Developing VFP Applications for Windows 7

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

Like Windows Vista before it, Windows 7 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 Windows 7-compatible applications with Visual FoxPro.

Introduction

Windows 7 is Microsoft's latest version of its venerable desktop operating system. Whether you're using Windows 7 yourself or not, it's likely your customers are, whether by upgrading an existing system or buying a new one with Windows 7 pre-installed. Because of some new features, including improved security and a revamped user interface, 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 Windows 7-compatible VFP applications. We'll first look at how Windows 7 impacts applications in general and VFP applications in particular, especially when it comes to security, folder and Registry virtualization, and deployment. We'll then go over opportunities to take advantage of Windows 7 features, including user interface improvements and taskbar functionality.

Many of the Windows features discussed in this document actually started with Vista. When I refer to Windows 7, the features discussed relate to Vista as well except as noted.

Security

One of the biggest changes Microsoft made in Windows Vista that affects applications is related to security. Although Windows 7 eases restrictions a bit, there's still a big impact on applications designed when earlier versions of Windows were in use. 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 2000 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 7 makes running as a standard user much easier through a feature called User Access Control, or UAC.

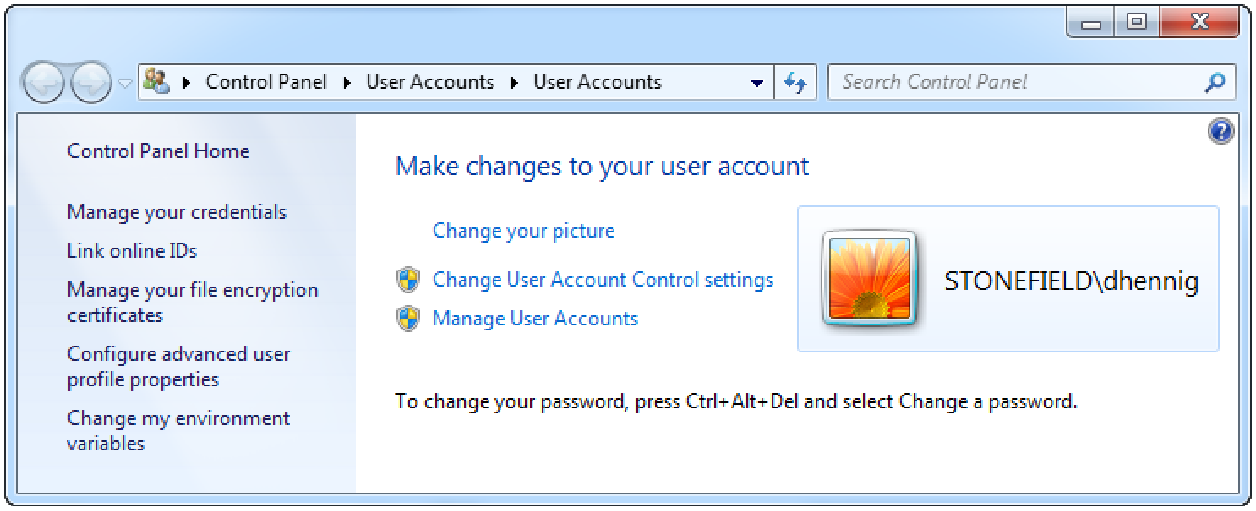
User Access Control

User access control starts with the login process. In versions of Windows prior to Vista, 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.

Starting with Vista, Windows now 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 2 for SeShutdownPrivilege (1 in Vista), 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 Windows 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, Windows automatically adds the shield to the icon for the program.


Control Panel ▸ User Accounts ▸ User Accounts
Search Control Panel
Control Panel Home
Manage your credentials
Link online IDs
Manage your file encryption certificates
Configure advanced user profile properties
Change my environment variables
Change your picture
Change User Account Control settings
Manage User Accounts

Make changes to your user account

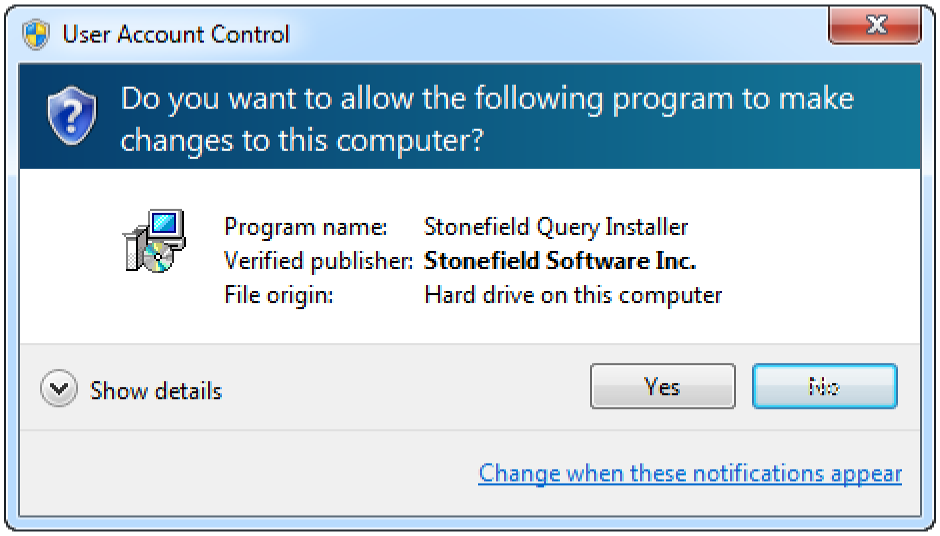
STONEFIELD\dhennig

To change your password, press Ctrl+Alt+Del and select Change a password.

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, Windows launches the process with the administrator security token, so it has full administrative access to the system.

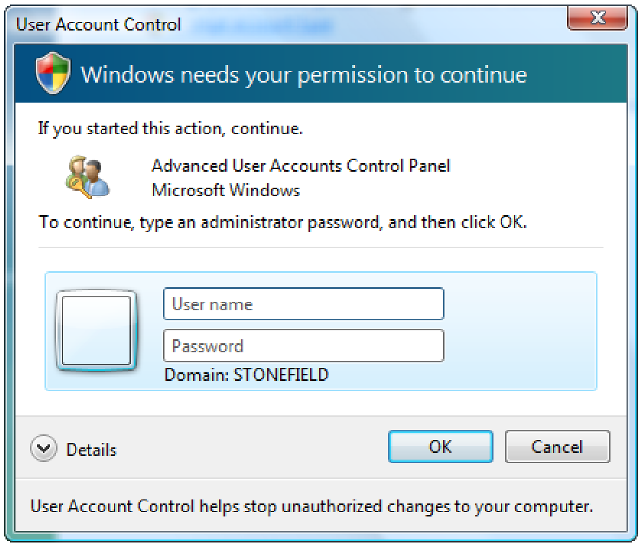


Do you want to allow the following program to make changes to this computer?

Program name: Stonefield Query Installer
Verified publisher: **Stonefield Software Inc.**
File origin: Hard drive on this computer

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, Windows runs the process as the administrative user, so again it has full administrative access to the system.


User Account Control

Windows needs your permission to continue

If you started this action, continue.

