Developing Visual FoxPro Applications for Windows Vista

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com
Blog: http://doughennig.blogspot.com

Overview

Windows Vista changes the rules for many aspects of application development, including the user interface, dialogs, deployment, security, and file access. This document looks at things you need to know to create Vista-compatible applications with Visual FoxPro.

You will learn:

- Why your application can't run as administrator any more
- How Vista's User Account Control impacts your application
- Changes your installation executable needs to work properly
- Taking advantage of new Vista dialogs

Introduction

Windows Vista is Microsoft's newest version of its venerable desktop operating system. Whether you're planning on moving to Windows Vista anytime soon or not, it's likely your customers are, whether by upgrading an existing system or buying a new one with Vista pre-installed. While one of Vista's goals is backward compatibility with existing applications, that goal is somewhat compromised by other goals, including improved security and a revamped user interface. As a result, it's likely to have an impact on your VFP applications, ranging from minimal to serious.

This document examines the things you need to know to create Vista-compatible VFP applications. I'll start by briefly looking at what benefits Vista brings to users and why they may chose to move to it. I'll then discuss how Vista impacts applications in general and VFP applications in particular, especially when it comes to security, folder and Registry virtualization, and deployment. I'll then go over opportunities to take advantage of new Vista features, including user interface improvements, Windows Desktop Search, RSS feeds, and XML Paper Specification.

Why Windows Vista?

There are a lot of reasons to move from your current operating system to Vista.

- **It's cool.** This might not seem like a compelling business reason to go through the pain of changing operating systems, but the fact is that Vista is both visually very appealing and has some new features users love. The new Aero Glass user interface makes windows look prettier than before (see **Figure 1**). Windows animations make the desktop seem livelier and more responsive.
- **It has some great productivity improvements.** Windows Desktop Search is a powerful tool for locating files, images, video, and even applications, and is blindingly fast. SuperFetch and ReadyBoost make the operating system cache applications and data in system memory or even USB flash drives to greatly improve performance. The new Gadgets Sidebar put small yet useful applications, like an RSS reader, clock, or stock ticker, right on the desktop.
- **Security.** One of the best reasons to upgrade to Vista is improved security. Vista is by default locked down to a much greater degree than previous versions of Windows, so it's much easier to keep the nasties off your system.
- **Bug fixes.** Every piece of software has bugs, and one that has millions of lines of code is sure to have lots of them. Like most new versions, Vista fixes a lot of bugs found in previous releases, so chances are good that problems you had with XP or earlier versions are gone now. I'm not suggesting Vista is perfect—there are likely some new bugs—but given the size and length of the beta, the average user isn't likely to encounter many.
- **Your customers have it or will get it.** As this document details, there are issues you need to be aware of to make your VFP applications work properly on Vista machines. It's almost certain that your customers already have or will soon have at least one or two Vista systems; as soon as they buy a new machine, they'll get it. There's no way you can determine what issues your application has on Vista unless you test it.

For more information on the benefits of using Vista, see Microsoft's white paper titled "The Advantages of Running Applications on Windows Vista," available at http://msdn.microsoft.com/en-us/library/bb188739.aspx.

Security

One of the biggest changes in Windows Vista that affects applications is related to security. Let's look at what changes Microsoft made and how they impact our applications.

The Problem with Security

Although Microsoft has recommended against it for years, most users run as local administrators on their systems. There are at least three reasons for this:

- Most application installers require administrative privileges because they install ActiveX controls or DLLs in the Windows System folder.

- Many applications write to files, such as INI or data files, in their installation folder (usually a subdirectory of Program Files), which requires administrative privileges in Windows XP and higher.
- Changes to configuration settings, even things as innocuous as system date and time, require administrative privileges.

As a result, most users learn the hard way to run as an administrator. If they don't, they can't install software, many applications fail, and they can't change the simplest things.

The problem with running as a local administrator is that all processes also run as administrator. This includes "malware," an abbreviation for "malicious software," which includes things such as viruses, worms, Trojan horses, spyware, and adware. Malware with administrative privileges can do anything it wants to your system, usually in the background and without your knowledge: delete or corrupt files, send spam messages, participate in denial-of-service attacks on other systems, or send confidential information such as passwords and credit card numbers to other computers.

One solution to malware is to ensure all users normally run with the least amount of privileges on their systems. The problem, though, is that it wasn't easy to elevate privileges for tasks, such as installing software, that require administrative privileges. The user either had to log off and log back in as an administrator, switch to an administrator account, or use the not-well-known Run As function.

Windows Vista makes running as a standard user much easier through a new feature called User Access Control, or UAC.

User Access Control

User access control starts with the login process. In previous versions of Windows, when an administrative user logs in, Windows creates a single security token. This token includes most administrative privileges and most administrative security identifiers (SIDs). This means any process that runs while this user is logged in has these same privileges.

Windows Vista, on the other hand, creates two security tokens for administrative users: a standard user token and an administrator token. This is sometimes referred to as a "split token." After logging in, Windows starts the user's desktop using the standard user token rather than the administrator token. This means any process launched runs as a standard user, minimizing the access it has to the system.

You can see this if you run TestPrivileges.PRG which accompanies this document. It displays which privileges you have on your system by calling HasPrivilege.PRG, written by Christof Wollenhaupt. 0 means the privilege is unavailable, 1 means it's disabled, and 2 means it's enabled. Start VFP and run TestPrivileges.PRG. My system shows 1 for SeShutdownPrivilege, 2 for SeChangeNotifyPrivilege, 1 for SeUndockPrivilege, and 0 for everything else (yours may vary). Then start VFP as administrator (right-click its icon and choose Run as Administrator) and run TestPrivileges.PRG again. This time, a lot more privileges are available or enabled.

Certain tasks, such as installing an application or tasks that change something global on the system, require administrative rights to perform. You can tell which tasks require this because they appear with a shield icon. In Vista dialogs, the icon appears beside the task, as shown in **Figure 1**. In the case of programs, such as a SETUP.EXE that requires administrative privileges, Vista automatically adds the shield to the icon for the program.

Figure 1. The shield icon beside certain tasks indicates they require administrative privileges.

Depending on what type of user you're logged in as, one of two things happens when you try to perform an action that triggers UAC:

- If you're logged in as an administrative user, the UAC dialog prompts you to confirm the action as shown in **Figure 2**. If you choose Continue, Vista launches the process with the administrator security token, so it has full administrative access to the system.

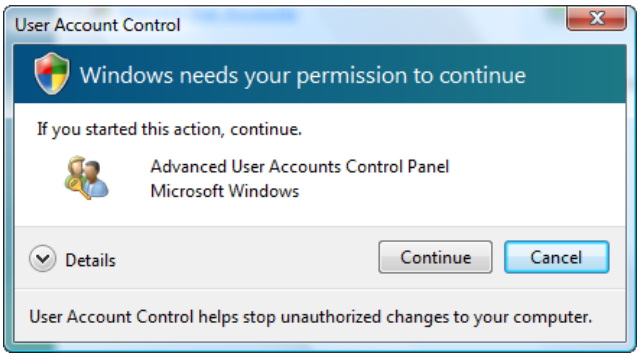

Figure 2. If you're logged in as an administrative user, the UAC dialog prompts you to confirm an action.

- If you're logged in as a standard user, the UAC dialog prompts you for the user name and password of an administrative user (see **Figure 3**). If the login is successful, Vista runs the process as the administrative user, so again it has full administrative access to the system.

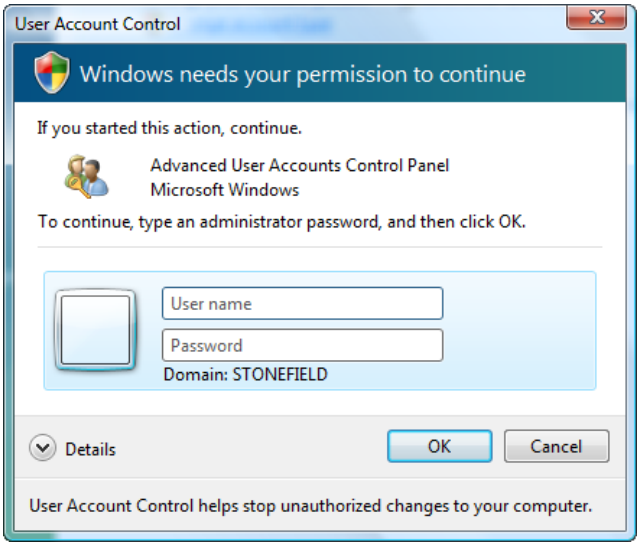

Figure 3. If you're logged in as a standard user, the UAC dialog asks you for the user name and password of an administrative user.

