespace results in higher content density, while more whitespace allows for looser density. As larger monitors and greater screen real estate grow ever more common on users' desktops, software developers are able to design for looser content density. The goal is to achieve both a more visually appealing design and a more readily usable interface.

The lesson for software developers is to embrace space as another valuable element in their designer's toolbox.

Use color

Color is equally essential design element. The appropriate use of color can greatly enhance both the appearance and the usability of a graphic interface. It can go a long way towards eliminating the mostly gray appearance you typically see in an "out of the box" VFP app.⁴

⁴ This is a good place to mention Kevin Ragsdale's excellent "*It's so, uh, gray...*" *Tips & Tricks to Improve Your App's User Interface* session from Southwest Fox 2011. Go back and read his whitepaper if you have access to it. Kevin provides a wealth of detail and many good examples on ways to get rid of the gray, along with other tips for improving the appearance of VFP apps. I'm indebted to Kevin for his ideas, some of which I've built on in this paper.

Unless you've been living under a rock, you've probably seen countless examples of the Windows 8 color palette, or variations on it, in many places on the Web as well as in the latest versions of some commercial desktop apps.

There seem to be a lot of references on the Web to "the Windows 8 color palette", some official and others probably not. Figure 8 shows one resource I've found particularly useful.



Figure 8: The Windows 8 color palette, from jasongaylord.com/blog/windows-8-color-palette.

When embarking on any given design project, one decision you'll want to make early on is the type of color theme to use. The basic choice is between a light theme, employing a white background with darker color elements, or a dark theme, using a dark gray or black background with lighter color elements. Personally I prefer a light theme in Visual FoxPro apps, but I see a lot of dark themed applications these days, particularly in graphic design and photo editing software, so users are probably growing accustomed to both.

The color palette is a starting point, but you're not constrained by it. If you see a color you like on the Web or in an app, grab it. As far as I know, nobody can patent a color. There's a terrific little Windows app called *Pixie*⁵ for grabbing colors you like. Fire it up, click anywhere on your screen and *Pixie* shows you the color values for that pixel. I think it was Doug Hennig who first introduced me to *Pixie* – thanks, Doug.

Flat design

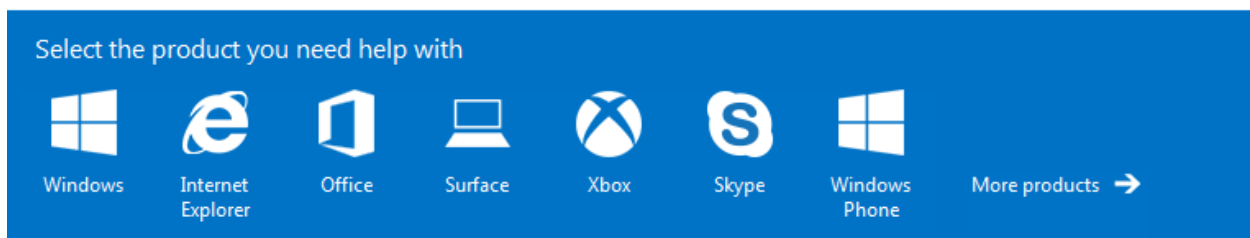
One of the most obvious design elements in the modern UI is the use of flat icons, which are a subset of the whole flat design paradigm. Flat icons make no attempt to look like real 3-D objects but are instead designed to be a simple, often minimalist, representation of the function they enable or the information they provide. In this way they are similar to the design of the common road signs we see and take for granted every day – Stop, Yield, Merge, Danger, etc.



"Flat is good, right?"

Flat icons need not be bland or colorless. Indeed, as we saw earlier, the appropriate use of color is an important element of a good graphical design and that applies to icons, too. While it's common to see flat icons with a white foreground image or text on a solid background color, or the reverse with a color foreground against a white background, there are many examples of multi-color flat icons, too.

Some designers use the same color for all icons in a group, as in Figure 9 from a Microsoft Support web page. Other implementations use both a unique shape and a unique color to provide the visual distinction between icons, as in Figure 10 from the yahoo.com website.



⁵ *Pixie* is available for download from www.nattyware.com/pixie.php.

Figure 9: This portion of a Microsoft Support web page uses monochromatic icons.

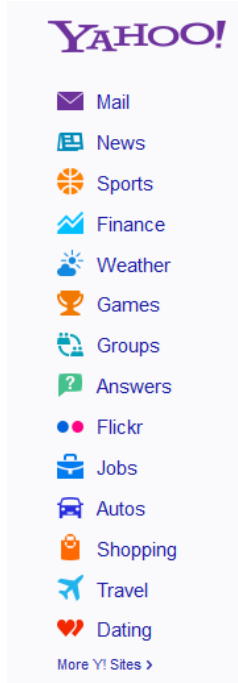


Figure 10: This sidebar navigation panel from yahoo.com uses flat icons, each with a unique shape and color to identify its content area.

Both of the above examples illustrate the use of flat icons in place of old-style icons that were designed with 3-D effects such as color gradients and a simulated light source. The flat icon effect yields a much cleaner interface with no loss of usability.

As another example of a modern UI found on the Web, consider the screenshot of the main content area of the Ruby developers page on the Windows Azure website, shown in Figure 11. The page is clean, the content areas are clear, and there’s nothing to distract from what’s important – a great example of achieving “less is more” with modern design concepts. This is the type of effect we can strive for in our Visual FoxPro applications, too.

Tutorials and Resources



Compute

Host website on Linux VM

Deploy website with Capistrano

Customize a domain name

Deploy with command-line

[SHOW ALL](#)



Data Services

Store data in blobs

Store data in tables

Manage and analyze data

[SHOW ALL](#)



App Services

Message between apps

Develop app with Service Bus

Communicate with Queues

Serve content with CDN

[SHOW ALL](#)

RDOC // Find
documentation for Windows
Azure Libraries for Ruby



OPEN SOURCE // View and
contribute to source code
on GitHub



FORUMS // Ask questions,
share insights, and discuss
the platform



Figure 11: The main content area of the Ruby developers page on the Windows Azure website is a great example of a nice, clean design. www.windowsazure.com/en-us/develop/ruby/

Finally, take a look at the main toolbar in version 6 of the Axialis IconWorkshop desktop app. Not surprisingly, this most recent version of an app for creating icons has itself adopted the flat design paradigm.



Figure 12: The Axialis IconWorkshop desktop app uses flat icons on its main toolbar.

Modernizing your VFP app

There are several areas where you can make changes to enhance the appearance of your VFP app's user interface. This section covers the areas where you can make the most difference: fonts, forms and controls (labels, text boxes, grids, etc.), menus, and system dialogs. Some of these changes take a fair amount of work while others require almost no effort at all.

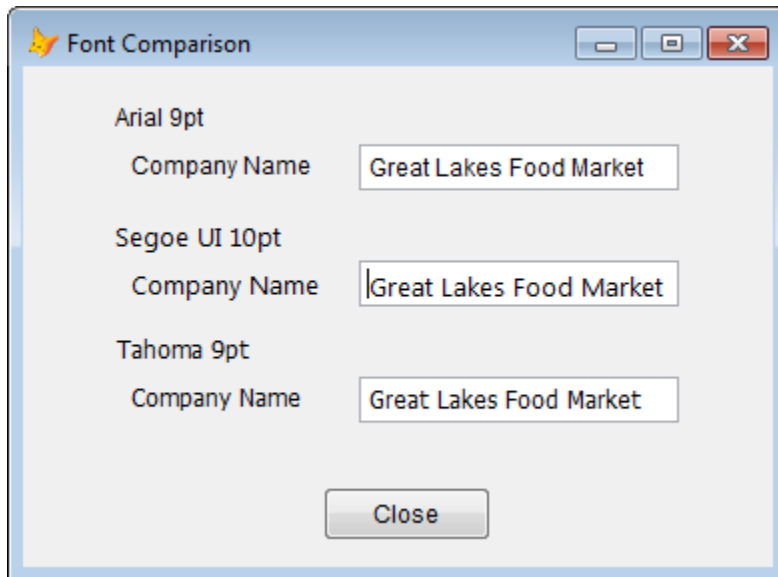
Fonts

Are you still using Arial as the default font for your forms and controls? You might be surprised to know that Arial continues to be the default even in VFP 9.0 SP2. There's nothing inherently wrong with Arial but it can make your app look out of date, especially when users are running it alongside other apps that use more modern fonts.

To give your VFP apps a more modern appearance, one of the easiest things you can do is switch to a modern font. The source authority for fonts and other design elements in modern Windows applications is the *User Experience Interaction Guidelines for Windows 7 and Windows Vista* and the *Windows 8 User experience guidelines* publications. Both are available for download in PDF format from the Microsoft website.

The default font for Windows 7 and later is Segoe UI 9pt for most controls, while the default font for Windows XP is Tahoma 8pt. The visual difference between these two fonts and Arial is subtle but noticeable.

Using Segoe UI or Tahoma in the standard size brings your app in compliance with the UX guidelines, but the trend I observe in modern UI development is toward larger font sizes than were typically used in the past. With that in mind, I decided for my own use to bump the default font sizes up by 1 pt. The default fonts I use in my own VFP apps are therefore Segoe UI 10pt on Windows 7 and above, and Tahoma 9pt on Windows XP. Figure 13 shows the appearance of these two fonts compared to Arial on a sample VFP form in both Windows 7 and Windows XP.



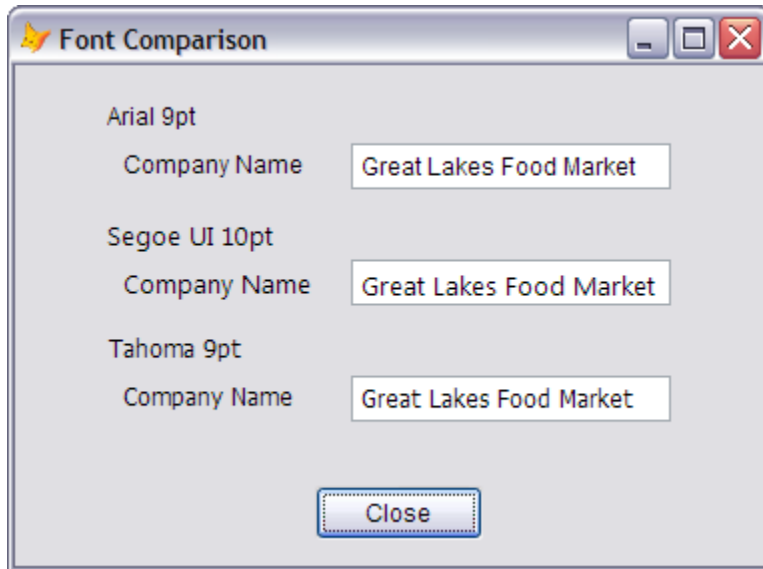


Figure 13: A comparison of Arial 9pt font to Segoe UI 10pt and Tahoma 9pt on Windows 7 (top) and Windows XP (bottom). Segoe UI is the default font for Windows 7 and above, but it is not installed by default on Windows XP machines.