Advanced User Accounts Control Panel
Microsoft Windows

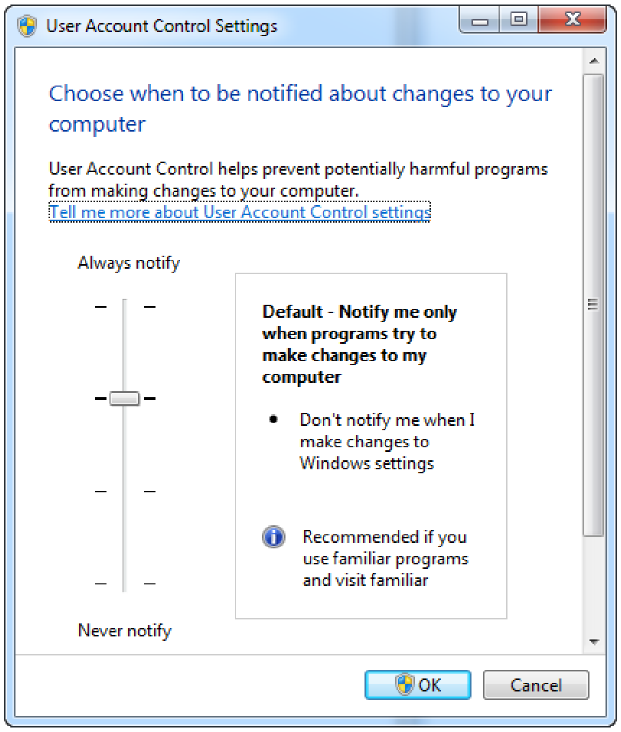
To continue, type an administrator password, and then click OK.

User name
Password
Domain: STONEFIELD

User Account Control helps stop unauthorized changes to your computer.

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.

Windows 7 makes this process easier by allowing you to control under what conditions the UAC dialogs appear. Clicking "Change User Account Control settings" in the User Accounts dialog shown in **Figure 1** displays the User Account Control Settings dialog (**Figure 4**). The default setting doesn't display the UAC dialog when you make changes to Windows settings, only when programs do. The default in Vista, which you can't easily change, is to always notify you.


User Account Control Settings

Choose when to be notified about changes to your computer

User Account Control helps prevent potentially harmful programs from making changes to your computer.
Tell me more about User Account Control settings

Default - Notify me only when programs try to make changes to my computer

- Don't notify me when I make changes to Windows settings

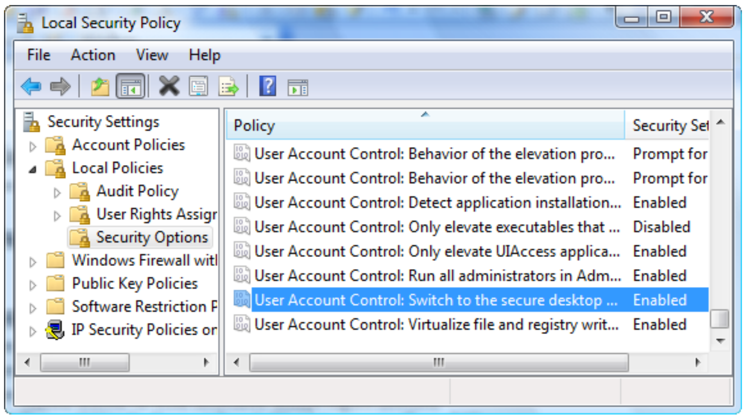
Recommended if you use familiar programs and visit familiar

Figure 4. The User Account Control settings dialog allows you to specify how intrusive UAC is.

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

Note that 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 dialogs for documentation purposes, like I did in this document. To do so, bring up the Local Security Policy editor (**Figure 5**). A fast way to do that is to click Start, type "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. 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.



Security Settings
- Account Policies
- Local Policies
  - Audit Policy
  - User Rights Assigr
  - Security Options
- Windows Firewall witl
- Public Key Policies
- Software Restriction P
- IP Security Policies or

Policy	Security Set
User Account Control: Behavior of the elevation pro...	Prompt for
User Account Control: Behavior of the elevation pro...	Prompt for
User Account Control: Detect application installation...	Enabled
User Account Control: Only elevate executables that ...	Disabled
User Account Control: Only elevate UIAccess applica...	Enabled
User Account Control: Run all administrators in Adm...	Enabled
User Account Control: Switch to the secure desktop ...	Enabled
User Account Control: Virtualize file and registry writ...	Enabled

Figure 5. You can disable the secure desktop using the Local Security Policy editor.

You may see the UAC dialogs a lot, depending on how you use your computer, although you see them a lot less in Windows 7 than in Vista. For example, they appear every time you run a program installer requiring administrative privileges (most do). Some people find this annoying, but I don't 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 "Never notify" in the User Account Control Settings dialog.

Windows makes it easy to set up standard users. In fact, all user accounts except the first one (which is created automatically when you install Windows) 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 starting with Vista. Many developers simply write to data files stored in the application folder (usually C:\Program Files\Some Folder) or 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*, Windows 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. (Note that on a 64-bit system, there's a subnode called Wow6432Node.)

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 and Windows Server 2008 Developer Story: Windows Vista Application Development Requirements for User Account Control" (http://tinyurl.com/2tlscg), "Developers must not rely on virtualization being present in subsequent versions of Windows." The most important reason, though, is that Windows 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 Settings.DBF 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 7 system and does the same thing. Thanks to virtualization, Windows writes the changes he makes to Settings.DBF 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 Settings.DBF in C:\Users\Sally\AppData\Local\VirtualStore\Program Files\Super Happy Fun Ball, so she doesn't use Bob's settings.

Because some types of VFP files consist of more than one physical file (for example, a table may consist of DBF, FPT, and CDX files), virtualization can cause the files to get out of sync. For example, if you write to a table in a protected directory, it's possible that the DBF is updated, and therefore virtualized, without the CDX and FPT being virtualized.

What about backups? If your backup program was written for Windows 7, 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, stored in virtualization folders, 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, it's better to revisit where your data is stored and use one of Windows' preferred locations rather than relying on virtualization. This doesn't mean you need to have one version of your application for Windows XP and one for Windows 7; although they have different paths in newer versions of Windows, 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 the version of Windows. 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 Windows 7. For example, the Super Happy Fun Ball application should store its global data in C:\ProgramData\My Company Name\Super Happy Fun Ball on Windows 7. 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.) However, you can set permissions for this folder so it is writable by all users.
- 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 7. 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 Windows 7. 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 Windows 7.
- 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

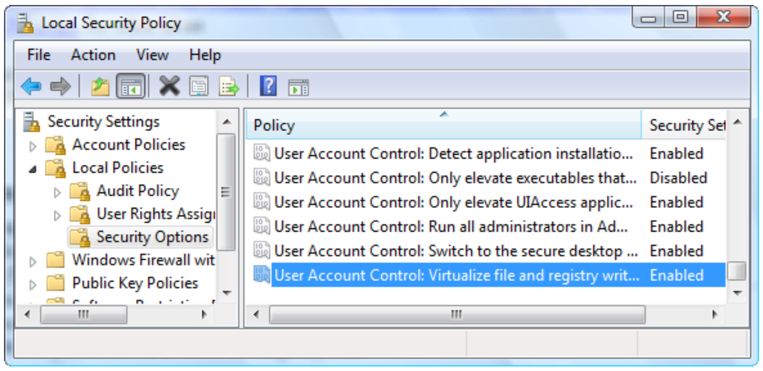
is that users may get annoyed with every application creating a new root folder, making their hard drive messy. A better solution is to store data in a subdirectory (such as Data) of your application's folder and set the permissions for that subdirectory so it's writable. I'll discuss how to do that in the "Installing Application" section of this document.