Administrative elevation, whether through the consent or login dialogs, is process-specific, so no other process has access to the administrative token.

Note that both UAC dialogs run in a different desktop, called the "secure" desktop, with the original one shown as a dithered bitmap. Only Windows processes can access the secure desktop, making it safer from malware trying to disguise itself as a Windows function.

There may be times when you need to turn off the secure desktop; for example, if you want to do screen shots of the UAC consent and elevation dialogs for documentation purposes, like I did in this document. To do so, bring up the Local Security Policy editor (**Figure 4**). A fast way to do that is to click Start, type "local sec," and, since "Local Security Policy" is likely the highlighted item in the list of matches, press Enter. Expand *Local Policies*, click *Security Options*, double-click the *User Account Control: Switch to the secure desktop when prompting for elevation* policy, and change the setting to Disabled. You must restart your system for the changes to take effect. Since this policy is enabled by default for good reason, be sure to re-enable it again once you've finished the tasks you disabled it for.

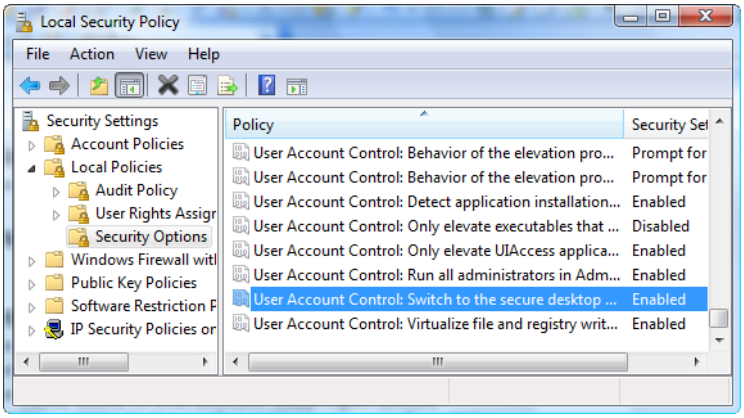

Figure 4. You can disable the secure desktop using the Local Security Policy editor.

Expect to see the UAC dialogs a lot, depending on how you use your computer. For example, it appears every time you run a program installer requiring administrative privileges (most do). Some people find this annoying (Apple did a humorous "I'm a Mac and I'm a PC" commercial exaggerating UAC; you can watch it at http://www.apple.com/getamac/ads/). I don't find this feature annoying because I know it's there to protect my system. If you'd rather work without UAC and risk the security problems it may cause, you can do so by selecting

User Accounts from the Control Panel, click User Accounts (yes, this seems redundant), select Turn User Account Control On or Off (this displays the UAC dialog), and turn off the setting in the dialog that appears; this requires a system restart to take effect.

Vista makes it easy to set up standard users. In fact, all user accounts except the first one (which is created automatically when you install Vista) are by default standard users. You have to specifically change them to be administrative users. I recommend setting up at least one standard user account so you can test your application both as administrator and standard user.

Protected Resources and Virtualization

Storing an application's data files in the correct place has gotten trickier under Windows Vista. Most developers simply write to INI files stored in the application folder (usually C:\Program Files\Some Folder) and store the data files in a Data subdirectory of that folder. While Microsoft strongly discouraged this practice in the past, now it's enforced; even if you're logged in as an administrator, writes to certain protected resources such as C:\Program Files or the HKEY_LOCAL_MACHINE hive in the Registry fail.

This would break most pre-Vista applications (or as Microsoft likes to call them, "legacy" apps), so to prevent that, they're run in XP compatibility mode. Through a process called *virtualization*, Vista redirects writes to protected resources to somewhere else. For example, it redirects writes to a file in C:\Program Files\Some Folder to C:\Users*username*\AppData\Local\VirtualStore\Program Files\Some Folder, where *username* is the name of the logged-in user. Writes to files in C:\Windows go to C:\Users*username*\AppData\Local\VirtualStore\Windows. Writes to the HKEY_LOCAL_MACHINE Registry hive are redirected to HKEY_CLASSES_ROOT\VirtualStore\Machine.

While it's great that Microsoft prevents our applications from breaking, this really is just a short-term solution, for several reasons. First, the user can turn off virtualization (I'll show you how later), so attempts to write to protected resources fail rather than being redirected. Second, according to a Microsoft white paper, "The Windows Vista Developer Story: Windows Vista Application Development Requirements for User Account Control" (http://msdn2.microsoft.com/en-us/library/aa905330.aspx), "Developers must not rely on virtualization being present in subsequent versions of Windows." The most important reason, though, is that Vista virtualizes these resources on a per-user basis, so other users on the system won't see them.

For example, suppose Bob logs into a Windows XP system, runs the Super Happy Fun Ball application, and changes some settings in the Options dialog. The application saves these settings in App.INI in the program directory. When Sally logs in and runs the application, she uses those same settings. The intention is that these settings are global or else the developer would have stored them in some user-specific location such as the HKEY_CURRENT_USER hive in the Registry.

Now suppose Bob logs into a Windows Vista system and does the same thing. Thanks to virtualization, the changes he makes are written to App.INI in C:\Users\Bob\AppData\Local\VirtualStore\Program Files\Super Happy Fun Ball. When Sally logs in and runs the application, it looks at the settings in App.INI in C:\Users\Sally\AppData\Local\VirtualStore\Program Files\Super Happy Fun Ball, so she doesn't use Bob's settings.

It's even worse with data. If the application stores its tables in the Data subdirectory of the application folder, when Bob adds records, they go in MyTable.DBF in C:\Users\Bob\AppData\Local\VirtualStore\Program Files\Super Happy Fun Ball\Data, whereas Sally's go into the table in her virtualized directory. If a backup process is Vista-aware, meaning virtualization isn't used, backing up the tables in C:\Program Files\Super Happy Fun Ball\Data backs up files that are never written to, and the real tables aren't backed up at all.

Here's another problematic scenario with virtualization: The user initially runs the application with UAC turned on and their files are virtualized. If they then turn off UAC and run the application again, file access is no longer virtualized so the application accesses the files in the Program Files folder rather than ones in the virtualization folder. As a result, all their previous settings and data appear to be lost, since they're accessing the wrong files.

Thus, to truly work on Vista, it's better to revisit where your data is stored and use one of the Vista preferred locations rather than relying on virtualization. This doesn't mean you need to have one version of your application for Windows XP or earlier operating systems and one for Vista; although they have different paths in Vista, these preferred locations exist in earlier versions too. You can use Windows API functions to find the location of each "special folder," passing the appropriate ID value for the type of folder you want.

SpecialFolders.PRG is a wrapper for these functions. It returns the appropriate values regardless of whether the application is running in Vista or XP. See the comments in the header of this PRG for a description of what parameters to pass.