You can of course go about making the change by editing the font of each individual control in every form in your apps, but that's a very tedious process. Once you're fully committed to using the newer fonts, it's much easier to simply to update your base class controls and forms – you *are* using base classes, aren't you? – to use the desired font.

If your apps will be running on Windows XP as well as Windows 7 and later, set the default font to the appropriate choice for the version of Windows you use in your primary development environment. Then add code in the base class Init methods to switch to the other font at runtime if the runtime version of Windows differs from your development environment.

Appendix A contains the code I use to switch between these two fonts at runtime, with the choice being determined by the version of Windows the app is running on. Placing this code in the Init() method of each control's base classes results in a "set it and forget it" solution.

I do my design and development work primarily on Windows 7, so I've made Segoe UI 10pt the default font in all my base classes. The Init() method code checks for this font and changes it to Tahoma 9pt at runtime if the app is running on Windows XP. Note that the code changes the font to Tahoma 9pt *only* if the app is running on Windows XP and *only* if the control is using Segoe UI 10 pt. This is so any controls intentionally designed to use another font family or font size remain unchanged.

If you're updating your app from Arial 9pt to Segoe UI 10pt or Tahoma 9pt, you'll want to review your forms and see if labels, text boxes, and other controls need to be made wider. Labels of course you can change by inspection at design time. For text boxes, a good rule of thumb is to set the width to the maximum content length in characters multiplied by 8. For

example, use a width of 160 for a text box designed to hold a 20-character field, and a width of 240 for a 30-character field. These widths comfortably accommodate both Segoe UI 10pt and Tahoma 9pt. This enables you to design in one font and rely on the base class code to dynamically switch to the other font at runtime without needing to adjust the width or height of labels and other controls.⁶

Of course, nothing compels you to use the Microsoft standard fonts. If you prefer to use the same font for all versions on Windows including Windows XP but still want to stick close to the standards, choose Tahoma over Segoe UI because the latter is not installed by default on Windows XP machines.

Forms

Perhaps more than any other single design element, forms are the "face" of your app from the user's point of view. The overall color scheme, font style, menu layout, arrangement of controls, and other design elements of your app are of course also factors, but users spend most of their time interacting with the forms that provide access to their data. It therefore makes sense to focus most of our attention on what we can do to make our forms appear more modern and appealing.

I built a sample form for this session to demonstrate the concepts being applied here. This form enables the user to view, edit, and navigate among the rows in the Employee table of the Northwind database. The form is derived from a base class that includes VCR-style controls to enable navigation plus the standard CRUD (create, retrieve, update, and delete) functions. For this reason, I refer to it as a "navigation form".

The idea in this section of the paper is to trace the evolution of this form beginning with its original grayish appearance through several incremental steps and finishing with a fully modernized version. The original version and the completed new version of the sample form, along with the base classes and other dependencies, are included in the session downloads. A copy of the Employees table from the Northwind database is also included in case you don't have your copy handy.

The original form

The base class for the sample form in its original state is *frmNavform_Old*, which is found in the *navform.vcx* class library. This class, in turn, derives from a top level base class named *frmBaseform_Old* located in the *baseform.vcx* class library. Although this level of complexity is not strictly necessary for a simple sample form, a lot of this sample is adapted from my own application framework so some of the class hierarchy came along for the ride.

Figure 14 shows the original form in design mode in its "out of the box" format, using native VFP base class controls with no modifications.

⁶ Grids are an exception – see the section on *Other things to consider* later in this paper.

The screenshot shows a Windows-style application window titled "Employee". The window has a standard Windows 7 title bar with minimize, maximize, and close buttons. The main area of the window is filled with a light gray dotted pattern. It contains two columns of text boxes. The left column has labels "Title", "Salutation", "First Name", "Last Name", "Address", "City", "Region", "Postal Code", and "Country" followed by text boxes labeled "txtTitle", "txtSalutation", "txtFirstName", "txtLastName", "txtAddress", "txtCity", "txtRegion", "txtPostalCode", and "txtCountry". The right column has labels "Home Phone", "Extension", "Birth Date", and "Hire Date" followed by text boxes labeled "txtHomePhone", "txtExtension", "txtBirthDate", and "txtHireDate". Below these text boxes is a "Notes" section with a label and a text area labeled "edtNotes". At the bottom of the window is a navigation bar with several icons: a left arrow, a right arrow, a magnifying glass, a document icon, a calendar icon, a printer icon, and a folder icon.

Figure 14: The original version of the sample form, in design mode. All screenshots are from Windows 7 unless otherwise noted.

In my opinion this is not a bad looking form, although I'm admittedly biased because I designed it. I've used various instances of this base class form extensively in several apps over the years, and my clients have been happy with it. But when seen from the perspective of modern UI design, this form definitely has an old-school appearance.

The navigation form has two modes: view mode, which is the default when the form is first opened, and edit mode, which happens when the user clicks the Edit button in order to make changes to the data. In view mode the text boxes are disabled and have a dark gray background, while in edit mode the textboxes are enabled and have a white background.

Figure 15 shows the form in view mode at runtime.

Figure 15: The original form at runtime, in view mode. Note the text boxes have a dark gray background to indicate they're disabled.

There are several things to note about this version of the form, things we'll be changing as we go along.

- The background color (BackColor property) of the form is the default 240,240,240, which is a light gray.
- The labels and controls use the default Arial 9pt font.
- The labels use the default background color of 240,240,240, which is the same as the form's background color and therefore makes them appear transparent.
- The title bar has a caption and shows the default Visual FoxPro logo along with the default Minimize, Maximize, and Close buttons.
- The navigation buttons are VFP command buttons, using bitmap images that ship with VFP.

Clicking the Edit button switches the form to edit mode. Code in the base class sets the Enabled property of the text boxes True for editing and changes their background color to white to indicate they're editable. Also, in edit mode the Edit and Delete buttons are assigned different icons to become the Save and Cancel buttons, while all the other buttons are disabled.

Figure 16 shows the navigation form at runtime in edit mode.

The screenshot shows a VFP form titled "Employee" with a yellow star icon. The form is in edit mode, with all text boxes enabled and having a white background. The data entered is as follows:

Field	Value
Title	Sales Representative
Salutation	Ms.
First Name	Nancy
Last Name	Davolio
Address	507 - 20th Ave. E.
City	Seattle
Region	WA
Postal Code	98122
Country	USA
Home Phone	(206) 555-9857
Extension	5467
Birth Date	12/08/1968
Hire Date	05/01/1992
Notes	Education includes a BA in psychology from Colorado State University. She also completed "The Art of the Cold Call." Nancy is a member of Toastmasters International.

The toolbar at the bottom contains several buttons. The Save button (floppy disk icon) and the Cancel button (curved arrow icon) are highlighted with a blue border, indicating they are enabled. Other buttons include navigation arrows, a grid icon, a document icon, a print icon, and a refresh icon.

Figure 16: The original form at runtime, in edit mode. Note the text boxes are enabled and have a white background. Also note that only the Save and Cancel buttons on the toolbar are enabled in edit mode.

One final note about the original version of this form. A long time ago, for whatever reason, I decided to use a darker gray background color for my form base classes. So in actual use, the form looks like what's shown in Figure 17.

Figure 17: The original form as it actually appeared in view mode in my app at runtime, using the non-standard, darker gray background color.

I bring this up because one consequence of that decision is that labels, whose base class background color I did not change from the default, no longer appear transparent at runtime. I'm at a loss to remember why I did this, although it's probably because black text would have been difficult to read on the dark gray background so I kept the labels' background lighter. It must have seemed like a good idea at the time, but I don't particularly like the look of this form today. More importantly, my users are probably getting tired of it, too — it's dark without being attractive.

The first iteration

As a first step towards toward modernization of this form, I not only reverted to the original lighter gray background but went all the way to white⁷. One reason for this is the emergence of white as by far the most common background color in modern design, including websites. Most of us, including our users, spend a lot of time on the Web these days, and the design styles we see there influence our ideas of what modern design "should" look like, even in a desktop app. So pay attention to what you see on the Web.

The first iteration toward modernization of the sample form is illustrated in Figure 18.

⁷ Actually, the background color is an off-white RGB(254,254,254), which is indistinguishable from full white to the naked eye. Text is often set to an off-black RGB(2,2,2). For a discussion of the reasons to use off-white and off-black instead of full white and full black, see Kevin Ragsdale's *"It's so, uh, gray..."* paper I referenced earlier. It's full of other great information and suggestions, too.