- 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 Windows 7. 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 Windows 7.
- 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, while it's C:\Users\username\AppData\Local for Windows 7. 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 Windows 7.
- 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 Windows 7.

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 6**). 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.



Policy	Security Set
User Account Control: Detect application installatio...	Enabled
User Account Control: Only elevate executables that...	Disabled
User Account Control: Only elevate UIAccess applic...	Enabled
User Account Control: Run all administrators in Ad...	Enabled
User Account Control: Switch to the secure desktop ...	Enabled
User Account Control: Virtualize file and registry writ...	Enabled

Figure 6. You can turn off virtualization using the Local Security Policy editor.

To flag that an application is Windows 7-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, Windows 7 considers the application to be Windows 7-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, former lead developer for Visual FoxPro, blogged about how to add a manifest to a VFP EXE at http://tinyurl.com/ypsgxe.

Other security issues

Building a COM object in VFP is an issue: VFP gives 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, Windows 7 uses a feature called Installer Detection. If Windows 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 Windows 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 Windows 7 as it is in XP and earlier 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 Windows 7 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 Windows 7 because doing so requires administrator privileges.
- Don't allow the application to be launched from the installer if the installer doesn't support executing the application with the normally non-elevated credentials of the user that started the installer. Inno Setup 5.2 and later supports this, but for earlier versions, add OnlyBelowVersion: 0,6 to the [Run] command so the option is only available for Windows XP or lower.

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 Windows 7. 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 Windows 7. 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://tinyurl.com/6bhd8l8 for details).

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: everyone-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!

Automatically updating applications

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://tinyurl.com/3grdvt7). Unfortunately, that doesn't work in Windows 7 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. One solution to this is 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. However, I think that's asking a lot of the user.

Here's the mechanism we use in Stonefield Query:

- A function in the application checks whether a newer version is available by downloading a file containing version information from our web server. We use Rick Strahl's wwFTP for file download, but you could also use Craig Boyd's free VFPConnection.FLL (http://tinyurl.com/333d4b5).
- If a newer version is available, we inform the user and ask them if they want to download it. If so, we download it to a writable folder. We use a subdirectory of the user's temporary files folder, which you can get from SYS(2023).
- Since you can't write to the folder the application is running from without administrative rights and even if you could, you can't overwrite the running EXE, you can't copy the updated version to the program folder. Instead, we launch another application (SQUpdate.EXE) using "RunAs" with ShellExecute as I mentioned earlier. That brings up the UAC dialog, so before doing so, we display a message box informing the user what's about to happen if they're running in Windows Vista or later (OS(3) returns "6" or greater). After using ShellExecute, we terminate the current application so it isn't running anymore.
- The launched application copies the update files into the program folder. It can do that because it has administrative rights and there's no problem overwriting the other files since they aren't running. Obviously, you can't use this mechanism to overwrite SQUpdate.EXE itself or the VFP runtimes, since they're in use, but any other files can be updated.
- After the copying is done, SQUpdate.EXE launches the main application and terminates itself.

The effect of this mechanism is that when the user checks for a newer version, it's downloaded, the main application closes for a moment, and then the new version opens.

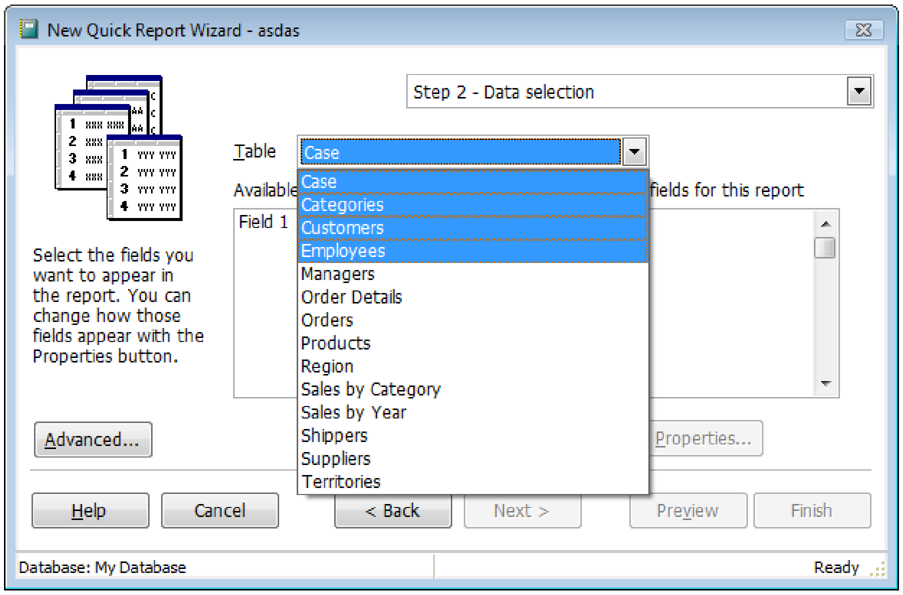
The nice thing about this mechanism is that it works on older and newer versions of Windows, doesn't require the user to manually download and install anything, and doesn't require the application to be run with administrative rights like I've seen other applications require.

VFP 9 Service Pack 2

If you intend to run applications on Windows 7, you should use VFP 9 Service Pack 2 (SP2). One of the goals of SP2 is compatibility with Vista and later versions of Windows. You can run earlier versions of VFP, including VFP 8, on Windows 7 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 7**). This makes it difficult to tell which one is selected. Calvin Hsia blogged the fix for this at http://tinyurl.com/34e8o7; the solution is to execute this code one time somewhere in your application:

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



Select the fields you want to appear in the report. You can change how those fields appear with the Properties button.

Figure 7. 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://tinyurl.com/6yvs9va. 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 8**. Fortunately, there are workarounds for this:

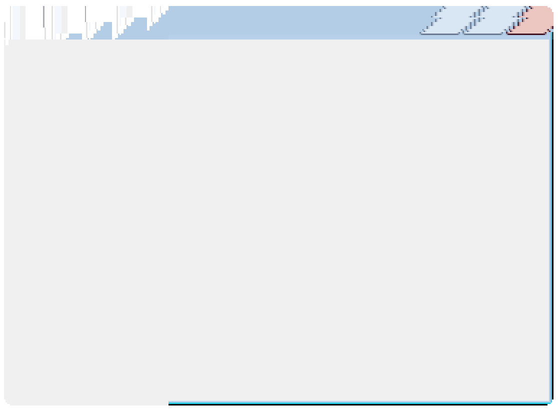


Figure 8. Fixed-size dialogs have border issues.

- 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. 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 9** shows the same form as **Figure 8** but with Desktop set to .T.