Here are some guidelines about which folders to use:

- Store read-only global data in the common application data folder. This folder has an ID of CSIDL_COMMON_APPDATA (0x23), which resolves to C:\Documents and Settings\All Users\Application Data on XP and C:\ProgramData on Vista. For example, the Super Happy Fun Ball application should store its global data in C:\ProgramData\My Company Name\Super Happy Fun Ball on Vista. Note that since this folder is read-only for standard users, it can only be used to store global settings created by an administrative user. (In fact, this folder doesn't even show up in Windows Explorer unless you have the "show hidden files and folders" setting turned on.)
- Store read-write global data in the Public folder; this is only Microsoft-recommended location that standard users can write to. There isn't a CSIDL value for this folder, but the environment variable PUBLIC points to it, so you can use GETENV('PUBLIC') to determine its location. By default, this variable contains C:\Users\Public in Windows Vista. The variable doesn't exist in Windows XP, so your code must handle the case where GETENV('PUBLIC') returns a blank value. Alternatively, you could use the parent of the folder specified by CSIDL_COMMON_DOCUMENTS, which gives C:\Documents and Settings\All Users\Documents on XP and C:\Users\Public\Documents on Vista, or CSIDL_COMMON_DESKTOP, which gives C:\Documents and Settings\All Users\Desktop on XP and C:\Users\Public\Desktop on Vista. Another alternative is to use the ALLUSERSPROFILE environment variable in Windows XP and PUBLIC in Vista:

  ```
  if os(3) < '6'
    lcPath = getenv('ALLUSERSPROFILE')
  else
    lcPath = getenv('PUBLIC')
  endif
  ```

  For example, the Super Happy Fun Ball application should store its read-write global data in C:\Users\Public\My Company Name\Super Happy Fun Ball on Vista.

- Alternatively, you can create your own folder in the root of the drive (for example, C:\Super Happy Fun Ball) to store application data. The problem with this approach is that users may get annoyed with every application creating a new root folder, making their hard drive messy.
- Store user-specific roaming data (copied to the server and back for every computer the user logs into if roaming profiles are used) in the user's application data folder. This folder is C:\Documents and Settings*username*\Application Data for XP and C:\Users*username*\AppData\Roaming for Vista. Its ID is CSIDL_APPDATA (0x1A). For example, the Super Happy Fun Ball application should store its user-specific data in C:\Users*username*\AppData\Roaming\My Company Name\Super Happy Fun Ball on Vista.
- Store local non-roaming data in the user's local application data folder. This folder's ID is CSIDL_LOCAL_APPDATA (0x1C). For XP, the path is C:\Documents and Settings*username*\Local Settings\Application Data for XP, while it's C:\Users*username*\AppData\Local for Vista. For example, the Super Happy Fun Ball application should store its user-specific local data in C:\Users*username*\AppData\Local\My Company Name\Super Happy Fun Ball on Vista.
- Although you shouldn't automatically store anything here, use the user's documents folder as the default for saving and opening "documents" (for example, Word documents generated by the application). Use CSIDL_PERSONAL (0x05) as the ID; this maps to C:\Documents and Settings*username*\My Documents on XP and C:\Users*username*\Documents on Vista.

You can get the ID for other special folders from http://msdn2.microsoft.com/en-us/library/bb762494.aspx.

As with previous versions of Windows, you can store per-user settings in the Registry in HKEY_CURRENT_USER (typically HKEY_CURRENT_USER\Software*Company Name**Application Name*). You can't store global settings that can be changed in HKEY_LOCAL_MACHINE anymore; either use the virtualized HKEY_CLASSES_ROOT\VirtualStore\Machine or store settings in a file in the common application data folder.

See VirtualizationTest.PRG that accompanies this document for a demonstration of folder and Registry virtualization.

You can turn off virtualization using the Local Security Policy editor. Expand *Local Policies*, click *Security Options*, double-click the *User Account Control: Virtualize file and registry write failures to per-user locations* policy, and change the setting to Disabled (**Figure 5**). After restarting your system, attempts to write to protected resources by applications running in XP compatibility mode fail; for example, MD C:\Program Files\Some Folder gives a "file access is denied" error in VFP. Also note that virtualization is automatically disabled if UAC is disabled. While I don't recommend turning off UAC or virtualization, it is a good idea to turn them off and test your application; this allows you to see where write failures occur so you can fix them.

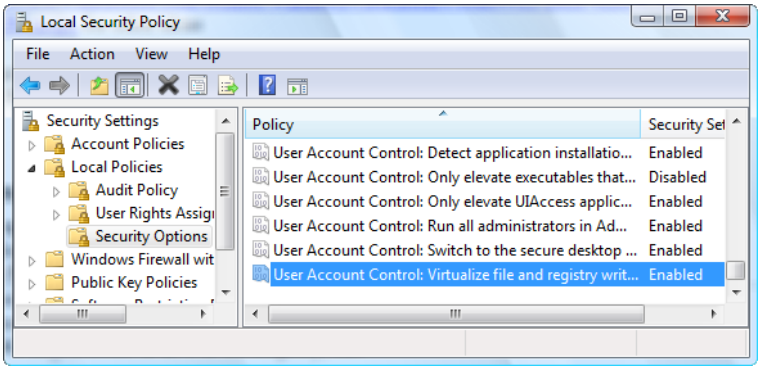

Figure 5. You can turn off virtualization using the Local Security Policy editor.

To flag that an application is Vista-aware and turn off XP compatibility mode, add a manifest file to the EXE's resources. The manifest file must have the following section:

```
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
  <security>
    <requestedPrivileges>
      <requestedExecutionLevel
        level="asInvoker"
        uiAccess="false"/>
    </requestedPrivileges>
  </security>
</trustInfo>
```

With this section in its manifest, Vista considers the application to be Vista-aware, which means it does not perform virtualization and automatically asks the user to elevate if the application requires administrator privileges. See the resources at the end of this document for resources discussing manifests.

Calvin Hsia, lead developer for Visual FoxPro, blogged about how to add a manifest to a VFP EXE at http://blogs.msdn.com/calvin_hsia/archive/2007/04/13/add-a-manifest-to-control-your-application-vista-uac-behavior.aspx.

Other Security Issues

While we were discussing Vista issues at the MVP Summit in March 2007, Christof Wollenhaupt pointed out that building a COM object in VFP may be an issue. Sure enough, when I tried to do this, VFP gave an error message because after building a COM DLL or EXE, it tries to register the COM object and it doesn't have permission. The workaround is to run VFP as an administrator when you need to do this.

One other issue I've run into: if you open any classes, forms, reports, etc. in the VFP home directory or any of its subdirectories (such as FFC or XSource), even if you don't save any changes, VFP updates the datetime stamp on the VCT file, which causes it to be virtualized.

Related to this is opening VFP tables in a protected folder. Even if you don't write to the table, VFP appears to update the table header, so immediately after the USE command, the table is virtualized. To avoid this, add the NOUPDATE clause to the USE command if the table isn't updated in the application.

If you need to determine whether your application is being run as administrator, use the IsUserAnAdmin API function. It returns 1 for true and 0 for false.

```
local llReturn
declare integer IsUserAnAdmin in shell32
return IsUserAnAdmin() = 1
```

If your application needs to run another application that requires elevation (for example, you want an application to be able to update itself by downloading a newer version then running a program that shuts down the current one and installs the new version), you can't just use RUN; that fails with an error. Instead, use the RunAs operation with ShellExecute; the user will then see the elevation dialog when the application runs.

```
declare integer ShellExecute in Shell32.DLL ;
  integer nWinHandle, ; && handle of parent window
  string cOperation, ;  && operation to perform
  string cFileName, ;   && filename
  string cParameters, ; && parameters for the executable
  string cDirectory, ;  && default directory
  integer nShowWindow   && window state
ShellExecute(0, 'RunAs', lcEXEFileName, lcParameters, lcDirectory, 1)
```

Installing Applications

Application installers have special requirements most other applications don't have: They usually need to copy files to protected folders, typically Program Files or the Windows System folder, and often need to update the Registry as well, such as when registering ActiveX controls and DLLs. Since these tasks require administrative privileges, they would normally fail unless the user specifically runs the installer as an administrative user.

To make this easier, Vista uses a new feature called Installer Detection. If Vista determines that an application is an installer, uninstaller, or application updater, it automatically displays the UAC dialog so the user can elevate. This is controlled through the *User Account Control: Detect application installations and prompt for elevation* setting in the Local Security Policy editor, which is normally turned on.

Some of the rules Vista uses during Installer Detection are:

- The filename contains words such as "setup," "install," or "update."
- Certain fields in the file's versioning resource or manifest contain specific installer-related keywords.
- The file contains a specific sequence of bytes known to occur in installer products.

Installer Detection means that, other than the UAC dialog, the user experience for installing an application is the same in Vista as it is in older Windows versions. However, from a developer perspective, UAC has significant implications for your setups.