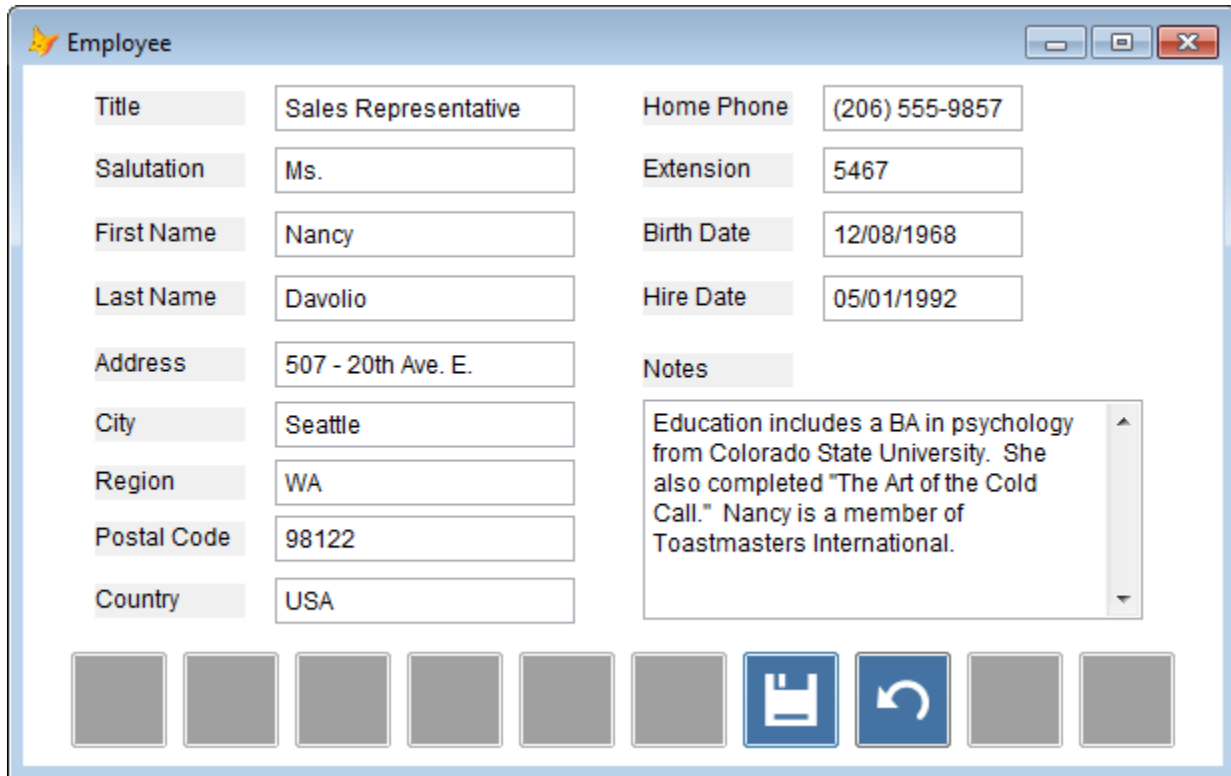
The screenshot shows a VFP form titled "Employee" in view mode. The form has a white background and a blue title bar. It contains several text boxes for data entry, organized in two columns. The left column includes fields for Title, Salutation, First Name, Last Name, Address, City, Region, Postal Code, and Country. The right column includes fields for Home Phone, Extension, Birth Date, Hire Date, and a Notes field. The Notes field contains a multi-line text entry. At the bottom of the form, there is a row of ten flat, square navigation buttons with icons for back, forward, search, add, edit, delete, print, and other functions.

Figure 18: The first steps toward modernizing the appearance of this form, shown in view mode at runtime.

There are a couple of obvious changes between this iteration and the original form: the background color has been changed to white, and the appearance of the navigation buttons has been dramatically altered by replacing the original bitmaps with new flat icons and a new shape.

Several other design elements remain unchanged. The form still has a full-height title bar with a caption, the VFP icon, and the default Minimize, Maximize, and Close buttons. The labels still have a light gray background and therefore no longer appear transparent against the white background. And although they have new icons and a new square shape, the navigation buttons are still VFP command buttons. Notice the border around each button: that's not part of the icon image, it's generated by VFP. To achieve a pure flat design, we'd like to get rid of that border.

Figure 19 shows the same form in edit mode.



The screenshot shows a VFP form titled "Employee" with a standard Windows-style title bar. The form contains several text boxes for data entry, organized into two columns. The left column includes fields for Title, Salutation, First Name, Last Name, Address, City, Region, Postal Code, and Country. The right column includes fields for Home Phone, Extension, Birth Date, and Hire Date. Below these fields is a "Notes" section with a scrollable text area. At the bottom of the form, there is a row of ten command buttons. The first six buttons are disabled and have a gray background. The seventh button, which contains a save icon (a floppy disk), is enabled and has a blue background. The eighth button, which contains a refresh icon (a circular arrow), is also enabled and has a blue background. The remaining two buttons are disabled and have a gray background.

Title	Sales Representative	Home Phone	(206) 555-9857
Salutation	Ms.	Extension	5467
First Name	Nancy	Birth Date	12/08/1968
Last Name	Davolio	Hire Date	05/01/1992
Address	507 - 20th Ave. E.		
City	Seattle		
Region	WA		
Postal Code	98122		
Country	USA		

Notes

Education includes a BA in psychology from Colorado State University. She also completed "The Art of the Cold Call." Nancy is a member of Toastmasters International.

Figure 19: The form in edit mode, using PNG images on the command buttons.

Oops! What happened to the images on the disabled navigation buttons? They disappeared when the buttons' Enabled property was changed to False. This happened because I used PNG images for the new command buttons, and VFP doesn't handle that image format as expected when the command button is disabled. In a later iteration of this form I'll replace the command buttons with image controls, but for now the problem can be solved by using bitmap images instead of PNG images.

Figure 20 is the same form in edit mode but using bitmap images instead of PNGs. This time, VFP handles the disabled state as expected, so the images still appear but with the desired gray background.

Figure 20: The form in edit mode, using bitmap images on the command buttons. Now the images on the disabled buttons still appear as expected when the form is in edit more.

We're getting closer to what we want here, but there are still a lot of things to do. For starters, I don't like the border around the command buttons. In a pure flat design, those icons should not have a border, so let's see what we can do about it.

The second iteration

In the second iteration of the form the command buttons have been replaced with VFP image controls, as shown in Figure 21.

The screenshot shows a Windows-style application window titled "Employee". Inside, there's a form with various input fields for employee data. The fields are arranged in two columns. The first column includes Title, Salutation, First Name, Last Name, Address, City, Region, Postal Code, and Country. The second column includes Home Phone, Extension, Birth Date, Hire Date, and a Notes field. The data entered is: Title: Sales Representative, Home Phone: (206) 555-9857, Extension: 5467, Birth Date: 12/08/1968, Hire Date: 05/01/1992, Address: 507 - 20th Ave. E., City: Seattle, Region: WA, Postal Code: 98122, Country: USA. The Notes field contains text about Nancy Davolio's education and Toastmasters membership. At the bottom of the form is a row of ten image controls: four navigation arrows (back, previous, next, forward), a search magnifying glass, a plus sign, a pencil (edit), a trash can (delete), a printer, and a document with an arrow (save). These image controls are not enclosed in a separate border.

Figure 21: The form in view mode using image controls instead of command buttons. Note that there is no longer a border around the navigation buttons.

Using image controls instead of command buttons gets rid of the unwanted border around the navigation buttons, but it also introduces a couple of complexities. For one thing, unlike a command button, there is no automatic change in the appearance of an image control when it's disabled. Therefore, when using image controls we have to use code to change the Picture property in order to provide the user with the visual cue that the button is disabled. We also need to use code to change it back again when the button is re-enabled.

The base classes from which the sample form is derived use a data-driven approach to dynamically change the icons at runtime. When the form is launched, code in these classes builds three collections of images – one each for the enabled, disabled, and mouseover state of the icon – from image files stored in folders specified in custom properties of the form. At runtime, other code then sets the Picture property to the appropriate image for each button depending on the state of the form.

Figure 22 shows the properties sheet for one version of the form, where you can see the custom properties and methods involved. The code gets a little complicated, but it's all there in the session downloads if you'd like to explore it.

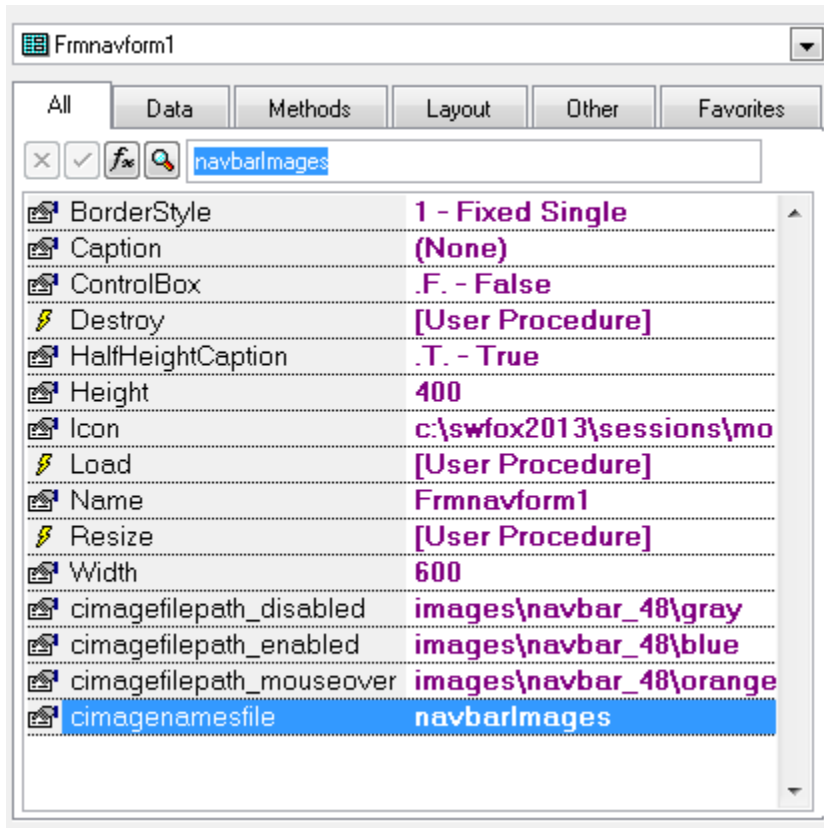


Figure 22: Base class code uses the values of the four "cImage..." properties to create collections of icons, which are then assigned to the navigation buttons at runtime depending on the state of the form.

The other issue is that command buttons have a *Default* and a *Cancel* property which image controls lack, so again it's necessary to implement this behavior in code. In the command button version of my navigation form, the Cancel property of the Close button is set to True while in view mode so the form closes if the user presses the Escape key. In edit mode, the Cancel property of the Undo button is set to True so the user can cancel changes and return to view mode by pressing the Escape key.

Both of these are convenient behaviors I didn't want to lose. To implement them in the image control version of the form, I decided the simplest way was to add a couple of miniature command buttons that are visible at design time but get which moved behind their corresponding image control at runtime so they're not visible to the user. The base class code that sets and resets the Cancel property of these two buttons in the command button version of the form therefore still works in the image control version, and the behavior is preserved.

Figure 23 shows the image control version of the form in design mode. You can see the two miniature command buttons near the bottom right, under their corresponding image controls. As mentioned earlier, the Picture property of the navigation buttons themselves is set at runtime, which is why they appear blank at design time.

The screenshot shows a VFP form titled "Employee" with a light blue header bar. The form has a dotted background. It contains several text boxes for data entry, arranged in two columns. The left column includes fields for Title, Salutation, First Name, Last Name, Address, City, Region, Postal Code, and Country. The right column includes fields for Home Phone, Extension, Birth Date, Hire Date, and a Notes area. The Notes area is a multi-line text box with a vertical scrollbar. At the bottom of the form, there is a row of ten square image controls, each containing a light gray 'X' icon. Two of these image controls are slightly larger than the others. In the bottom right corner, there are two small, rectangular command buttons, one of which is visible at design time but hidden at runtime.