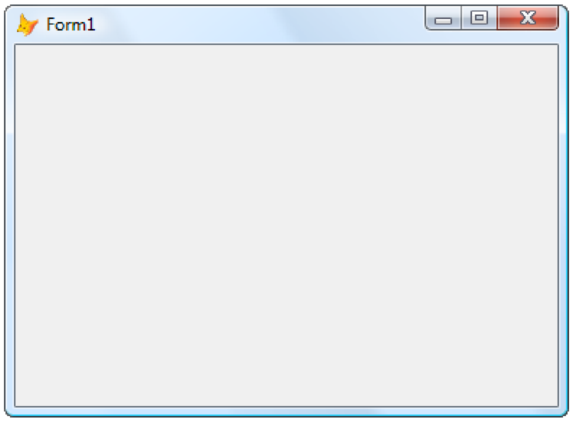


Figure 9. This is the same form as **Figure 8** but with Desktop set to .T. It not only has the correct border, it now supports Aero Glass.

Microsoft fixed these issues in SP2 so there's no need for workarounds.

Using Windows 7 features in VFP applications

One of the first things you notice about Windows 7 is that its user interface is much more attractive than XP and 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 Windows 7-like, including selecting the proper fonts, supporting Aero, creating new icons, using the new dialogs, and using the Windows 7 taskbar.

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.

Windows 7 uses a much nicer 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://tinyurl.com/5txp49c.

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 10**) 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).

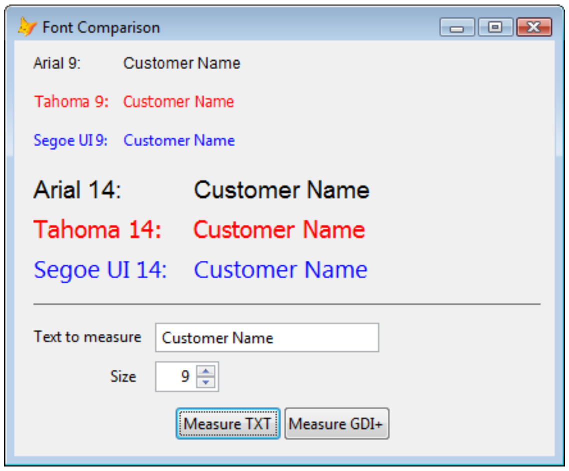
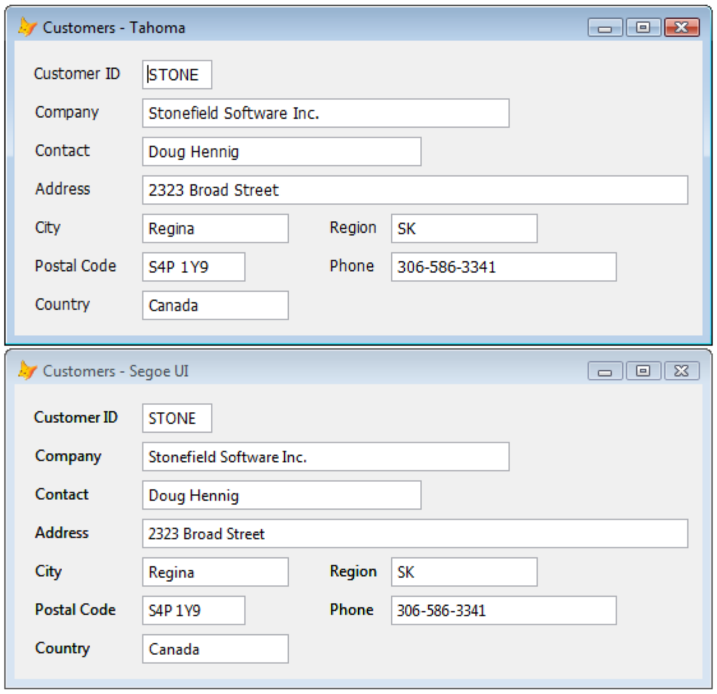


Figure 10. FontTest.SCX shows the relative sizes of text in Segoe UI vs. Tahoma and Arial.

Because your applications may run on both newer and older operating systems, you may wish to use Segoe UI when the application runs on Vista and later. 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 11** shows two copies of the same form, one using Tahoma and the other Segoe UI (albeit both on Windows 7) and you can see that object size and positioning is correct without making any adjustments.



Customer ID	STONE		
Company	Stonefield Software Inc.		
Contact	Doug Hennig		
Address	2323 Broad Street		
City	Regina	Region	SK
Postal Code	S4P 1Y9	Phone	306-586-3341
Country	Canada		

Customer ID	STONE		
Company	Stonefield Software Inc.		
Contact	Doug Hennig		
Address	2323 Broad Street		
City	Regina	Region	SK
Postal Code	S4P 1Y9	Phone	306-586-3341
Country	Canada		

Figure 11. You can usually change from Tahoma to Segoe UI without resizing or moving objects.

Aero support

Like Vista before it, Windows 7 provides a theme named Aero that provides transparent, rounded window borders, window shadows, title bar buttons that "glow" when you hover the mouse pointer over them, and other nice visual effects. By default, VFP forms don't support Aero, but you can set the Desktop property to .T. to turn on this support. 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.

Icons

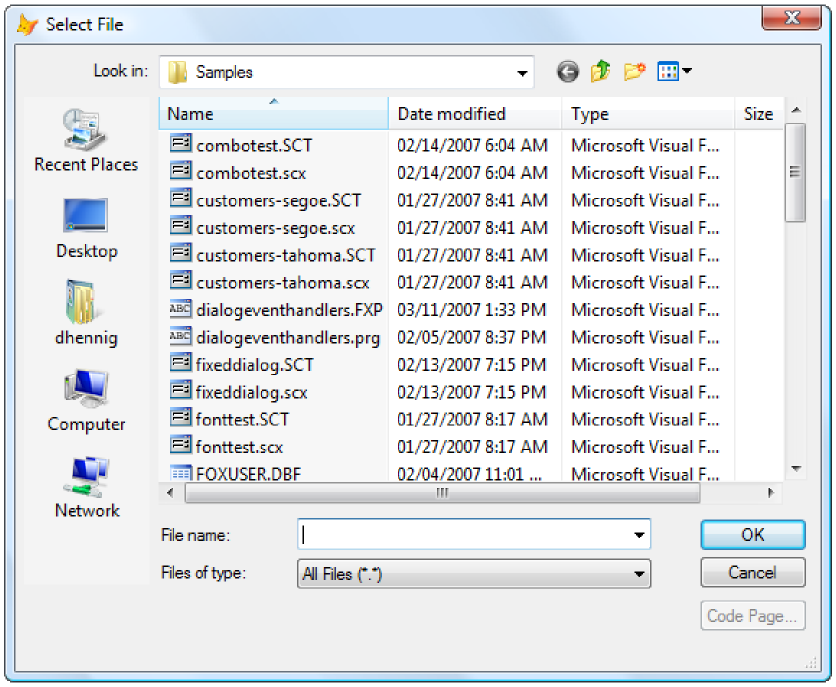
Although you can use older icons with Windows 7 applications, you should change to newer icons to make your applications more modern looking. As detailed at http://tinyurl.com/4f39g23, Windows 7 icons come in a wider range of sizes than previously required. For example, desktop icons are now 48 x 48 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 Windows 7. 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.

File dialogs

Windows Vista and later have new dialogs for common tasks such as opening and saving files. These dialogs have a lot more functionality and a more attractive appearance.

Figure 12 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.