A result of running as an administrative user is that user-specific tasks relate to that user rather than the standard user. For example, suppose the installer stores some settings in the HKEY_CURRENT_USER hive in the Registry. The problem is that it stores those settings for the administrative user, so once the installer is done and the user runs the application (now running it as the standard user), they can't see those settings. The same is true if the user launches the application from the installer, a feature many installers provide as a quick way to get the user started with the application. Since Vista is still elevated when it starts the application, the application runs as the administrative user. If the user specifies configuration settings and the application stores those settings in a user-specific location (the HKEY_CURRENT_USER hive in the Registry or a user-specific folder), the next time the user runs the application, they do so as the standard user and won't see those settings.

The solution is to not allow any user-specific tasks in your application's installer. This means:

- Don't write settings to HKEY_CURRENT_USER in the Registry. Some installers ask the user for certain configuration settings, such as the location of data files. Store those settings in HKEY_LOCAL_MACHINE or have the application itself prompt the user for these settings the first time it's run rather than doing it in the installer.
- Don't write to files in user-specific folders.
- Create icons for all users rather than the current user. For Inno Setup scripts, use {commondesktop}, which references the common desktop folder used by all users, rather than {userdesktop}, which is user-specific, as the location for desktop icons.

- Register all DLL and OCX files in the installer. Although it's not common, some applications dynamically register the DLLs and OCXs they use. That won't work under Vista because doing so requires administrator privileges.
- Don't allow the application to be launched from the installer. For Inno Setup scripts, add OnlyBelowVersion: 0,6 to the [Run] command so the option is only available for Windows XP or lower. Alternatively, use Inno Setup version 5.2 or later, which executes an application with the normally non-elevated credentials of the user that started the installer.

Another issue relates to registering DLLs and ActiveX controls. Some of the support files needed by VFP applications, including OLEAUT32.DLL and STDOLE2.TLB, come with Vista. In the case of Inno Setup, adding the "onlyifdoesntexist" flag ensures these files aren't overwritten. However, because of the "regserver" flag, Inno Setup will attempt to re-register these files. This causes an error on Vista. This isn't a problem because the files are already registered, but can be unnerving to users installing your application. There are two solutions to this: add the "noregerror" flag to prevent the installer from displaying an error message, or use Inno Setup version 5.1.11 or later, which uses a different registration process (see http://jrsoftware.org/files/is5-whatsnew.htm for details).

It's more difficult to automatically update your application. For example, some applications detect when a newer version is available and give the user the option to download and install the new version; for a description of how to do this and source code to implement it in your applications, see Rick Strahl's white paper "Automatic Application Updates over the Web" (http://www.west-wind.com/presentations/wwCodeUpdate/codeupdate.asp). Unfortunately, that doesn't work in Vista anymore because the application can't write to the Program Files folder so the new version can't be installed; it fails with an "access denied" error. Some of the solutions to this are to have the user run the application as administrator (right-click its icon and choose Run as Administrator) so it has the privileges necessary to write to Program Files or ask the user to manually download and install the update. Another is to name the updater program something like "Update.EXE" so Vista sees it as an installer and automatically asks the user to elevate.

If you want to store writable files into a subdirectory of your application folder (perhaps because you want to store them in a known fixed location), you'll need to create that folder in the installer and give all users write permissions to it. In Inno Setup, add something like this:

```
[Dirs]
Name: "{app}\Data"; Permissions: users-modify
```

Don't be tempted to use this to assign write permissions to the program folder itself or you've just opened a huge security hole on your client's system!

VFP 9 Service Pack 2 and Sedna

If you intend to run applications on Vista, there are two things you need: VFP 9 Service Pack 2 (SP2) and Sedna.

One of the goals of SP2 is Vista compatibility. You can run earlier versions of VFP, including VFP 8, on Vista and while they work correctly, some user interface issues make the application appear awkward. These issues are:

- As you move your mouse over an expanded combobox, every line you touch becomes highlighted (**Figure 6**). This makes it difficult to tell which one is selected. Calvin Hsia blogged the fix for this at http://blogs.msdn.com/calvin_hsia/archive/2007/04/26/foxpro-menu-items-combo-boxes-not-refreshing-selected-item-under-aero-in-vista.aspx; the solution is to execute this code one time somewhere in your application:

  ```
  declare integer GdiSetBatchLimit in Win32API integer
  GdiSetBatchLimit(1)
  ```

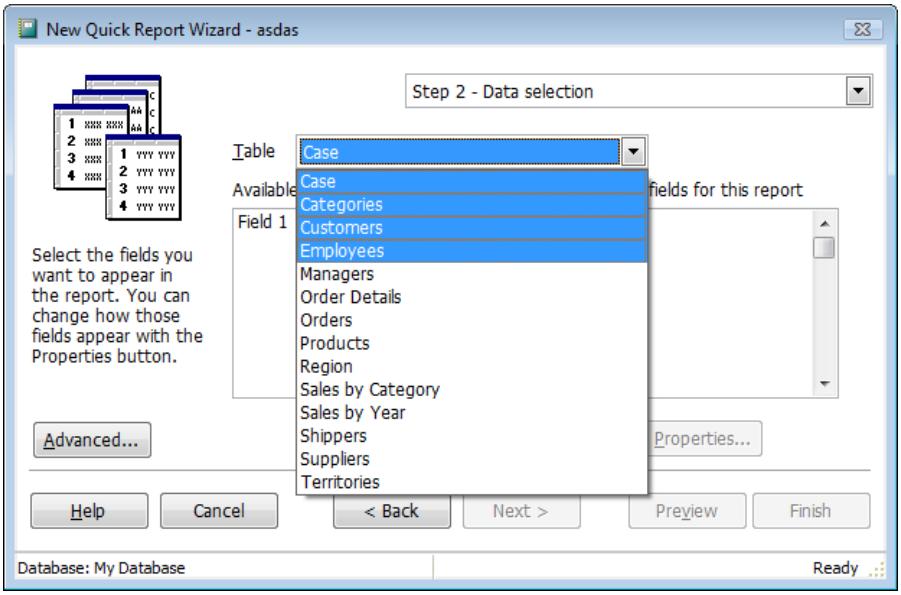

Figure 6. Every combobox row you move your mouse over becomes highlighted.

- Although I haven't encountered this, Rick Strahl blogged about a similar effect with shortcut menus at http://www.west-wind.com/wconnect/weblog/ShowEntry.blog?id=597. The fix for this is the same as it is for comboboxes.
- VFP incorrectly renders the borders of forms with BorderStyle set to something other than 3-Sizable. Sometimes the border displays correctly, sometimes part of it is missing, and sometimes it doesn't appear at all. If you click the Close box, you'll notice another close box appear until you release the mouse button. Dragging the form off-screen "smears" the border badly as you can see in **Figure 7**. Fortunately, there are workarounds for this:

Figure 7. Fixed-size dialogs have border issues.

  - Set BorderStyle to 3-Resizable and set the Anchor property of each control appropriately. This has the benefit that the form is now resizable. That isn't necessarily appropriate for each form, however.
  - Set BorderStyle to 3-Resizable and set MinHeight and MaxHeight to Height and MinWidth and MaxWidth to Width. This makes the form non-resizable since the minimum and maximum dimensions are the same. However, the cursor still turns into a resize icon when the user puts the mouse pointer on the form's edges, which may be confusing.
  - If the form is a child of _SCREEN, activate the window in the Init method:

```
if This.ShowWindow = 0
```

```
  activate window (This.Name) in screen noshow
endif This.ShowWindow = 0
```

The NOSHOW clause prevents the form from appearing before it's supposed to. If the form lives in a top-level form, specify the name of that form instead. Thanks to Tracy Pearson for this tip.

- Set Desktop to .T. The form will remain non-resizable but the border displays correctly. In fact, this has the benefit of making the form support Aero Glass; the border is semi-transparent and rounded rather than square, the control buttons "glow" when you hover the mouse pointer over them, and the window is shadowed. However, it also means the user can move the form outside the VFP main screen or your top-level form. I actually think that's an advantage for the type of applications I develop, but may not work for you. Also, if you use the MODIFY REPORT command in your application to allow the user to modify a report, the Report Designer window appears behind your application forms, which makes it very difficult to work with. **Figure 8** shows the same form as Figure 7 but with Desktop set to .T.