Figure 23: The second iteration of the navigation form, using image controls instead of command buttons. Note the two miniature command buttons near the bottom, which are visible at design time but hidden at runtime.

With all of this in place, the navigation form has now taken another step toward a more modern UI using true flat icons while preserving the functionality of the original command buttons.

Figure 24 shows the form in edit mode at runtime. In edit mode, the Picture property of the disabled navigation buttons changes to display the same icons but with a light gray background. The two enabled icons still have the original blue color to indicate they're still enabled, but they now have different icons representing their Save and Cancel function while in edit mode.

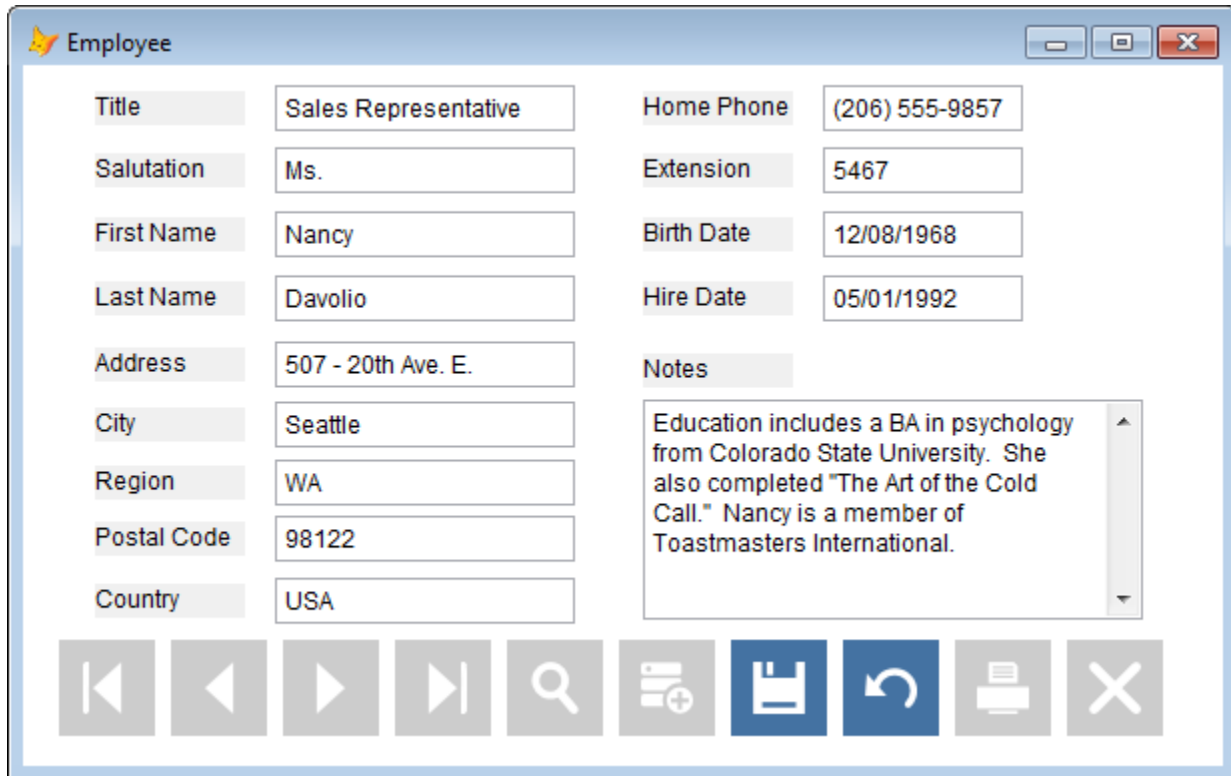


Figure 24: The form in edit mode using image controls instead of command buttons. The appearance of the disabled navigation buttons is handled in code by using a different picture.

The third iteration

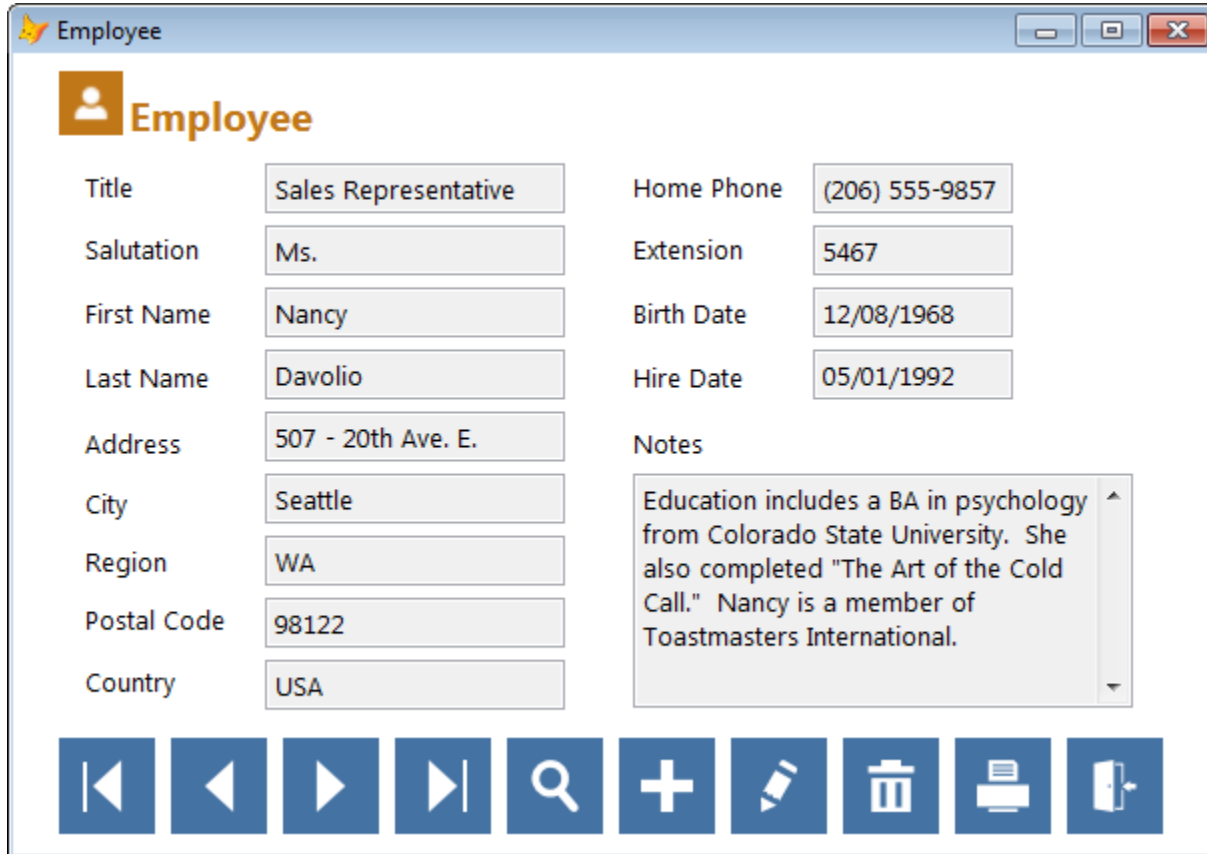
Although we've made a lot of progress, there is still some work to be done.

For one thing, the labels still have a light gray background and do not appear transparent against the form's white background. There are a couple of ways to handle this, including changing the background color of the labels to be the same as that of the form or setting the label's BackStyle property to Transparent. Either of these can be accomplished by changing each individual label or by creating a subclass.

Going forward, I knew I would want the vast majority of my labels to be transparent so I created a subclassed label control with its BackStyle property set to Transparent. I now use this class as the default label on all forms. I then have to change only the small minority of labels I *don't* want to be transparent. The labels in the third iteration of the sample form are instances of this transparent label class.

Another change along the path towards a more fully modern UI concerns the standard Windows title bar. In the modern design paradigm, a form (or a web page) typically has a header area above its content. The header area provides space for a relatively large icon and title to visually differentiate it from the content area below. Therefore the height of the header area is much greater than the height of the standard Windows title bar. Although we're not specifically targeting Windows 8 here, the *Page layout design* section of the *Windows 8 User Experience Guidelines* document is a good reference for this concept.

It's easy enough to make this change by simply increasing the height of the form and adding an attractive icon and label in a complementary color, as shown in Figure 25. After making this change, the header and body content areas of the form look pretty good but the icon and caption in the standard Windows title bar are beginning to look redundant.



The screenshot shows a VFP form window titled "Employee". The title bar is blue and contains a yellow star icon, the text "Employee", and standard Windows window controls (minimize, maximize, close). The form itself has a white background. At the top left of the form is a blue square icon with a white person silhouette, followed by the word "Employee" in a large, bold, blue font. Below this is a data entry form with two columns of fields. The left column contains: Title (Sales Representative), Salutation (Ms.), First Name (Nancy), Last Name (Davolio), Address (507 - 20th Ave. E.), City (Seattle), Region (WA), Postal Code (98122), and Country (USA). The right column contains: Home Phone ((206) 555-9857), Extension (5467), Birth Date (12/08/1968), Hire Date (05/01/1992), and a Notes field. The Notes field contains the text: "Education includes a BA in psychology from Colorado State University. She also completed 'The Art of the Cold Call.' Nancy is a member of Toastmasters International." At the bottom of the form is a blue bar containing ten white icons: a left arrow, a double left arrow, a right arrow, a double right arrow, a magnifying glass, a plus sign, a pencil, a trash can, a printer, and a door.

Title	Sales Representative	Home Phone	(206) 555-9857
Salutation	Ms.	Extension	5467
First Name	Nancy	Birth Date	12/08/1968
Last Name	Davolio	Hire Date	05/01/1992
Address	507 - 20th Ave. E.		
City	Seattle		
Region	WA		
Postal Code	98122		
Country	USA		

Notes
Education includes a BA in psychology from Colorado State University. She also completed "The Art of the Cold Call." Nancy is a member of Toastmasters International.

Figure 25: The form has a nice new title area, but now the standard title bar now looks redundant.

We can get rid of the default VFP icon by replacing it with a blank icon, and we can get rid of the word Employee on the title bar by setting the form's caption to *None*. This yields the result shown in Figure 26.

Figure 26: The title bar is still there, but overall the form looks much cleaner after removing the icon and the caption.

One downside of removing the caption is that, when minimized, the form appears as a blank shape with no icon and no caption. Not very useful, is it?



Figure 27: With no title bar icon or caption, the minimized representation of the form is not very useful.

This is actually easy to deal with. Using a bit of custom code in the `Resize()` event method, as shown in Listing 1, we can set the caption to be the desired value when the form is minimized and blank for all other states. This code also takes care of setting the form's `Icon` property to show the desired icon in the title bar when the form is minimized but a blank icon when it's not.