Look in: Samples

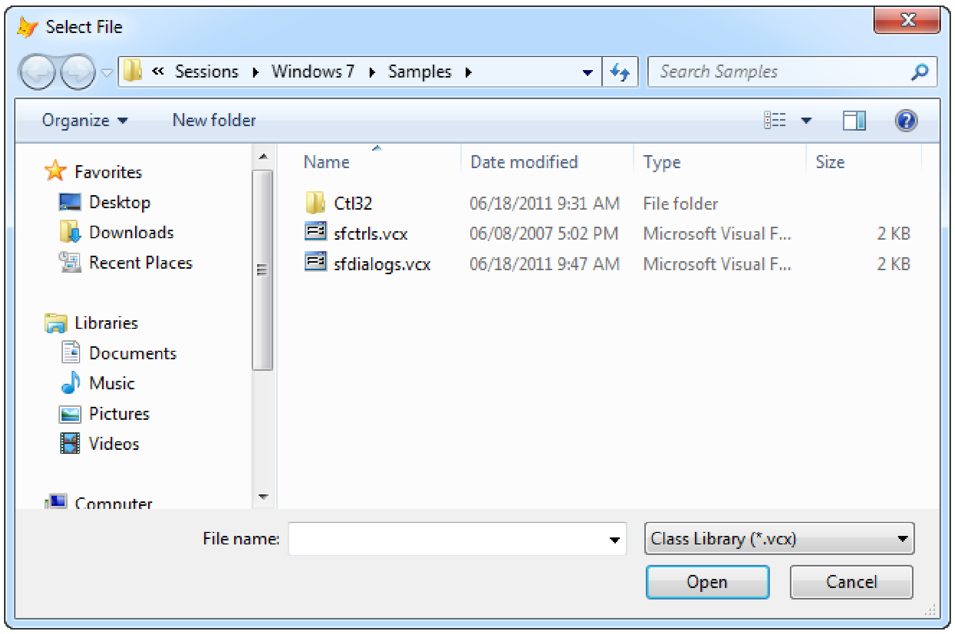
Name	Date modified	Type	Size
combotest.SCT	02/14/2007 6:04 AM	Microsoft Visual F...	
combotest.scx	02/14/2007 6:04 AM	Microsoft Visual F...	
customers-segoe.SCT	01/27/2007 8:41 AM	Microsoft Visual F...	
customers-segoe.scx	01/27/2007 8:41 AM	Microsoft Visual F...	
customers-tahoma.SCT	01/27/2007 8:41 AM	Microsoft Visual F...	
customers-tahoma.scx	01/27/2007 8:41 AM	Microsoft Visual F...	
dialogeventhandlers.FXP	03/11/2007 1:33 PM	Microsoft Visual F...	
dialogeventhandlers.prg	02/05/2007 8:37 PM	Microsoft Visual F...	
fixeddialog.SCT	02/13/2007 7:15 PM	Microsoft Visual F...	
fixeddialog.scx	02/13/2007 7:15 PM	Microsoft Visual F...	
fonttest.SCT	01/27/2007 8:17 AM	Microsoft Visual F...	
fonttest.scx	01/27/2007 8:17 AM	Microsoft Visual F...	
FOXUSER.DBF	02/04/2007 11:01 ...	Microsoft Visual F...	

File name:

Files of type: All Files (*.*)

Figure 12. The VFP GETFILE() function is an older-looking dialog.

As you can see in **Figure 13**, the Windows 7 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 Windows 7 save file dialogs have a similar comparison.



Name	Date modified	Type	Size
Ctl32	06/18/2011 9:31 AM	File folder	
sfctrls.vcx	06/08/2007 5:02 PM	Microsoft Visual F...	2 KB
sfdialogs.vcx	06/18/2011 9:47 AM	Microsoft Visual F...	2 KB

Figure 13. The Windows 7 open file dialog looks modern and has features the older dialog doesn't.

An easy way to take advantage of the newer dialogs in Windows 7 is to use the ctl32 library, created by Carlos Alloatti of Argentina. ctl32 is a library of 22 controls that provide modern versions of some of the functions built into VFP or available as VFP or ActiveX controls. Included in the library are replacements for the VFP GETFILE(), PUTFILE(), GETPICT(), and GETDIR() functions that use the latest dialogs available to your operating system, meaning they work correctly whether you're using Windows XP or Windows 7.

Download the ctl32 library from http://www.ctl32.com.ar and unzip it in some folder. Add the following files added to your project:

- ctl32.prg
- ctl32.vct and ctl32.vcx
- ctl32_api.prg
- ctl32_classes.prg
- ctl32_functions.prg
- ctl32_structures.prg
- ctl32_vfp2c32.prg
- vfpx.vct and vfpx.vcx

These files add about 1.5MB to your executable. That may seem like a lot, but once you start working with ctl32, you'll likely use many of the controls and replace ActiveX controls you formerly had to include with your application.

The classes of interest here are ctl32_OpenFileDialog, ctl32_SaveFileDialog, ctl32_OpenPictDialog, and ctl32_FolderBrowserDialog.

Rather than using ctl32_OpenFileDialog and ctl32_SaveFileDialog directly, I created a wrapper class for them called SFCommonDialog in SFDialogs.VCX. This class has the public properties shown in **Table 2**.

Table 2. Properties of SFCommonDialog.

Property	Description
aFileNames[1]	An array of filenames returned from the dialog
cDefaultExtension	The default file extension to display
cFileName	The name of file selected or initially set as default
cFilePath	The path files were selected from
cFileTitle	The file title property of the selected file(s)
cInitialDirectory	The initial directory to show files from
cTitlebarText	The caption for dialog title bar
lAllowMultiSelect	.T. to allow selection of multiple files
lFileMustExist	.T. if the file must exist
lHideReadOnly	.T. to hide read-only files from list
lNewExplorer	.T. to use the “new” explorer user interface and features such as the Places bar
lNoPlacesBar	.T. to not include Places bar in the dialog
lNoValidate	.T. to not validate the file name
lOverwritePrompt	.T. to prompt to overwrite if the user selects an existing file
lPathMustExist	.T. if the path must exist
lSaveDialog	.T. to use the Save dialog instead of the Open one
nFileCount	The number of files selected from the dialog
nFilterIndex	Specifies which of the filters the user selected from the dialog or the default filter to use

SFCommonDialog has three methods: AddFilter(cDescription, cFilter), which adds file type filters, ClearFilters(lClearAll), which clears the file type filters, and ShowDialog, which displays the dialog and returns .T. if the user clicked OK.

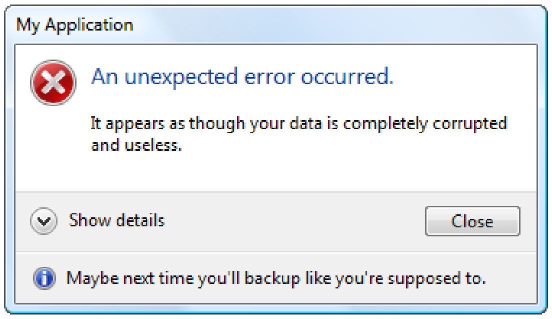
Here’s an example, taken from Dialogs.SCX, that displays a Save dialog; this code produces the dialog shown in **Figure 13.** This code specifies that VCX and all files (the default) should be available. It sets nFilterIndex to 2 to force VCX files as the default file type, and lOverwritePrompt to .T. so the user receives a warning if they select an existing file.

```
with Thisform.oDialog as SFCommonDialog of SFDialogs.vcx
    .AddFilter('Class Library (*.vcx)', '*.vcx')
    .nFilterIndex      = 2
    .lSaveDialog       = .T.
    .cTitlebarText     = 'Select File'
    .cDefaultExtension = 'vcx'
    .lOverwritePrompt  = .T.
    if .ShowDialog()
        lcFile = 'You chose ' + .cFileTitle + ' from ' + .cFilePath
        messagebox(lcFile)
    endif .ShowDialog()
```

```
endwith
```

Task dialog

Windows 7 has a 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 14** shows an example of a task dialog.