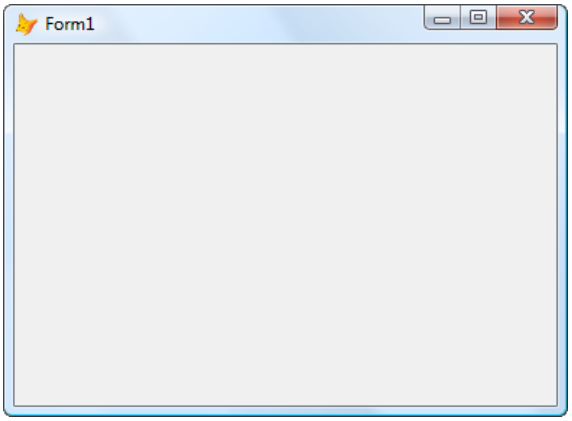

Figure 8. This is the same form as Figure 7 but with Desktop set to .T. It not only has the correct border, it now supports Aero Glass.

Microsoft is aware of these issues and has committed to fixing them in SP2.

One of the components of Sedna is the Vista Toolkit. This toolkit provides several classes that take advantage of new features in Vista, including the new task dialog and updated file open and save dialogs, Windows Desktop Search, and RSS feeds. I'll discuss this in detail later in this document.

Using Vista Features in VFP Applications

Taking advantage of Vista's new features makes your application more modern in appearance and behavior. I'll discuss four areas in which you can do this: user interface, Windows Desktop Search, RSS feeds, and XML Paper Specification.

User Interface

One of the first things you notice about Vista is that its user interface is much more attractive than older operating systems. While some decry this as "eye candy," having a great looking desktop makes using your computer more interesting and fun and hopefully makes you more productive, although that's tough to measure.

There are several things you can do with your VFP applications to make them more Vista-like, including selecting Vista fonts, using the new dialogs, and creating new icons.

Font Selection

Operating systems before Windows Vista use MS Sans Serif as the system font. MS Sans Serif isn't a TrueType or ClearType font, so it doesn't scale very well at different resolutions, making it difficult to create a form that looks good under all conditions. As a result, most VFP developers use the default font for VFP controls, Arial, or some other TrueType font (I prefer Tahoma), which means VFP forms look different than system dialogs.

Vista introduces a new system font: Segoe UI (pronounced "SEE-go"). It's a ClearType font designed specifically for user interfaces. As a result, it scales nicely and is the recommended font for all dialogs. For more information about Segoe UI, see http://msdn2.microsoft.com/en-us/library/aa511295.aspx.

As you can see in **Table 1**, Segoe UI's font characteristics are similar to Tahoma. Note that the average character width (FONTMETRIC(6)) for Segoe UI is slightly larger than Tahoma but the maximum width (FONTMETRIC(7)) is slightly smaller. As a result, strings tend to be roughly the same width in both fonts, and both are a little smaller than Arial text.

Table 1. The font characteristics of Segoe UI are similar to Tahoma.

Font Name	FONTMETRIC(6) 9 pt.	FONTMETRIC(7) 9 pt.	FONTMETRIC(6) 14 pt.	FONTMETRIC(7) 14 pt.
Tahoma	5	25	8	40
Segoe UI	6	22	10	36
Arial	5	32	8	51

The source code for this document includes FontTest.SCX (**Figure 9**) which shows the relative sizes of text in Segoe UI vs. Tahoma and Arial. Enter the text to test and the size, then click the Measure TXT button to measure the text size as it would appear in a form (using TXTWIDTH(Text, Font, Size) * FONTMETRIC(6, Text, Font, Size)) or the Measure GDI+ button to measure the text size as it would appear in a report (using the FFC GDI+ classes).

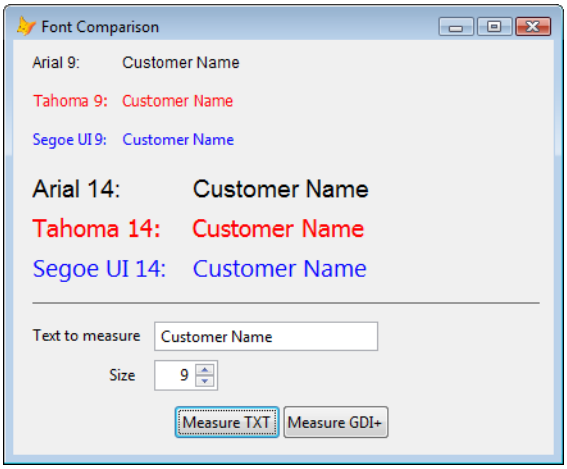

Figure 9. FontTest.SCX shows the relative sizes of text in Segoe UI vs. Tahoma and Arial.

Because your applications may run on both Windows Vista and older operating systems, you may wish to use Segoe UI when the application runs on Vista. To do this, change your base classes to select the font based on the operating system using code similar to this in the Init method:

```
if os(3) >= '6'
  This.FontName = 'Segoe UI'
endif os(3) >= '6'
```

Since text is roughly the same size in both Segoe UI and Tahoma, you don't have to worry about adjusting the width of the object, especially if the AutoSize property of the control (if it has one) is .T., or the space between objects, although you should test your forms on different operating systems to confirm that they look correct. **Figure 10** shows two copies of the same form, one using Tahoma and the other Segoe UI (albeit both on Vista) and you can see that object size and positioning is correct without making any adjustments.

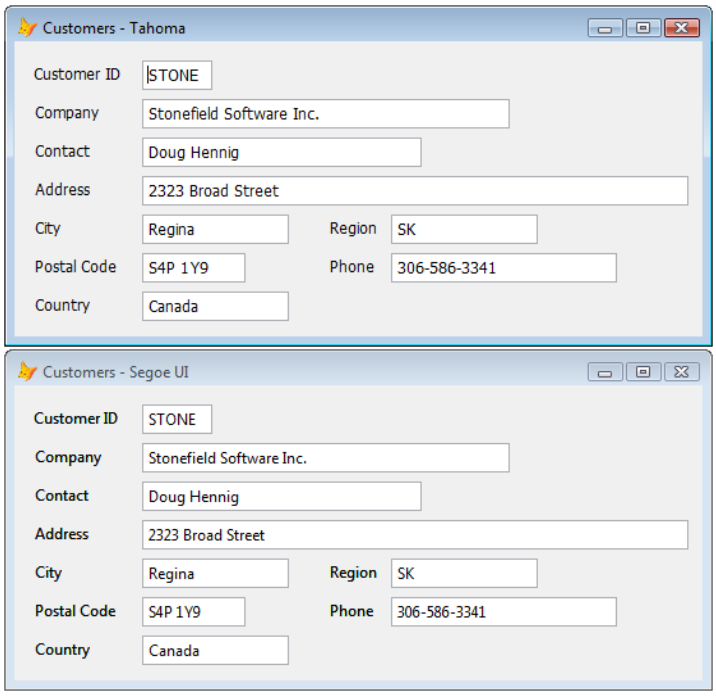

Figure 10. You can usually change from Tahoma to Segoe UI without resizing or moving objects.

Dialogs

Vista has new dialogs for common tasks such as opening and saving files, plus a replacement for the message box dialog. These dialogs have a lot more functionality and a more Vista-like appearance.

Figure 11 shows the dialog presented by the VFP GETFILE() function. Although this dialog does have a Recent Places button, it still looks like a dialog from an older operating system.