Listing 1: This `Resize()` method code sets the form's `Caption` and `Icon` properties depending on the `WindowState` of the form.

```
* frmNavForm.Resize()  
WITH this  
    .Caption = IIF( .WindowState = 1,"Employee", "")  
    .Icon = IIF( .WindowState = 1, "user.ico", "blank.ico")  
ENDWITH  
DODEFAULT()
```

With this code in place, the minimized form has the expected appearance with both an icon and a caption, as shown in Figure 28.

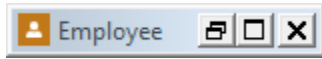


Figure 28: The form now has the expected icon and title when minimized.

This would actually be a pretty good place to stop if we wanted to. All functionality from the original version of the form has been preserved, but the form now has a much cleaner, more modern appearance. Compare Figure 26 to Figure 17 to see how far we've come.

If you've read this far, congratulations. Feel free to pause and take a breather. When you're ready to keep reading, press on — there's one more iteration to come.

The fourth iteration

The ControlBox

Within the paradigm of modern design and flat icons, the appearance of the standard Minimize, Maximize, and Close icons on a form's title bar seems out of place. We can easily remove these icons by setting the form's `ControlBox` property `False`, but then we lose some functionality we probably want to preserve. So, what can we do?

I decided to start with the Minimize function. After getting rid of the standard minimize icon, we need to replace it with one of our own. We then need to implement the desired functionality in code.

To do this, I created a new 16x16-pixel icon and placed it in the upper right corner of the body of the form. This icon's color is consistent with the rest of the form, and its flat style is appropriate for a modern design. The icon's appearance mimics the appearance of the standard Minimize icon so its function should be obvious to the user, but I set its *ToolTipText* to "Minimize" as an additional visual cue.

Figure 29 show how the form looks after making these changes. Note that the form's title bar is now completely empty and has therefore become merely a design element instead of a functional one. Because of this, I decided to set the form's *HalfHeightCaption* property `True` to achieve what I consider a more balanced appearance.

Figure 29: The form in view mode with the Title area added. The default title bar is now empty (no icon and no min, max, or close buttons) and a custom minimize button has been added to the body of the form.

The new minimize icon's functionality is implemented in its Click event method, as shown in Listing 2. This code simply changes the form's WindowState to 1 (minimized).

Listing 2: The custom Minimize icon's functionality is implemented in its Click() event method.

```
thisform.WindowState = 1
```

At this point we need to modify the custom code in the Resize() event method so the control box *does* appear when the form is minimized but goes away again when the form's size is restored. We can also remove the line that sets the Icon property and simply set it at design time in the usual way. By setting the ControlBox property True or False according to whether the form is or is not minimized, the icon automatically shows or doesn't show. The updated Resize() event method code is shown in Listing 3.

Listing 3: The form's ControlBox property is set True when the form is minimized, otherwise it's False.

```
* frmNavForm.Resize()
WITH this
    .Caption = IIF( .WindowState = 1,"Employee", "" )
    .ControlBox = .WindowState = 1
ENDWITH
DODEFAULT()
```


The same kind of thing can be done with customized Maximize/Restore and Close icons if desired, but I'm not sure they're needed. The sample form already has a Close button on the navigation toolbar, and I question how often anyone uses the Maximize icon even if the form is resizable. So for our purposes here, I decided there was no need for those two icons.

Mouse effects

Another way to make the form more visually appealing is to add mouse effects to the toolbar icons, changing their color to indicate they're "hot" as the mouse moves over them.

The functionality for this is baked in to the *frmNavForm_imageButtons* class with code in the *MouseEnter()* and *MouseLeave()* events, along with a set of icons with identical shapes but different colors than the originals. The presence of an appropriate value in the form's *cImagePath_mouseover* property, as shown in Figure 30, triggers base class code that builds a collection of mouse effects images and sets the form's *IMouseFX* property True, which in turn activates the mouse effects at runtime.

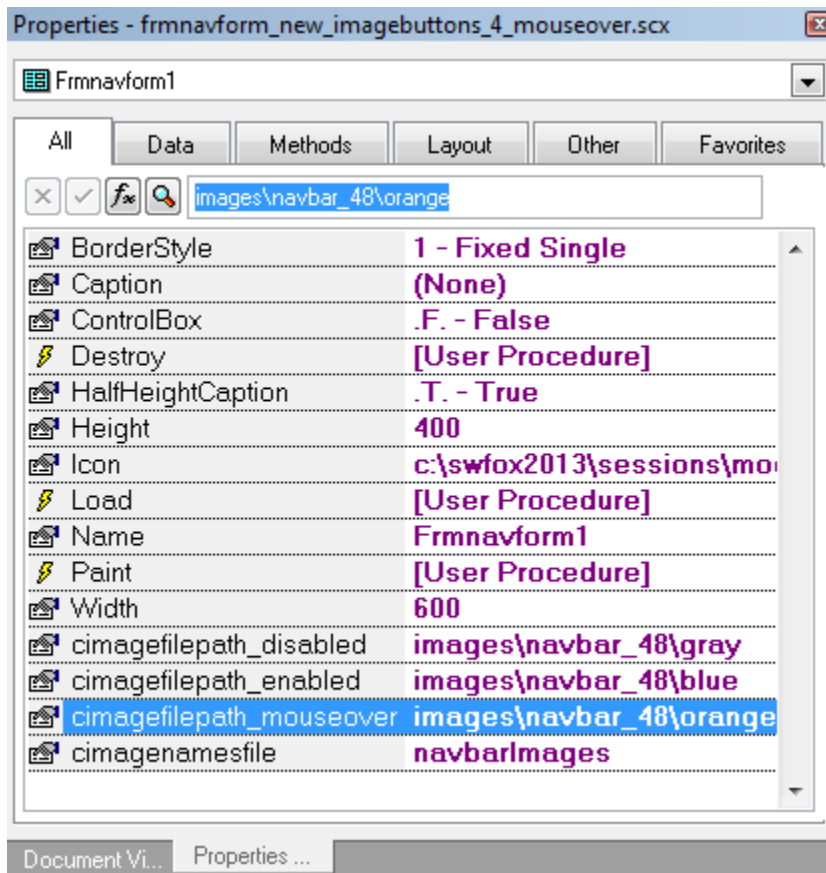


Figure 30: The mouse effects are implemented simply by specifying a folder with a set of icons in a different color, in this case orange. Class code takes care of all the details.

With this in place, each toolbar icon changes color as the mouse enters or leaves it. Figure 31 shows the effect as the mouse passes over the Edit button.

Employee			
Title	Sales Representative	Home Phone	(206) 555-9857
Salutation	Ms.	Extension	5467
First Name	Nancy	Birth Date	12/08/1968
Last Name	Davolio	Hire Date	05/01/1992
Address	507 - 20th Ave. E.	Notes	Education includes a BA in psychology from Colorado State University. She also completed "The Art of the Cold Call." Nancy is a member of Toastmasters International.
City	Seattle		
Region	WA		
Postal Code	98122		
Country	USA		

Navigation icons: Back, Previous, Next, Forward, Search, Add, Edit (active), Delete, Print, Help.

Figure 31: The form in view mode with mouse-over effects added.

And there you have it – our final result. Our old, gray form got a total makeover and now sports a modern appearance while retaining the full functionality of the original. This form is included in the session downloads as *frmNavform_new*.

Icon sizes

The standard image formats for Windows icons are 16x16, 24x24, 32x32, and 48x48. I think the 48x48 size I've used on the sample form is appropriate, but some people might consider it a bit large. On a smaller form, icons that big would almost certainly look out of place.

With that in mind, the base classes were designed to make it relatively easy to create an instance of the form using different size icons. The most time consuming task is actually creating the icons.

Figure 32 shows how the form looks using 32x32 icons. A container class for the navigation icons would also be an improvement and would make it easier to substitute one size for another.



Employee

Title	Sales Representative	Home Phone	(206) 555-9857
Salutation	Ms.	Extension	5467
First Name	Nancy	Birth Date	12/08/1968
Last Name	Davolio	Hire Date	05/01/1992
Address	507 - 20th Ave. E.		
City	Seattle	Notes Education includes a BA in psychology from Colorado State University. She also completed "The Art of the Cold Call." Nancy is a member of Toastmasters International.	
Region	WA		
Postal Code	98122		
Country	USA		

Navigation icons: Back, Previous, Next, Forward, Search, Add, Edit, Delete, Print, Help.

Figure 32: The form in view mode using 32x32 icons instead of 48x48.

Other things to consider

This section extracts and summarizes a couple of things that are embedded in the discussion above, as well as presenting some additional considerations you may run into.

Icons

If you don't want an icon to appear on your form's title bar, you can't simply set the Icon property to *None* because then you get the default VFP icon. One way to have no icon at all is to set the ControlBox property False, but this also eliminates the Minimize, Maximize, and Close buttons, which may not be what you want. The alternative is to create a blank 16x16 icon and use it as the form's Icon property.

There is one side effect to using a blank icon. If the form's ControlBox property is True and the form has a caption, the blank icon results in a blank space to the left of the caption, as shown in Figure 33. If this is undesirable, use a non-blank icon and either make it visible at all times or use code as in Listing 1 to turn the icon and the caption on or off depending on the WindowState of the form.

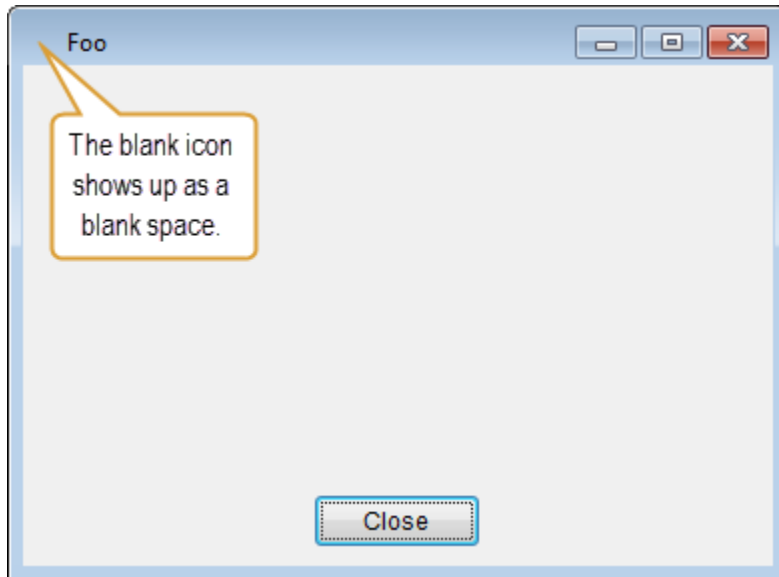


Figure 33: If a form's ControBox property is True and the form has a caption, a blank icon shows up as a blank space to the left of the caption.