My Application

An unexpected error occurred.

It appears as though your data is completely corrupted and useless.

Maybe next time you'll backup like you're supposed to.

Figure 14. The Windows 7 task dialog is much more customizable than the message box dialog.

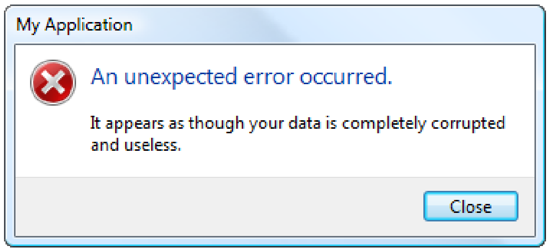
The API for these dialogs is quite complex and difficult to use from VFP. Fortunately, Craig Boyd has created a wrapper COM object written in .NET for them, making them easy to use in VFP. This object is part of the Vista Toolkit component in Sedna and can be downloaded from VFPX (http://tinyurl.com/3kw79cb). You need to register VistaDialogs4COM.dll for COM using RegAsm (the .Net equivalent of RegSvr32). The samples for this document includes Register.BAT with the following contents (edit the path for VistaDialogs4COM.dll so it's correct on your system). Right-click and choose Run as Administrator to register the DLL.

```
rem First unregister
C:\Windows\Microsoft.NET\Framework\v2.0.50727\regasm.exe VistaDialogs4COM.dll /u

rem Now re-register to generate TLB
C:\Windows\Microsoft.NET\Framework\v2.0.50727\regasm.exe VistaDialogs4COM.dll /codebase
/tlb
```

The following code, taken from TaskDialogSimple1.PRG, shows a simple example using the task dialog. This example doesn't do much more than MESSAGEBOX() would, but it looks more modern (**Figure 15**).

```
#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
```


My Application

An unexpected error occurred.

It appears as though your data is completely corrupted and useless.

Figure 15. 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 14**.

```
#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.

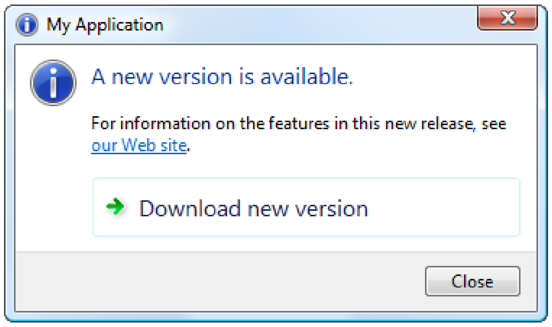
```
loButton = createobject('VistaDialogs4COM.TaskDialogCommandLink')
loButton.Instruction = 'Download new version'
loTaskDialog.AddControl(loButton)
loCommandLinkEventHandler = newobject('TaskDialogCommandLinkEventHandler', ;
  'DialogEventHandlers.PRG')
eventhandler(loButton, loCommandLinkEventHandler)
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 16**.


My Application

A new version is available.

For information on the features in this new release, see our Web site.

Download new version

Close

Figure 16. 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, would be nice.

Windows 7 taskbar

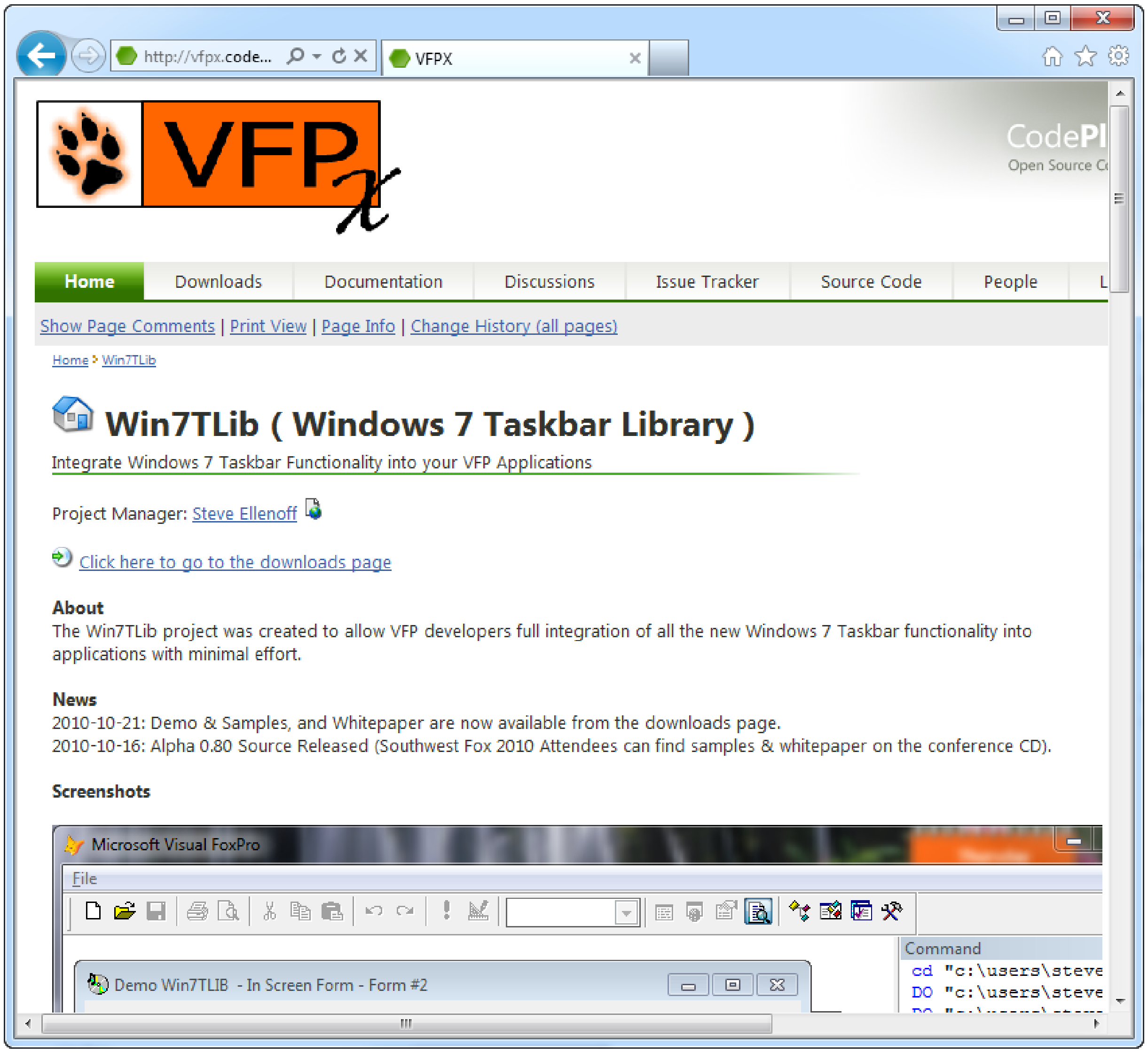
Microsoft made an incredible amount of changes to the taskbar in Windows 7. These features help you to be a lot more productive than earlier versions of Windows. I won't go through the list of features here; they're covered in great detail in many blog posts, including http://tinyurl.com/6le7flt and http://tinyurl.com/6gvlfxo.