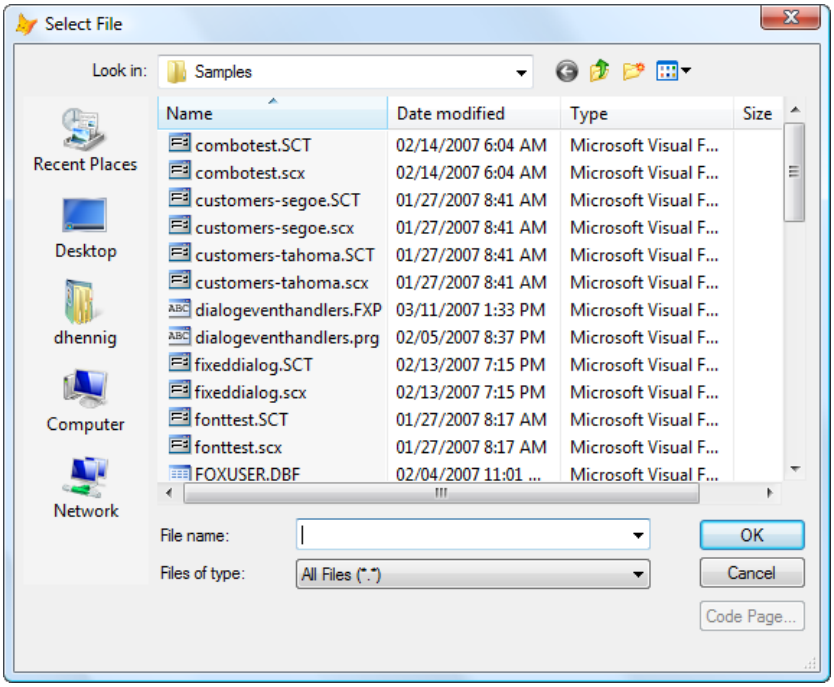

Figure 11. The VFP GETFILE() function is an older-looking dialog.

As you can see in **Figure 12**, the Vista open file dialog not only has a more modern interface, it has several features the older dialog doesn't, including back and forward buttons, the "breadcrumb" folder control, and access to Windows Search. The VFP PUTFILE() and Vista save file dialogs have a similar comparison.

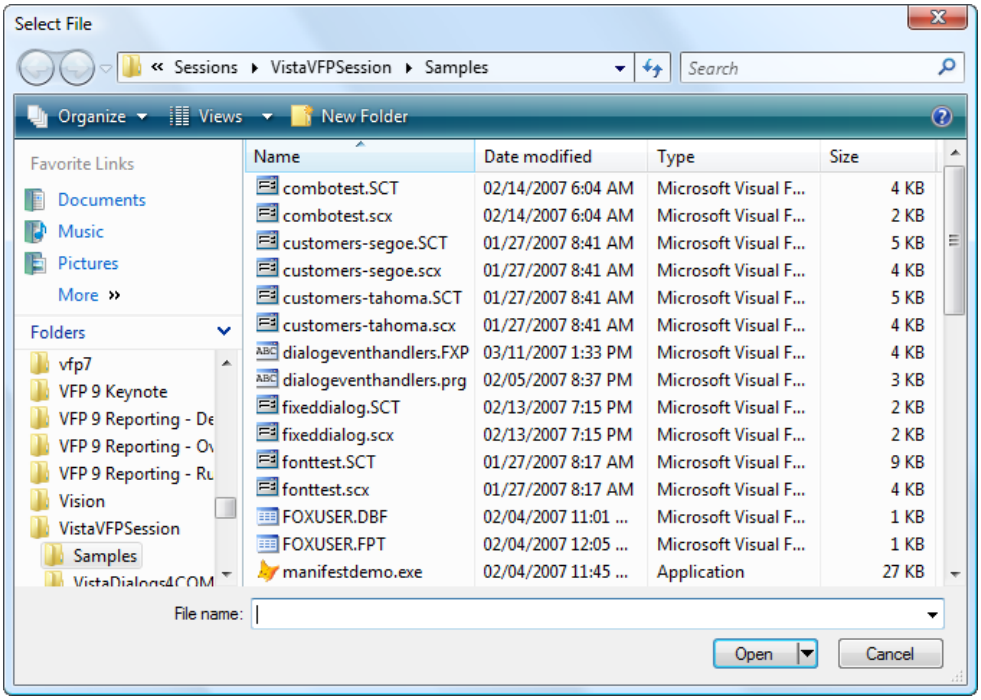

Figure 12. The Vista open file dialog looks modern and has features the older dialog doesn't.

Vista has a new task dialog that replaces the older message box dialog. This dialog has a more modern appearance plus is much more customizable: you can specify your own buttons; add controls such as command link buttons, radio buttons, checkboxes, and progress bars; include footers and collapsible detail text; and even react to events fired by the various controls. **Figure 13** shows an example of a task dialog.

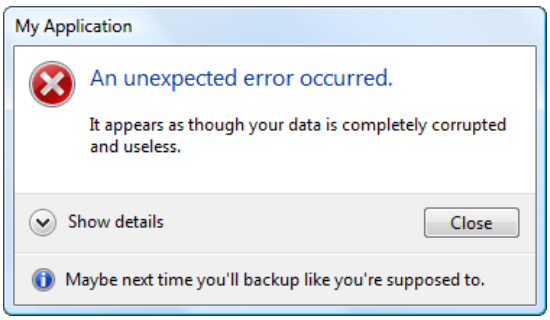

Figure 13. The Vista task dialog is much more customizable than the message box dialog.

The API for these dialogs is quite complex and difficult to use from VFP. Fortunately, MVP Craig Boyd has created a wrapper COM object written in .NET for them, making them easy to use in VFP. This object is part of the Vista Toolkit component in Sedna and can be downloaded as a Community Technology Preview (CTP) from the VFP home page (http://msdn2.microsoft.com/en-us/vfoxpro/default.aspx).

Here's some code, taken from OpenFileDialog.PRG that accompanies this document, showing how to display the open file dialog; the resulting dialog is shown in Figure 12.

```
local loOpenFileDialog, ;
  lcFileNames, ;
  lcFile
```

```
* Instantiate the open file dialog object and set its properties.

loOpenFileDialog = createobject('VistaDialogs4COM.CommonOpenFileDialog')
with loOpenFileDialog
  .CheckFileExists = .T.
    && ensure the selected file exists
  .CheckPathExists = .T.
    && ensure the selected folder exists
  .ShowPlacesList = .T.
    && populate the Favorite Links section
  .DereferenceLinks = .T.
    && specifies whether the dialog box returns the location of the file
    && referenced by the shortcut or whether it returns the location
    && of the shortcut (.lnk). Also, if a shortcut to a folder is
    && selected, that folder is opened in the dialog; otherwise, that
    && folder is returned
  .Title = 'Select File'
    && dialog caption; if omitted, "Open" is used

* Display the dialog and get the list of files the user selected if they didn't
* cancel.

  if not .ShowDialog()
    lcFileNames = ''
    for each lcFile in .FileNames
      lcFileNames = iif(empty(lcFileNames), '', chr(13)) + lcFileNames + lcFile
    next lcFile
    messagebox('You selected:' + chr(13) + lcFileNames)
  endif not .ShowDialog()
endwith
```

The following code, taken from TaskDialogSimple1.PRG, shows a simple example using the Vista task dialog. This example doesn't do much more than MESSAGEBOX() would, but it looks more modern (**Figure 14**).

```
#include VistaDialogs.H
local loTaskDialog

* Show a message box so we can compare to the task dialog.

messagebox('An unexpected error occurred: It appears as though your data ' + ;
  'is completely corrupted and useless.', 16, 'My Application')

* Instantiate the task dialog object and set some properties.

loTaskDialog = createobject('VistaDialogs4COM.TaskDialog')
with loTaskDialog
  .Caption         = 'My Application'
  .Instruction     = 'An unexpected error occurred.'
  .Content         = 'It appears as though your data is completely ' + ;
    'corrupted and useless.'
  .MainIcon        = TDERRORICON
  .StandardButtons = TDCLOSE

* Display the dialog.

  .Show()
endwith
```

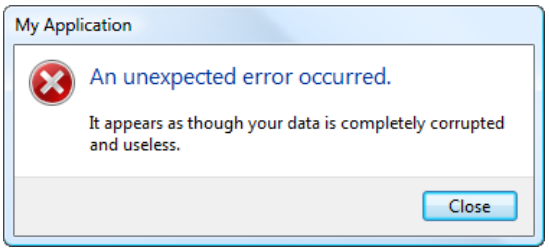

Figure 14. This simple instance of the task dialog has a more modern appearance than MESSAGEBOX() provides.