There are several good software tools for creating and editing icons. My long-time favorite is IconWorkshop from Axialis, and Syncfusion's Metro Studio is a cool new tool. There are also numerous sources for downloading icons on the Web, many of which are free. The Resources section of this paper has links for some of these tools and download sources; a Web search will turn up many others.



I've been using the term "icon" as a generic way of talking about the images used on menus and toolbars regardless of whether they are .ico, .gif, .png, .bmp, or some other image format. The *Graphics Support in Visual FoxPro* topic in the VFP Help file says VFP 8.0 and later can handle all image formats supported by GDI+ on Windows XP and later. However, I've run into some weird runtime anomalies with .png and .bmp files, such as a blank area appearing where an icon should be or an icon showing up in the wrong size. The problems could certainly be due to my own ignorance or mistakes, but in any case changing to .gif images resolved them in at least one situation. I've also encountered error 1167, "Icon is corrupt or in wrong format", when working with certain .ico files, particularly if the file contains an icon in RGB/A format. In those situations, adding an identical icon in 32-bit (RGB) or even 8-bit (256-color) format has usually solved the problem.

Themes

When you set out to create an app or a form using the modern UI design paradigm, one of your first decisions is to choose a theme. The basic choice is between a light theme or a dark theme, followed by a choice of colors appropriate for that theme.

In a light theme you'd typically use a white or off-white background with black or near-black text and bright colors for images and icons. The sample forms in this session are based on a light theme. A dark theme is essentially the inverse, employing a black, near-black, or dark gray background with white or off-white text and muted colors for images and icons.

The Solarize and Zenburn themes available in some popular text editors are examples of dark themes. I use a version of a dark theme in my VFP IDE. Take a look at the screenshot in Figure 34 and consider how you could adapt it to a VFP form.

```
* Build a collection object of keys and images from navbarImages.dbf
LOCAL loCollection as Collection
loCollection = CREATEOBJECT( "collection")
IF NOT USED( "navbarImages")
    USE navbarImages IN 0
ENDIF
LOCAL lcPath
lcPath = GETDIR()
SELECT navbarImages
SCAN
    loCollection.Add( lcPath + ALLTRIM( navbarImages.cFileName), ALLTRIM( navbarImages.cKey))
ENDSCAN
```

Figure 34: I use a color scheme based on a dark theme in my VFP IDE.

Labels

Except in special circumstances, labels look best when made to appear with a transparent background. This can be accomplished either by assigning them the same background color as their parent object or by setting the BackStyle property to Transparent. If you want the majority of your labels to be transparent, consider making this setting in your label base class. That way, the only exceptions you'll have to deal with are the few you don't want to be transparent.

Text boxes

When using dynamic font control code to switch between Segoe UI 10pt and Tahoma 9pt for text boxes and other controls at runtime, your design time environment will likely differ from at least one of the possible runtime environments. How then do you determine the appropriate width for text boxes and other controls at design time in order to avoid potential truncation of their contents at runtime?

As mentioned earlier, a good rule of thumb is to multiply the maximum number of characters the text box is designed to hold by eight. For example, if a text box is designed to hold a maximum of twenty characters, use a width of $20 \times 8 = 160$. Use a width of 240 for a thirty-character field, and so on. A width of 110 works well for a formatted phone number like (123) 456-7890.

Grids

When using dynamic font control code with grids, I've found it's sometimes necessary to save and restore the value of the grid's HeaderHeight and RowHeight. Otherwise these properties seem to revert to the default values when the font is changed, resulting in headers and rows that may not be tall enough to accommodate their entire content in the new font. The code for handling this can be found in the grids section of Appendix A.

Menus

There isn't a lot you can do about the appearance of menus in a VFP app, but there are a couple of ways to customize them. One is with icons and the other is with fonts.

Menu icons

One easy way to improve the appearance of your app is to customize its menu pads with icons. If you're striving for a modern UI appearance, flat icons can be a good choice.

Figure 35 shows the menu pad from a tool I developed for working with the Mercurial version control system from within VFP. Although the menu pad uses the default font, the use of flat icons helps to give it more of a modern appearance.

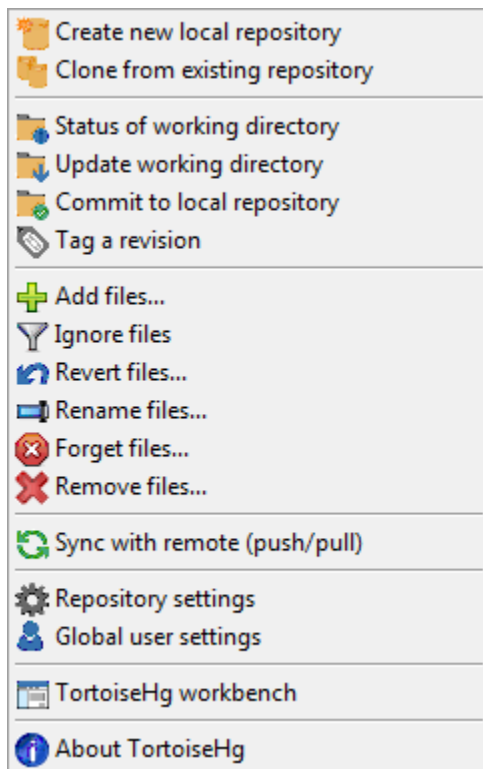


Figure 35: This menu pad uses icons from a set of *Pure Flat 2013 Icons* from Axialis, the same company that makes IconWorkshop.

Menu fonts

As far as I know, there isn't any way to change the font on the main menu bar in a VFP app. If that's the bad news, the good news is you *can* change the font on menu pads and shortcut (aka popup) menus. This is accomplished with a neat little hack I found in a thread on the

MSDN Visual FoxPro forum.⁸ If this technique is officially documented somewhere I haven't found it, so credit goes to the author of the thread.

The trick is to append a font specification to the message in the Message field of the menu's Prompt Options dialog. The font specification is a string formatted as

```
" FONT 'SEGOE UI',10 "
```

Place this string in a variable – *pcMenuFont* in this example – and append its value to the Message field as using macro substitution, as shown in Figure 36.

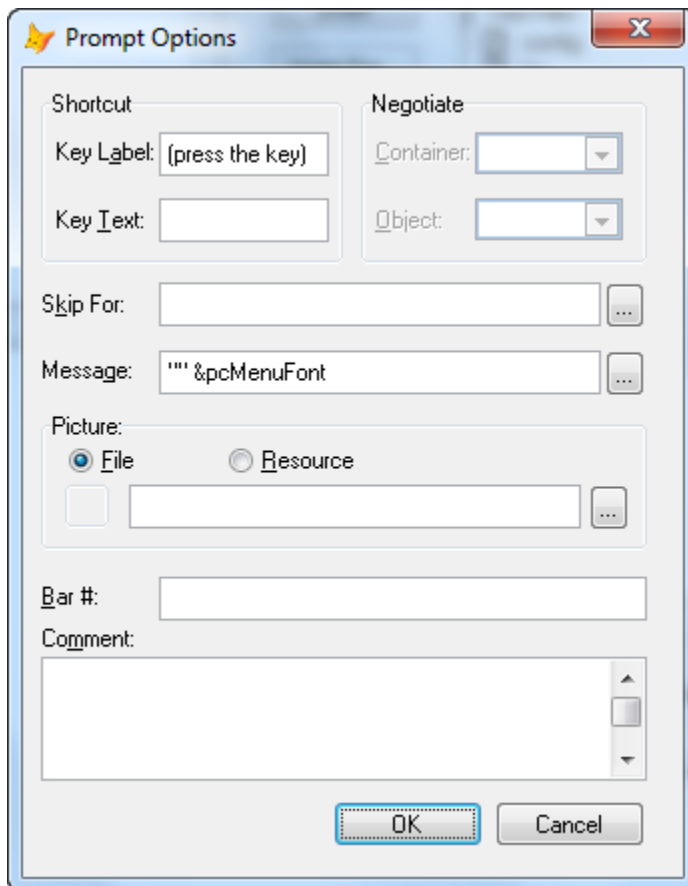


Figure 36: You can control the font used in popup menus by appending a font specification in the Message field of the menu's Prompt Options dialog.

Note that the font specification variable must follow a message string. In other words, you can't use only the font specification variable by itself, although the message string can be empty if you don't want to insert an actual message. It's a bit difficult to see in Figure 36, so Listing 4 shows the Message field in a more readable form, with an empty message string and the font specification inserted via macro substitution.

⁸ social.msdn.microsoft.com/Forums/en-US/visualfoxprogeneral/thread/2a57282c-b333-42a6-9e9f-3237cb326816/

Listing 4: A font specification can be inserted after the message string in the Message field of the Prompt Options dialog.

```
"" &pcMenuFont
```

For this session, I created a shortcut menu and hooked it up to the Click event of the Print button on the new, modern version of the sample form. I wanted the menu's font to match the font used in the rest of the form, so I created a private variable named *pcMenuFont* in the shortcut menu's Setup code and set its value according to the runtime version of Windows, as show in Listing 5.

Listing 5: Code in the shortcut menu's Setup method sets the desired font.

```
PRIVATE pcMenuFont  
pcMenuFont = IIF( VAL( OS(3)) < 6, " FONT 'TAHOMA',9 ", " FONT 'SEGUE UI',10 ")
```

The *pcMenuFont* variable is then referenced as via macro substitution in the Message field of each menu items' Prompt Option dialog, as explained above. The runtime result is shown in Figure 37.