Even better, your application can control many aspects of the taskbar, including what the icon for your button should look like, what image to display for the preview, what items appear in the jumplist (the list of links that appears when you right-click the taskbar button), and many other things. The following blog posts just start to touch on programming the taskbar:

- http://tinyurl.com/3ajhuxb
- http://tinyurl.com/6gvlfxo
- http://tinyurl.com/67cxxox
- http://tinyurl.com/69kg9ba
- http://tinyurl.com/62kt897

Unfortunately, it isn't possible to interact with the taskbar directly in VFP. Fortunately, VFP developer Steve Ellenoff has created a wrapper library for the taskbar API he calls Win7TLib and made it available to other VFP developers on the community VFX site.

To download Win7TLib, navigate your browser to http://vfpx.codeplex.com and click the Win7TLib link in the Project list to go to the Win7TLib home page (**Figure 17**), then click the download link for the download page from which you can download the source code, demos, and complete documentation.



Win7TLib (Windows 7 Taskbar Library)

Integrate Windows 7 Taskbar Functionality into your VFP Applications

Project Manager: Steve Ellenoff

Click here to go to the downloads page

About

The Win7TLib project was created to allow VFP developers full integration of all the new Windows 7 Taskbar functionality into applications with minimal effort.

News

2010-10-21: Demo & Samples, and Whitepaper are now available from the downloads page.

2010-10-16: Alpha 0.80 Source Released (Southwest Fox 2010 Attendees can find samples & whitepaper on the conference CD).

Screenshots

```
cd "c:\users\steve
DO "c:\users\steve
```

Figure 17. You can download Win7TLib from its page on VFPX.

I'm not going to go into detail on all of Win7TLib's features; Steve's documentation is very thorough and he provides a lots of demo code as well. Rather than going through his demos or creating my own, I thought I'd show you a real-world example of how I'm using Win7TLib to make my flagship application, Stonefield Query, take advantage of new features in Windows 7.

Although Win7TLib has a lot of features, I make specific use of the following:

- Adding a custom button to the taskbar thumbnail window (see **Figure 18**). In my case, clicking the button previews the selected report.

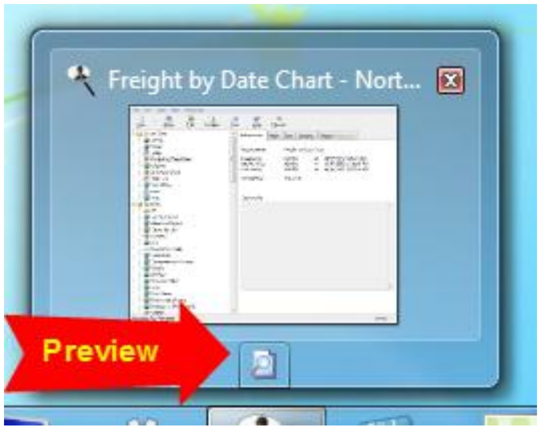


Figure 18. Thanks to Win7TLib, the thumbnail includes a custom button.

- Adding a custom category to the jumplist (see **Figure 19**), allowing the user to pin reports to the jumplist and run them even when the application isn't open.

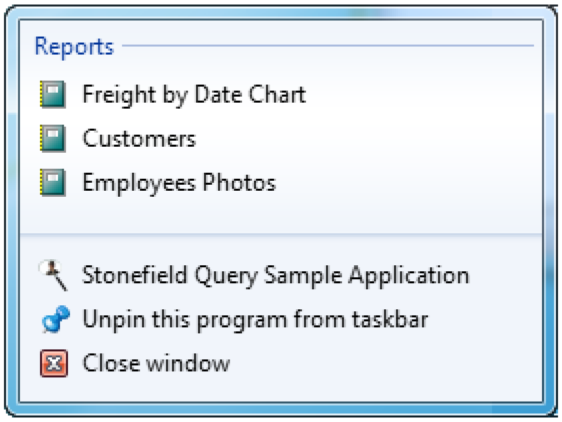

Reports
Freight by Date Chart
Customers
Employees Photos
Stonefield Query Sample Application
Unpin this program from taskbar
Close window

Figure 19. You can add custom categories to your application's jumplist.

- Associating certain file extensions with the application. In my case, I associate SFX files, which are XML files containing report definitions, with Stonefield Query. The benefit is that these files display using the icon you wish and automatically launch the application when double-clicked.
- Displaying a progress meter within the taskbar button, in my case when a report runs. **Figure 20** shows what the button looks when a report is running.

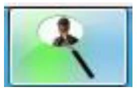


Figure 20. While a report runs, the taskbar button displays a progress meter.

- Flashing the taskbar button; that's done after the report preview is ready when Stonefield Query doesn't have focus.

To use Win7TLib in your application, add Win7TLib.PRG and Win7TLib_Visual.VCX to your project and copy Win7TLib.DLL to your application folder or include it in the VFP path. You'll of course also need to deploy Win7TLib.DLL with your application.

Somewhere during application startup, initialize Win7TLib. Since I want my application to work in earlier versions of Windows as well, I wrap the initialization code in a version check. It also checks whether we're running the application from the IDE or as an EXE. If we don't do this, then registration of services that later code performs does it for VFP9.EXE rather than the application's EXE. This means I don't get Win7TLib services when running in the IDE but that's a small tradeoff for mixing up services between VFP9.EXE and your EXE.

```
* Determine if we're running Windows 7 or later. We'll only do this in the
* runtime, not the IDE, to avoid confusing VFP9.EXE with SFQUERY.EXE.

plWindows7OrLater = os(3) >= '6' and iif(os(3) = '6', os(4) = '1', .T.) and ;
   version(2) = 0

* Initialize the Win7TLib library.

if plWindows7OrLater
   set procedure to Win7TLib
   llWin7Initialized = Initialize_Win7TLib()
endif plWindows7OrLater
```

Before the application terminates, uninitialize the library to release resources:

```
if llWin7Initialized
   UnInitialize_Win7TLib()
endif llWin7Initialized
```

Win7TLib needs a subclass of its Taskbar_Library_Settings class (located in Win7TLib.PRG) named Win7TLib_Application_Settings. This subclass contains the specific settings for your application. The two most important settings as cAppID, which uniquely identifies your application to Windows 7, and cDefaultForMode, which specifies how Win7TLib deals with windows. I use "TOP" since Stonefield Query runs as a top-level form application; see the Win7TLib documentation for other values. Win7TLib automatically instantiates Win7TLib_Application_Settings so it needs to be in a PRG included in the VFP path or opened with SET PROCEDURE.

```
define class Win7TLib_Application_Settings as TaskBar_Library_Settings ;
   of Win7TLib.PRG
   cAppID = 'Stonefield.Query.Application'
   cDefaultFormMode = 'TOP'
enddefine
```

The next thing to do is drop an instance of Taskbar_Helper (in Win7TLib_Visual.VCX) onto your main application form or some other form that's always open. I named the instance oTaskBarHelper.