The next example, taken from TaskDialogSimple2.PRG, shows additional customization of the dialog. It creates the dialog shown in **Figure 13**.

```
#include VistaDialogs.H
local loTaskDialog

* Instantiate the task dialog object and set some properties.

loTaskDialog = createobject('VistaDialogs4COM.TaskDialog')
with loTaskDialog
  .Caption              = 'My Application'
  .Instruction          = 'An unexpected error occurred.'
  .Content              = 'It appears as though your data is ' + ;
    'completely corrupted and useless.'
  .ExpandedText         = "I would recommend you restore from a backup, " + ;
    "but I know you don't have one. You'll have to re-enter everything."
  .ExpandedControlText  = 'Hide details'
  .CollapsedControlText = 'Show details'
  .MainIcon             = TDERRORICON
  .StandardButtons      = TDCLOSE
  .FooterText           = "Maybe next time you'll backup like you're supposed to."
  .FooterIcon           = TDINFORMATIONICON

* Display the dialog.

  .Show()
endwith
```

The final example, TaskDialogAdvanced.PRG, shows how to add controls such as command links and progress bars to the dialog and how to bind to events. I'll discuss this code in sections. The first part creates the dialog and sets its properties.

```
#include VistaDialogs.H
local loTaskDialog, ;
  loTaskDialogEventHandler, ;
  loOurEventHandler, ;
  loButton, ;
  loCommandLinkEventHandler, ;
  loTaskDialogResult, ;
  lcResult

* Instantiate the task dialog object and set some properties.

loTaskDialog = createobject('VistaDialogs4COM.TaskDialog')
with loTaskDialog
  .Caption         = 'My Application'
  .Instruction     = 'A new version is available.'
  .Content         = 'For information on the features in this new ' + ;
    'release, see <a href="http://www.stonefieldquery.com">our Web ' + ;
    'site</a>.'
  .MainIcon        = TDINFORMATIONICON
  .StandardButtons = TDCLOSE
  .Cancelable      = .T.
```

You need an event handler to bind to COM events; the classes in DialogEventHandlers.PRG, which comes with the Vista Toolkit, provide event handlers for the different objects that may be used by the task dialog. You could add code directly to the events in these classes, but then you'd need to clone the PRG for every different dialog you want. Instead, create another object which acts as the event handler for the event handler; this object can be a custom object for each dialog. First, instantiate the generic TaskDialogEventHandler class in DialogEventHandlers.PRG and use it as the event handler for the task dialog. Then create a custom event handler. Later code binds events from the first object to methods of the second.

```
loTaskDialogEventHandler = newobject('TaskDialogEventHandler', ;
  'DialogEventHandlers.PRG')
eventhandler(loTaskDialog, loTaskDialogEventHandler)
loOurEventHandler = createobject('DialogEventHandler')
```

The task dialog supports hyperlinks in text. Notice in the code above the <a> tag in the Content property, with http://www.stonefieldquery.com as the target URL; if you're familiar with HTML, you'll recognize this as a hyperlink to a Web site. Clicking the hyperlink fires an event, so the following code enables hyperlinks and binds to the event.

```
.HyperlinksEnabled = .T.
bindevent(loTaskDialogEventHandler, 'ITaskDialogEvents_TDHyperlinkClick', ;
  loOurEventHandler, 'OnHyperlinkClicked')
```

Next create a command link button, add it to the task dialog, and bind to its click event.

```
loButton = createobject('VistaDialogs4COM.TaskDialogCommandLink')
loButton.Instruction = 'Download new version'
loTaskDialog.AddControl(loButton)
loCommandLinkEventHandler = newobject('TaskDialogCommandLinkEventHandler', ;
  'DialogEventHandlers.PRG')
eventhandler(loButton, loCommandLinkEventHandler)
bindevent(loCommandLinkEventHandler, 'ITaskDialogCommandLinkEvents_TDClick', ;
  loOurEventHandler, 'OnCommandLinkClicked')
```

The code then calls the Show method to display the dialog.

The DialogEventHandler class is based on Custom. Its OnHyperlinkClicked method, fired when the user clicks a hyperlink in the task dialog, navigates to the specified URL using the ShellExec API function.

```
function OnHyperlinkClicked(tcLinkText)
  declare integer ShellExecute in SHELL32.DLL ;
    integer nWinHandle, string cOperation, string cFileName, ;
    string cParameters, string cDirectory, integer nShowWindow
  ShellExecute(0, '', tcLinkText, '', '', 0)
endfunc
```

The resulting dialog is shown in **Figure 15**.

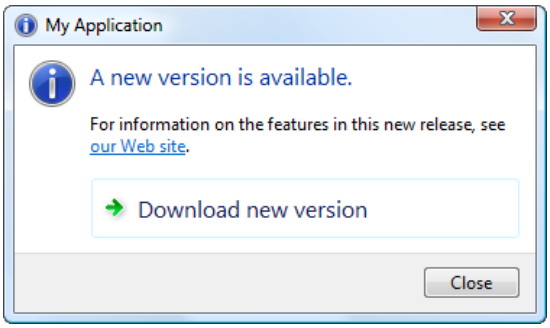

Figure 15. This dialog shows hyperlinks and a custom command link control.

As of this writing, the Vista Toolkit is incomplete; more functionality, such as being able to add radio buttons and other controls to the task dialog, is in the works.

The Vista Toolkit actually contains a number of other cool components, including MSFeeds, which provides access to the Windows RSS feed store, and WDS, which allows you to access Windows Desktop Search. However, neither of these actually require Vista, since the Windows RSS feed store is included with Internet Explorer 7 and Windows Desktop Search can be installed on Windows XP systems.

Icons

Although you can use older icons with Vista applications, you should change to Vista-like icons to make your applications more modern looking. As detailed at http://msdn.microsoft.com/library/en-us/UxGuide/UXGuide/Resources/WhatsnewInVista/Icons.asp, Vista icons come in a wider range of sizes than previously required. For example, desktop icons are now 48 x 48 pixels by default, but may need 256 x 256 versions as well. Also, ICO files now support the PNG format, which has many advantages over other formats, including better transparency support.

As a result, you need an icon editor capable of creating icons at the range of sizes needed by Vista. There are numerous icon editors available; I use Axialis IconWorkshop (http://www.axialis.com), which is easy to use and can automatically scale icons from 256 x 256 down to 16 x 16, including alpha channel (transparency) support.

Other UI Considerations

MSDN has several documents describing Vista user interface guidelines; "Windows Vista User Experience Guidelines" (http://msdn2.microsoft.com/en-us/library/aa511258.aspx) is the starting point, with links to other documents. These documents describe more than just the physical appearance of dialogs, but also when to use certain controls, the tone of text, and even how to make dialogs easier to use. I recommend reading and following the ideas in these documents to make your VFP applications more consistent with other Vista-aware applications.

Windows Desktop Search

Although it competes with Google Desktop Search and other search tools, Windows Desktop Search (WDS) is built into Vista and integrated with Windows Explorer and other dialogs. You can even install it on older operating systems. WDS is thorough—it searches files, contacts, emails, images, videos, etc.—and lightning fast. Even better, it comes with an OLE DB provider, so it's available to use in applications. The connection string for the WDS provider is "Provider=Search.CollatorDSO;Extended Properties='Application=Windows';."

Craig Boyd created a demo of WDS as part of the Vista Toolkit. You can use this demo as a test bench to try out various types of queries with WDS.

I stripped Craig's demo down and created a simple class for querying WDS called SFWDS (located in SFWDS.VCX). It has two custom properties: cConnectionString, which contains the OLE DB provider connection string, and cErrorMessage, which contains the text of any error message. It has one method, Search, responsible for performing a search and filling a cursor with the results. Pass it a comma-delimited list of fields and optionally a WHERE clause and cursor name. If you don't specify the cursor name, Search uses a SYS(2015) name. This code uses a CursorAdapter to perform the query and convert an ADO Recordset into a VFP cursor. One interesting (or weird) thing about the WDS provider is that it returns a lot more records than you'd expect, most of which have nulls in each field. Search takes care of that by removing all-null records.

```
lparameters tcFields, ;
  tcWhere, ;
  tcCursorName
local lnSelect, ;
  lcWhere, ;
  lcCursorName, ;
  loConnection as ADODB.Connection, ;
  loRecordset as ADODB.Recordset, ;
  lcTempCursor, ;
  loCursorAdapter as CursorAdapter, ;
  laFields[1], ;
  lnFields, ;
  lnI, ;
  llReturn, ;
  laError[1], ;
```

```
  loException as Exception

* Save the current workarea.

lnSelect = select()

* Handle the WHERE clause.

if vartype(tcWhere) = 'C' and not empty(tcWhere)
  lcWhere = ' where ' + tcWhere
else
  lcWhere = ''
endif vartype(tcWhere) = 'C' ...

* If a cursor name wasn't specified, create one.

if vartype(tcCursorName) = 'C' and not empty(tcCursorName)
  lcCursorName = tcCursorName
else
  lcCursorName = sys(2015)
endif vartype(tcCursorName) = 'C' ...
try

* Create and set up an ADO Connection object.

  loConnection = createobject('ADODB.Connection')
  loConnection.CursorLocation = 3  && adUseClient
  loConnection.Open(This.cConnectionString)

* Create an ADO Recordset object.

  loRecordSet = createobject('ADODB.Recordset')
  loRecordSet.ActiveConnection = loConnection

* Create a temporary cursor name.

  lcTempCursor = sys(2015)

* Use a CursorAdapter to perform the query.

  loCursorAdapter = createobject('CursorAdapter')
  with loCursorAdapter
    .Alias          = lcTempCursor
    .SelectCmd      = 'select ' + tcFields + ' from SystemIndex' + lcWhere
    .DataSourceType = 'ADO'
    .DataSource     = loRecordSet

* Fill CursorSchema because otherwise the fields come in too narrow.

    lnFields = alines(laFields, tcFields, 5, ',')
    for lnI = 1 to lnFields
      laFields[lnI] = strtran(laFields[lnI], '.', '_')
      .CursorSchema = .CursorSchema + ;
        iif(empty(.CursorSchema), '', ',') + laFields[lnI] + ' C(128)'
    next lnI

* If the query succeeds and we have some records, do a SELECT so we grab only
* valid records (the provider returns a lot of null records).

    if .CursorFill(.T.)
      llReturn = reccount() > 0
      if llReturn
        lcWhere = ''
        for lnI = 1 to lnFields
          lcWhere = lcWhere + iif(empty(lcWhere), '', ' or ') + ;
            'not isnull(' + laFields[lnI] + ')'
        next lnI
        select * from (lcTempCursor) where &lcWhere ;
```

```
          into cursor (lcCursorName)

* If we don't have any records, select the former workarea.

      else
        This.cErrorMessage = 'No matches found.'
        select (lnSelect)
      endif llReturn

* An error occurred, so get the error message and reselect the former workarea.

    else
      aerror(laError)
      This.cErrorMessage = laError[2]
      select (lnSelect)
      endif .CursorFill()
  endwith

* Clean up before we exit.

  loConnection.Close()

* An error occurred so get the error message.

catch to loException
  This.cErrorMessage = loException.Message
  select (lnSelect)
endtry
return llReturn
```

GetContacts.PRG creates a cursor of contact information from your Windows (not Outlook or other application) contacts.

```
lcFields = 'System.Contact.EmailAddress, System.Contact.FirstName, ' + ;
  'System.Contact.LastName, System.Contact.BusinessTelephone, ' + ;
  'System.Contact.HomeTelephone'
loSearch = newobject('SFWDS', 'SFWDS.VCX')
if loSearch.Search(lcFields, "System.Kind='contact'")
  browse
else
  messagebox(loSearch.cErrorMessage)
endif loSearch.Search(lcFields)
```

GetFiles.PRG locates DBF files in indexed locations (not necessarily the entire drive):

```
lcFields = 'System.FileName, System.ItemPathDisplay'
loSearch = newobject('SFWDS', 'SFWDS.VCX')
if loSearch.Search(lcFields, "System.FileExtension = '.dbf'")
  browse
else
  messagebox(loSearch.cErrorMessage)
endif loSearch.Search(lcFields ...
```

The Shell Properties document (http://msdn2.microsoft.com/en-us/library/ms788673.aspx) documents the properties you can query on.

RSS Feeds

Internet Explorer 7, which comes with Vista but can also be installed on other versions of Windows, provides a common RSS feed store and a COM object, Microsoft.FeedManager, which interfaces with it. The MS Feeds component of the Vista Toolkit includes a class library for accessing and managing the RSS feed store and a sample form showing how to create a simple RSS feed reader.

XML Paper Specification

Another new feature in Vista is the XML Paper Specification (XPS) document format (you can install XPS on older systems as well). Some people call this format a PDF killer because its intention is to provide the same type of system-independent viewable document but using a standard-based approach. I'm not so sure about whether it'll kill PDF or not, but Microsoft does provide a printer driver that makes it easy to generate an XPS document from any printable source.

To generate an XPS document from a VFP application, simply SET PRINTER TO NAME "Microsoft XPS Document Writer" before running a report or other print job. The driver prompts the user for the filename; I haven't found a programmatic way to set it yet.

You can read more about XPS at http://msdn.microsoft.com/msdnmag/issues/06/01/XMLPaperSpecification/default.aspx.

Resources

Lots of resources are available for learning more about Windows Vista security, user interface, and other features. Here are some of the resources I've used:

- The Advantages of Running Applications on Windows Vista (http://msdn.microsoft.com/en-us/library/bb188739.aspx)
- The Windows Vista Developer Story: Windows Vista Application Development Requirements for User Account Control (http://msdn2.microsoft.com/en-us/library/aa905330.aspx) or the more complete Windows Vista Application Development Requirements for User Account Control Compatibility (http://download.microsoft.com/download/5/6/a/56a0ed11-e073-42f9-932b-38acd478f46d/WindowsVistaUACDevReqs.doc)
- Getting Started with User Account Control in Windows Vista Beta 2 (http://msdn.microsoft.com/en-us/library/bb188740.aspx)
- The Windows Vista Developer Story: Application Compatibility Cookbook (http://msdn.microsoft.com/en-us/library/aa480152.aspx)
- Understanding and Configuring User Account Control in Windows Vista (http://www.microsoft.com/technet/WindowsVista/library/00d04415-2b2f-422c-b70e-b18ff918c281.mspx)
- Windows Vista User Experience Guidelines (http://msdn2.microsoft.com/en-us/library/aa511258.aspx)
- Registry Virtualization (http://msdn2.microsoft.com/en-us/library/aa965884.aspx)
- System Font (Segoe UI) (http://msdn2.microsoft.com/en-us/library/aa511295.aspx)
- Icons (http://msdn.microsoft.com/library/en-us/UxGuide/UXGuide/Resources/WhatsnewInVista/Icons.asp)
- Shell Properties (http://msdn2.microsoft.com/en-us/library/ms788673.aspx)
- A First Look at APIs For Creating XML Paper Specification Documents (http://msdn.microsoft.com/msdnmag/issues/06/01/XMLPaperSpecification/default.aspx)

Hasta la Vista, Baby

Whether you're planning to use Windows Vista yourself or not, you have to prepare for your customers who do. Go over your applications and ensure that files you write to are stored in accessible locations rather than the Program Files folder, use the HKEY_CURRENT_USER hive in the Registry rather than HKEY_LOCAL_MACHINE, adjust your installer to handle the Vista issues outlined in this document, and take advantage of some of the new features in Vista, including new dialogs and fonts, Windows Desktop Search, RSS feeds, and XPS documents. This will give your application a more modern look and a longer shelf life.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), the award-winning Stonefield Query, and 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 the "What's New in Visual FoxPro" series (the latest being "What's New in Nine") 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). Doug wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor and Advisor Guide. He has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is one of the administrators for the VFPX VFP community extensions Web site (http://www.codeplex.com/VFPX). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://fox.wikis.com/wc.dll?Wiki~FoxProCommunityLifetimeAchievementAward~VFP).