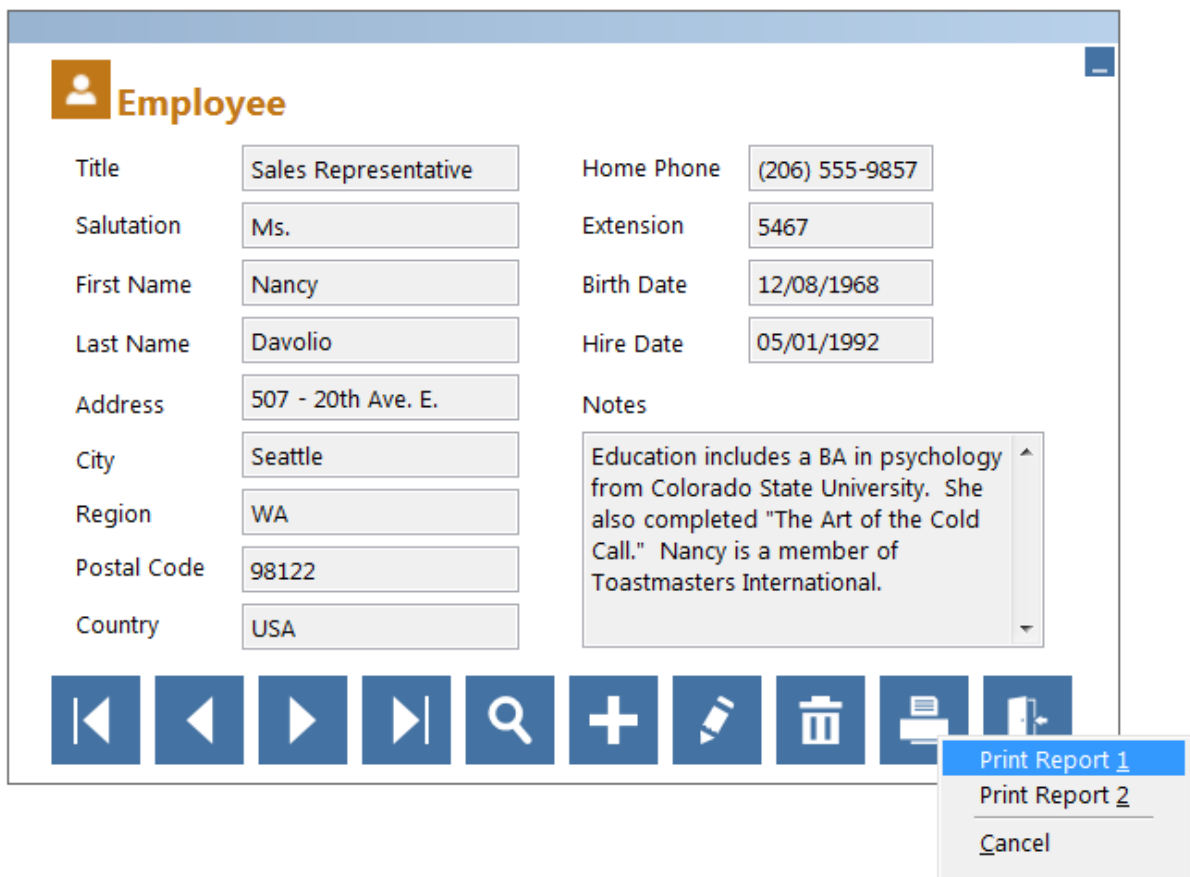


Figure 37: The shortcut menu off the Print button has been customized font to use Segoe UI 10pt font, matching the font used in the rest of the form.

Although I didn't do so in this example, I could also have added icons to the shortcut menu for further customization.

Main menu alternative

If you really want to go all out with the modern UI paradigm in your VFP app, you can consider doing away with the traditional main menu and instead using a set of icons docked to the top of the screen. It's not difficult to do this, although I'll admit that at this point I've had limited success getting it to look the way I really want it to.

Earlier we saw an example of this kind of toolbar on the Microsoft support site website, which is shown again in Figure 38 for easy reference.

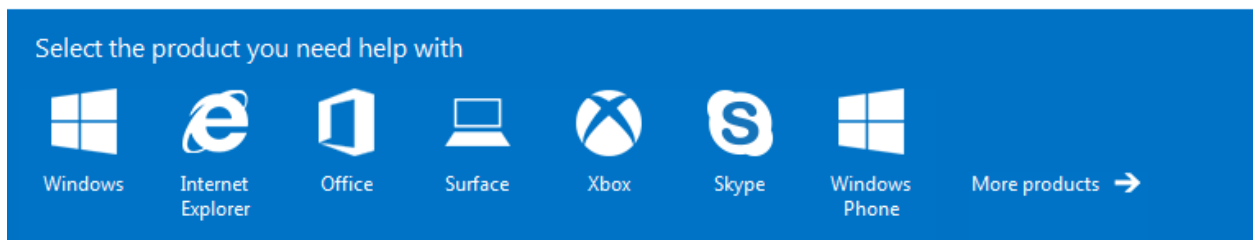


Figure 38: This portion of a Microsoft Support web page uses icons to implement the equivalent of a main menu. With the exception of the last one, the icons are 80x90. In each case, the text is part of the image.

Ideally, I'd like to be able to create something similar in a VFP app. At first I experimented with the toolbar control, but there were a couple of issues: the width of the toolbar didn't span the entire top of the application's screen the way I wanted it to, and the Toolbar control lacks a `BorderStyle` property so I couldn't get rid of the border. Figure 39 shows what the app looks like with a toolbar control in place of the main menu.

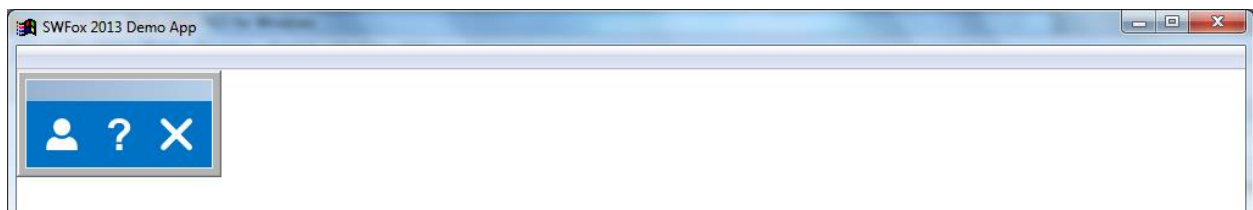


Figure 39: Using a toolbar control in place of the traditional main menu was less than ideal. The toolbar control has a border and its width doesn't span the entire screen.

Replacing the toolbar control with a form enabled me to take care of both the width problem and the border problem. The toolbar form still has an unwanted title bar that can't be avoided, but the overall appearance is better than before. After increasing the space between the icons, the app at runtime appears as shown in Figure 40.

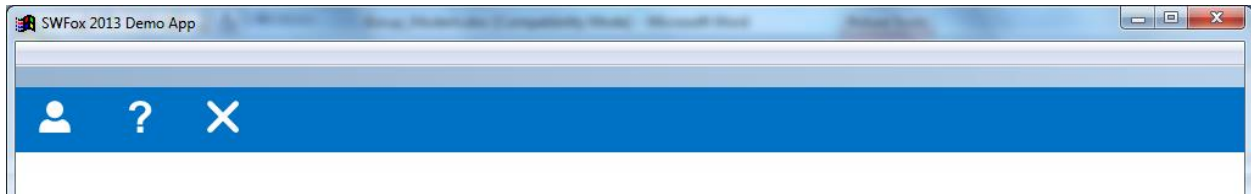


Figure 40: The width and border problems are solved by using a form instead of a toolbar.

We can take one more step to bring the overall effect closer to what we want by adding text labels beneath the icons, as shown in Figure 41.

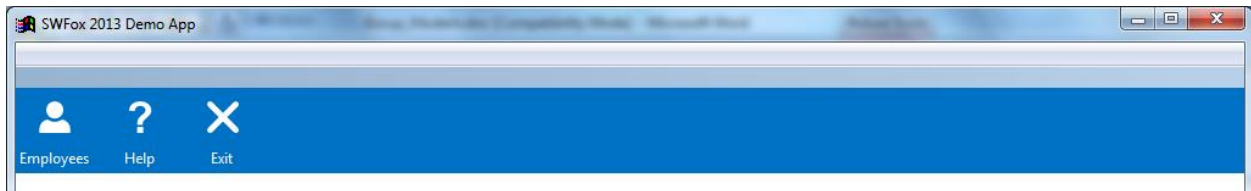


Figure 41: Adding labels and increasing the spacing helps the overall effect.

The icons in this example each perform a single action – opening a form, displaying a Help message, or exiting the app. For situations where an icon needs to link to more than one action, the icon’s Click() event can be made to open a shortcut menu with as many choices as needed. This is functionally equivalent to how clicking an item on the traditional main menu opens a menu pad or submenu.

I’m sure you can see how to continue with this idea by adding additional icons to complete the main toolbar and hooking up whatever shortcut menus are required to implement the app’s functionality. Whether this alternative to a traditional main menu is something you’d actually want to use in a live app is up to you, but the idea here is to demonstrate how it can be done.

The session downloads include the code and icons used in Figure 41.

System dialogs

If you use the GetDir() function in your VFP apps and haven’t already done so, do your users a favor and start using the new style *Browse For Folder* (BIF) dialog in favor of the older *Select Directory* dialog you get by default.

The syntax for the GetDir() function is:

```
GETDIR([cDirectory [, cText [, cCaption [, nFlags [, lRootOnly]]]])
```

By default, VFP brings up the old-style *Select Directory* dialog, as shown in Figure 42.

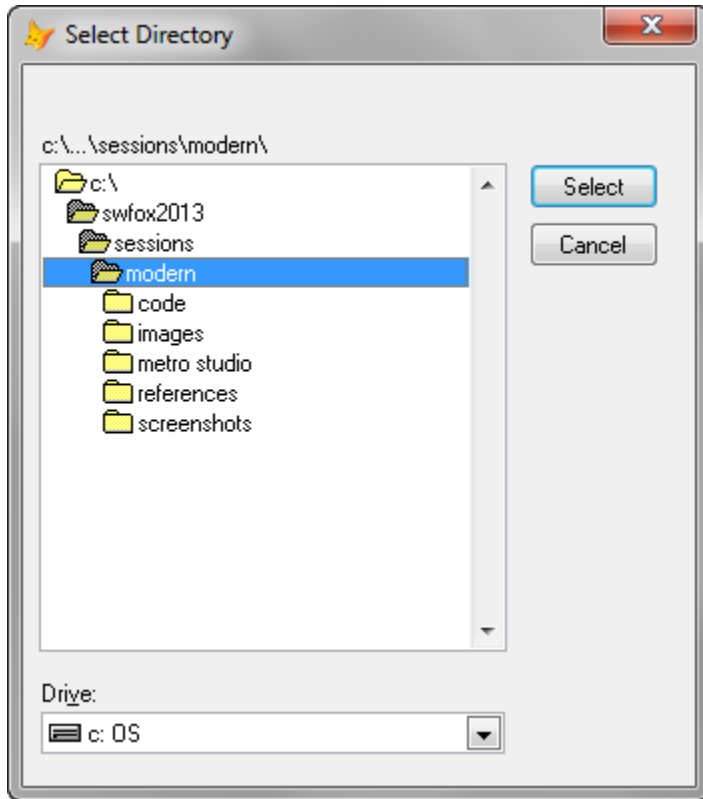


Figure 42: The old-style dialog for `GetDir()`.

The *nFlags* parameter can be used to set options for this dialog. The permitted values are commonly referenced as defined constants, which I've included in Appendix B for easy reference. These constants are also documented in the *GETDIR() Function* topic in the VFP Help file. It's a bit-level parameter so the values are additive if more than one is used.

You can improve both the appearance and the functionality of your app by passing the defined constant `BIF_NEWDIALOGSTYLE` as the *nFlags* parameter in every call to the `GetDir()` function. This tells VFP to display the new-style *Browse For Folder* dialog, as illustrated in Figure 43.

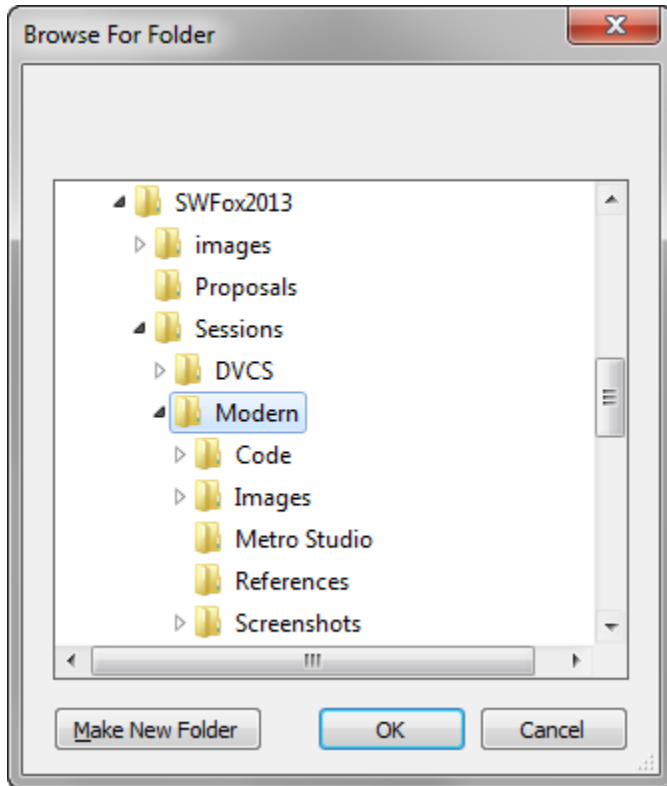


Figure 43: The new-style dialog for `GetDir()`.

Not only is this a better looking dialog, but it also includes new functionality such as the *Make New Folder* button. Other options for the *nFlags* parameter enable you to exert additional control over the functionality of this dialog.

Keep in mind that these options apply only to the `GetDir()` function. Other related functions such as `GetFile()` and `PutFile()` do not offer a choice of style or functionality.

Summary

The essence of the Modern UI design paradigm is to **do more with less** and to **put content before chrome**. It's a look you see in many places these days, both on the Web and in the latest releases of many desktop apps. The design is readily identifiable by its increased use of whitespace, larger fonts, and flat icons, all of which serve to focus the user's attention on the essential content in a pleasing manner without extraneous clutter.

Most VFP apps have been around for several years. Although they remain functionally as useful and valuable as ever, their appearance may be starting to look out of date when compared to the modern style users are now seeing elsewhere. The goal of this paper is to present ways you can modernize the appearance of your VFP apps, bringing them up to date and keeping them high on your users' list of favorites for years to come.

Resources

References

Windows 8 User Experience Guidelines (Win8_UXG_GA.pdf)

www.microsoft.com/en-us/download/details.aspx?id=30704

Designing UX for Apps

msdn.microsoft.com/en-us/library/windows/apps/hh779072.aspx

Make Great Windows Store Apps [formerly Make Great Metro style apps]

msdn.microsoft.com/library/windows/apps/hh464920.aspx

Index of UX Guidelines for Windows Store Apps

msdn.microsoft.com/en-us/library/windows/apps/hh465424.aspx

Modern UI Style Overview

www.slideshare.net/JakeTaylor1/windows-8-modern-ui-design-concepts

Microsoft Design Principles

msdn.microsoft.com/en-us/library/windows/apps/hh781237.aspx

Putting the Microsoft Design Language to Work

<https://skydrive.live.com/view.aspx?cid=FB0D8F97004979CD&resid=FB0D8F97004979CD!13711&app=PowerPoint>

Software

Axialis Icon Workshop

www.axialis.com

Syncfusion Metro Studio 2

www.syncfusion.com/downloads/metrostudio

Icon sources

20 Free Flat Icon Sets

speckyboy.com/2013/05/29/20-free-flat-icon-sets/

40 Free Icon Sets

speckyboy.com/2013/07/09/40-new-free-icon-sets/

Axialis Professional Stock Icons

www.axialis.com

Icon Experience

www.iconexperience.com/m_collection/icons/

Copyright © 2013 Rick Borup. Windows® is a registered trademark of Microsoft Corporation in the United States and other countries. Apple®, iPhone®, iPad®, and iPod® are registered trademarks of Apple, Inc. All other trademarks are the property of their respective owners. Images attributed to iStockPhoto are used under license; unattributed images are the property of their respective owners.

Appendix A

This is the code I use in my base classes to switch between the two standard fonts at runtime, depending on the version of Windows the app is running on. Placing this code in the Init() method of each control's base classes results in a "set it and forget it" solution.

I do my design and development work primarily on Windows 7, so I've made Segoe UI 10pt the default font in all my base classes. The Init() method code checks for this font and changes it to Tahoma 9pt at runtime if the app is running on Windows XP. Note that the code changes the font to Tahoma 9pt *only* if the app is running on Windows XP and *only* if the control is using Segoe UI 10 pt. This is so any controls intentionally designed to use another font family or font size remain unaffected.

Some controls require slightly different code than others, so the code is presented in sections according to the type of control. Also note that in some cases the code modifies properties other than the font. You are welcome to use this code in your own apps, but I recommend you take the time to understand what it does and make any changes you feel are necessary for your own use.

Generic Init method code:

```
IF VAL( OS(3)) < 6 && Windows XP or earlier
  WITH this
    IF.FontName = "Segoe UI" AND .FontSize = 10 && Developer's default setting
      .FontName = "Tahoma"
      .FontSize = 9
    ENDIF
  ENDWITH
ENDIF
```

Init method code for option groups:

```
LOCAL lni
FOR lni = 1 TO this.ButtonCount
  WITH this.Buttons[lni]
    .BackStyle = .Parent.BackStyle
    .BackColor = .Parent.BackColor
    .ForeColor = RGB( 2,2,2) && Option group does not have a ForeColor property.
    IF VAL( OS(3)) < 6 && Windows XP or earlier
      IF .FontName = "Segoe UI" and .FontSize = 10 && Developer's default setting
        .FontName = "Tahoma"
        .FontSize = 9
      ENDIF
    ENDIF
  ENDWITH
ENDFOR
```

Init method code for command groups:

```
IF VAL( OS(3)) < 6 && Windows XP or earlier
  FOR lni = 1 TO this.ButtonCount
    WITH this.Buttons[lni]
      IF .FontName = "Segoe UI" AND .FontSize = 10 && Developer's default setting
```

```
        .FontName = "Tahoma"
        .FontSize = 9
    ENDIF
ENDWITH
ENDFOR
ENDIF
```

Init method code for page frames:

```
IF VAL( OS(3)) < 6 && Windows XP or earlier
    FOR lni = 1 TO this.PageCount
        WITH this.Pages[l ni]
            .BackColor = this.Parent.BackColor
            IF .FontName = "Segoe UI" AND .FontSize = 10 && Developer's default setting
                .FontName = "Tahoma"
                .FontSize = 9
            ENDIF
        ENDWITH
    ENDFOR
ENDIF
```

Init method code for grids:

```
IF VAL( OS(3)) < 6 && Windows XP or earlier
    WITH this
        IF .FontName = "Segoe UI" AND .FontSize = 10 && Developer's default setting
            LOCAL lnHeaderHeight, lnRowHeight
            * Save header height and row height,
            * 'cause they get changed when the font changes.
            lnHeaderHeight = .HeaderHeight
            lnRowHeight = .RowHeight
            .FontName = "Tahoma"
            .FontSize = 9
            * Reset header height and row height.
            .HeaderHeight = lnHeaderHeight
            .RowHeight = lnRowHeight
        ENDIF
    ENDWITH
ENDIF
```


Appendix B

Here are the header file constants you can use as parameters with the new-style Browse For Folder (BIF) dialog. These constants can also be found in the VFP Help file in the *GETDIR()* Function topic. Note that BIF_USENEWUI and BIF_NEWDIALOGSTYLE are equivalent, so use whichever one you prefer.

```
* Browse in Folder (BIF) Constants
#define BIF_RETURNONLYFSDIRS    0x00000001    && 1
* Return only file system directories (physical locations). If a user selects
* folders that are not part of the file system, the OK button is grayed.

#define BIF_DONTGOBELOWDOMAIN   0x00000002    && 2
* Do not include network folders below the domain level in the tree view control
* (For example, My Computer and My Networks).

#define BIF_RETURNFSANCESTORS   0x00000008    && 8
* Return only file system ancestors. If a user selects anything other than a file
* system ancestor, the OK button is grayed.

#define BIF_EDITBOX              0x00000010    && 16
* The browse dialog includes an edit control in which the user can type the name of
* an item. Available on Windows 98 and above, or with Internet Explorer 4.0 or
* higher (assuming shell integration option selected). Requires version 4.71 of
* shell32.dll.

#define BIF_VALIDATE             0x00000020    && 32
* Validates the editbox contents. If the editbox is used, it is necessary to
* validate the user-specified content. If the user types an invalid name into the
* edit box, the Cancel button becomes the only selection available. This flag is
* ignored if BIF_EDITBOX is not specified.

#define BIF_USENEWUI             0x00000040    && 64
#define BIF_NEWDIALOGSTYLE       0x00000040    && 64
* Use the new user-interface. Setting this flag provides the user with a larger,
* resizable dialog box. Additional functionality includes: drag and drop capability
* within the dialog box, reordering, context menus, new folders, delete, and other
* context menu commands. Support in Windows 2000 and above. Requires version 5.00 of
* shell32.dll.

#define BIF_BROWSEINCLUDEFILES   0x00004000    && 16384
* The browse dialog will display files as well as folders. Available on Windows 98
* and above, or with Internet Explorer 4.0 or higher (assuming shell integration
* option selected). Requires version 4.71 of shell32.dll.
```