Somewhere after the form has been instantiated, call upon the services you want registered. For example, as I mentioned earlier, Stonefield Query creates a taskbar toolbar button that previews the selected report when clicked, registers its SFX report files so they have the desired icon and automatically launch Stonefield Query when double-clicked, and adds a custom category named "Reports" to the jumplist where the user can pin reports they frequently run. The following code, executed during startup, does these tasks. First, let's look at the code that creates the taskbar toolbar button. This code defines a single button with an icon of Preview.ICO and a tooltip of "Preview."

```
if plWindows7OrLater
   with This.oTaskbarHelper.oTaskbar.Toolbar
      .nButtonCount = 1
      loButton      = .GetButton(1)
      with loButton
         .cIcon    = 'preview.ico'
         .cTooltip = "Preview"
      endwith
      .UpdateToolbar()
   endwith
```

The next set of code associates SFX files with the application. The code first unregisters an association with those files, then associates them with Stonefield Query. The third parameter is the string to display as the "file type". In the fourth parameter, which specifies the path for the icon file to use for SFX files, oApp.cAppDir is the directory the application is running from.

```
   with This.oTaskbarHelper.oTaskbar.JumpList.FileTypeReg
      lcExt = "SFX"
      .UnRegisterFileType(lcExt, .T.)
      .RegisterFileType(lcExt, .T., 'Stonefield Query Report file', ;
         oApp.cAppDir + 'Report.ico')
   endwith
```

The last set of code adds a custom category named "Reports" to the jumplist.

```
   loCategory = createobject('JumpList_Custom_Category', 'Reports')
   with This.oTaskbarHelper.oTaskbar.JumpList
      .lInclude_CustomCategories = .T.
      .AddCustomCategory(loCategory)
   endwith
endif plWindows7OrLater
```

While a report runs, Stonefield Query displays a progress meter in its taskbar button. In this case, we aren't showing the exact progress, since that isn't known, but rather a moving "marquee" effect showing that something is happening. The following code does this:

```
if plWindows7OrLater
   This.oTaskbarHelper.oTaskbar.TaskbarButton.SetProgressStyle('Indeterminate')
endif plWindows7OrLater
```

After the report is done, the following code clears the progress meter and flashes the taskbar button three times if the application is minimized:

```
if plWindows7OrLater
   This.oTaskbarHelper.oTaskbar.TaskbarButton.ClearProgress()
   if This.WindowState = 1
      This.oTaskbarHelper.oTaskbar.TaskbarButton.Flash(3)
   endif This.WindowState = 1
endif plWindows7OrLater
```

The user can pin a report they run frequently to the Stonefield Query taskbar so it appears in the custom Reports category. The following code, executed when the user right-clicks a report and chooses “Pin to Taskbar” (which only appears if plWindows7OrLater is .T.), does that:

```
local loLink, ;
   loJumpList, ;
   loCategory
with This
   loLink = createobject('JumpList_Link')
   with loLink
      .cAppPath   = _vfp.ServerName
      .cTitle     = This.oReport.cReportName
      .cIconPath  = oApp.cAppDir + 'Report.ico'
      .cArguments = '"report=' + This.oReport.cReportName + '" preview'
   endwith
   loJumpList = .oTaskbarHelper.oTaskbar.JumpList
   loCategory = loJumpList.CustomCategories.GetItem(1)
   loCategory.AddJumpListItem(loLink)
   loJumpList.CreateJumpList()
endwith
```

This code creates a JumpList_Link object, which contains the properties of the link in the jumplist. cAppPath is set to the full path for the application’s EXE. cTitle is the text to display in the jumplist, in this case the name of the report, and cIconPath is the name of the icon to display. cArguments is the list of parameters to pass to the application specified in cAppPath; in this case, the parameters specify that the application is supposed to run the specified report and send it to the preview window. The code then adds the link to the first custom category (“Reports” in this case).

When the user clicks a button in the taskbar toolbar, the On_Toolbar_Button_Click event of your Taskbar_Helper instance fires. Among other parameters, the event is passed the ID of the clicked button. The following code in the On_Toolbar_Button_Click event of my instance runs the current report when the first (and only) button is clicked:

```
lparameters toToolbar, ;
   toForm, ;
   tnID
do case
   case tnID = 1 and vartype(Thisform.oReport) = 'O'
      Thisform.RunReport(.T.)
```

```
endcase
```

When the user double-clicks a file associated with the application (an SFX file in my case), Windows calls the application and passes it "/handledocument:" followed by a full path to the file. Your application can then do something with that parameter. In my case, Stonefield Query previews the report.

Win7TLib has many other features, including the ability to control the image that appears for both the toolbar thumbnail and live preview, overlaying the toolbar button with other images (great for status notification, for example), grouping buttons, and lots of other things I'm sure you'll find useful.

Other UI considerations

MSDN has several documents describing Windows 7 user interface guidelines; "Windows User Experience Interaction Guidelines" (http://tinyurl.com/5ttcy4) 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 Windows 7 applications.

Resources

Lots of resources are available for learning more about Windows 7 security, user interface, and other features. Here are some of the resources I've used:

- Windows 7 Developer Guide (http://tinyurl.com/4bhyjfh)
- The Advantages of Running Applications on Windows Vista (http://tinyurl.com/4hldyws)
- The Windows Vista and Windows Server 2008 Developer Story: Windows Vista Application Development Requirements for User Account Control (http://tinyurl.com/67bu3d) or the more complete Windows Vista Application Development Requirements for User Account Control Compatibility (http://tinyurl.com/mfdyff)
- Getting Started with User Account Control in Windows Vista Beta 2 (http://tinyurl.com/4efdrjo)
- The Windows Vista and Windows Server 2008 Developer Story: Application Compatibility Cookbook (http://tinyurl.com/yevsyab)
- Windows User Experience Interaction Guidelines (http://tinyurl.com/5ttcy4)
- Registry Virtualization (http://tinyurl.com/4w8wl8b)
- System Font (Segoe UI) (http://tinyurl.com/5txp49c)
- Icons (http://tinyurl.com/4f39g23)
- Craig Boyd's blog (http://www.sweetpotatosoftware.com/SPSBlog)

Summary

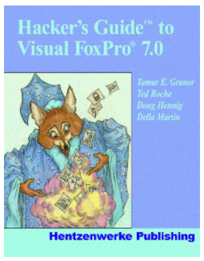
Whether you're planning to use Windows 7 yourself or not, you have to prepare for your customers who do. Go over your applications and ensure that files you write to are stored in accessible locations rather than the Program Files folder, use the HKEY_CURRENT_USER hive in the Registry rather than HKEY_LOCAL_MACHINE, adjust your installer to handle the Windows 7 issues outlined in this document, and take advantage of some of the new features in Windows 7, including new dialogs and taskbar functionality. This will give your application a more modern look and a longer shelf life.

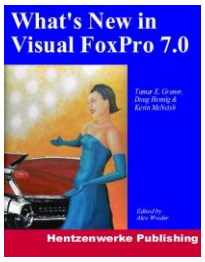
Biography

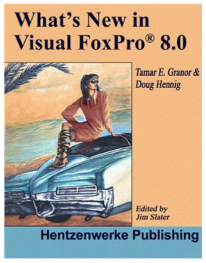
Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace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

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

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






What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing