The Latest Techniques in Deploying VFP Applications

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Corporate Web sites: www.stonefieldquery.com
and www.stonefieldsoftware.com
Personal Web site : www.DougHennig.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

Deployment of your application is easily as important as the rest of the development cycle since it's the first encounter your users have with the application. This document looks at some new ideas for application deployment, including creating a workstation-only installer, installing your application on a server without requiring a workstation installation (zero footprint deployment), installing the .NET framework, and how to update your applications over the Internet.

Introduction

Although it's sometimes a last-minute consideration, deploying your application is an extremely important part of the development process. After all, if the user can't install your software, you won't get paid. The installer is often the first interaction the user has with your software; if something goes wrong, it gives them a bad impression about the software and your company in general. Also, problems during installation can be the cause of a lot of technical support calls. So, creating the best possible installation experience for the user is as important as creating the best possible application usage experience.

Application deployment is such a huge subject that books could be written on it, and in fact have been. *Deploying Visual FoxPro Solutions*, by Rick Schummer, Rick Borup, and Jacci Adams (available from Hentzenwerke, http://www.hentzenwerke.com), is the go-to book for everything related to deploying VFP applications. Rather than covering all things deployment related, this document focuses on some new techniques for deploying Visual FoxPro applications. Specifically, we'll look at:

- What files really need to be installed and where should they go.
- Why you should digitally sign your executable and your setup application.
- What's involved in installing the application on a server and running it on a workstation.
- What registration-free installation is and how to do it.
- Automating the deployment process.
- Installing the appropriate version of the .NET framework if your VFP application uses .NET components.
- Automatically updating your application when a new version is released.

Before we begin, I should mention that I've been using Inno Setup to create installers for applications for about a decade. Inno Setup is free, fast, lightweight, and uses text-based scripts so it's easy to work with and you can even generate scripts programmatically, something I do frequently. For more information about Inno Setup, see my white paper titled "Installing Applications Using Inno Setup," available from the Technical Papers page of my web site (http://doughennig.com/papers/default.html). This document uses Inno Setup scripts to illustrate deployment concepts, but they're equally applicable to your installation tool of choice.

What really needs to be installed and where

For many years, VFP developers installed the VFP runtime files in the Windows System folder because that's what the old Setup Wizard did it and we didn't know any better. However, as operating systems became more locked down and developers started running into version conflicts when multiple applications using different runtimes files were installed, we started refining the locations of the files, and even which files are really needed.

As an example of the latter point, until recently my setup executable used to install ASycFilt.DLL, ComCat.DLL, OLEAut32.DLL, OLEPro32.DLL, StdOLE2.TLB, HH.EXE, HHCtrl.OCX, ITIRCL.DLL, and ITSS.DLL. While these are all files that VFP applications need to run, they've been core operating system files for a really long time so they can be assumed to be present. Why did I install them then? Because I always had and hadn't taken a close look at what really needs to be installed and where.

Table 1 lists the files usually mentioned as required by a VFP application and where they should be installed on a user's system, such as at http://tinyurl.com/krc6fjf. (The list doesn't include the app files themselves or any ActiveX controls used by the application.) In the Location column, "Win" means the Windows folder (usually C:\Windows), "Sys" means the Windows system folder (usually C:\Windows\System32 on a 32-bit system and C:\Windows\SysWOW64 on a 64-bit system), "Common" means the VFP common files folder (usually C:\Program Files\Common Files\Microsoft Shared\VFP on a 32-bit system and C:\Program Files (x86)\Common Files\Microsoft Shared\VFP on a 64-bit system), and "App" means the application folder. In the Register column, Yes means this is a COM component that needs to be registered using RegSvr32 as part of the installation process.

Table 1. Files installed as part of a VFP application installation.

File	Location	Purpose	Register	Required
MSVCR71.dll	Sys	Visual C++ runtime	No	Yes
GDIPlus.dll	Sys	GDI+ graphics support	No	Yes
VFP9R.dll	Common	VFP 9 runtime	Yes	Yes
VFP9T.dll	Common	VFP 9 runtime	Yes	If multithreaded DLL
VFP9REnu.dll	Common	VFP 9 English resource	No	Yes
VFP9R*.dll		VFP 9 language resource files (such as VFP9RDeu.dll for German)	No	If necessary
FoxHHelp9.exe	Common	VFP 9 help support	Yes	If CHM used
FoxHHelpPS9.dll	Common	VFP 9 help support	Yes	If CHM used
MSXML3.dll	Sys	XML support	Yes	If XMLTOCURSOR and CURSORTOXML used
MSXML3R.dll	Sys	XML support	No	If XMLTOCURSOR and CURSORTOXML used
MSXML4.dll	Sys	XML support	Yes	If XMLAdapter used
MSXML4R.dll	Sys	XML support	No	If XMLAdapter used
ReportBuilder.app	App	Report designer	No	If MODIFY REPORT used
ReportPreview.app	App	Report preview	No	If REPORT FORM PREVIEW used
ReportOutput.app	App	Report output	No	If REPORT FORM used

Let's look at this list critically. First, it turns out that you don't actually need to install MSXML3*.* since those files have been part of the operating system since Windows XP and I'm going to assume that's the oldest operating system we'll support. If you must support older systems such as Windows 2000 and you use CURSORTOXML() or XMLTOCURSOR() in your application, feel free to install these files.

Second, as noted in the table, MSXML4 is only required if you use the XMLAdapter class; if you don't, you can skip those files.

Next, let's look at the VFP runtime files. You don't need VFP9T.DLL unless your application includes a multi-threaded COM server, so you may be able to drop that from the list. Also, it turns out you can install the runtimes in the application folder rather than the Common folder, and in that case, you don't have to register them. Why would you want to do that? After all, putting them in the Common folder means any VFP application can use them, saving on disk space. First, disk space is cheap these days and the VFP runtimes don't take up a lot of room: less than 6 MB. Second, as hard as it is to believe, there are still lots of developers out there using a variety of VFP 9 runtimes: the original release, SP1, SP2, and the SP2 patches. If your application uses the features of the latest SP2 build and the user installs another VFP application that overwrites the runtimes in the Common folder with the original release version, your application breaks. Also, what if the user uninstalls the other application and part of that process removes the VFP 9 runtimes? That shouldn't happen but you can't guarantee what their uninstaller does. That also kills your application. So, putting the VFP 9 runtimes in the application folder removes the requirement to register the files and protects you from version-itis or uninstalled runtimes.

It also turns out that you can put MSVCR71.DLL and GDIPlus.DLL in the application folder, and I think you should for the same reasons outlined above.

The FoxHHelp*.* files should be installed in a common location because FoxHHelp9.EXE has to be registered so it can't be in an application-specific folder if more than one VFP application is installed. However, in testing, it turns out that FoxHHelpPS9.DLL doesn't have to be registered.

Listing 1 shows the contents of a text file named VFPCoreFiles.txt that can be #INCLUDEd in an Inno Setup script to install the core set of VFP 9 runtime files every VFP application needs. To this set of files, you can add MSXML4*.* if your application uses XMLAdapter, MSXML3*.* if your application uses CURSORTOXML() or XMLTOCURSOR() and needs to run on Windows 2000 or earlier, VFP9T.DLL if you have any multi-threaded COM servers, Report*.APP depending on your reporting needs, and any of the VFP9R resource DLLs beside the English one. You can also remove the FoxHHelp*.* files and the [Run] section if your application doesn't use CHM-based help.

Listing 1. VFPCoreFiles.txt specifies the core set of runtime file every VFP application needs.

```
[Files]
Source: "{#VFPCoreFiles}\msvcr71.dll";      DestDir: "{app}";
   Flags: ignoreversion
Source: "{#VFPCoreFiles}\gdiplus.dll";      DestDir: "{app}";
   Flags: ignoreversion
Source: "{#VFPCoreFiles}\vfp9r.dll";        DestDir: "{app}";
   Flags: ignoreversion
Source: "{#VFPCoreFiles}\vfp9renu.dll";     DestDir: "{app}";
   Flags: ignoreversion
Source: "{#VFPCoreFiles}\foxhhelp9.exe";    DestDir: "{cf}\Microsoft Shared\VFP";
```

```
   Flags: sharedfile uninsneveruninstall
Source: "{#VFPCoreFiles}\foxhhelpps9.dll";  DestDir: "{cf}\Microsoft Shared\VFP";
   Flags: sharedfile uninsneveruninstall

[Run]
Filename: "{cf}\Microsoft Shared\VFP\foxhhelp9.exe"; Parameters: /regserver
```

If you don't have them handy, you can download VFP runtimes from Christof Wollenhaupt's web site: http://www.foxpert.com/runtime.htm.

The setup executable

Not unreasonably, users expect application installation to be smooth and error-free. Inno Setup makes it easy to create such an installer with features such as:

- Allowing them to choose the program group and folder to install the application in. I don't know about you but I hate installers that don't allow me to choose both the drive and the folder as I usually install applications on my D: drive.
- Displaying a license agreement and any special notes or instructions the user needs to be aware of.
- Optionally creating a desktop icon and optionally running the application after installation is complete.
- Displaying the correct information for the application in the Add or Remove Programs dialog.

Listing 2 is the content of MyApp.ISS, an Inno Setup script for an application named MyApp included with the sample files for this document.

Listing 2. MyApp.ISS is an Inno Setup script to create an installer for a simple application.

```
; These two lines go in every installer.
#define VFPCoreFiles    "..\..\Runtimes"
#include VFPCoreFiles + "\VFPCoreFiles.txt"

; Define the location of the application files.
#define AppFiles        "..\"

; Define the application name, executable, company, copyright, etc.
#define MyAppRoot       "myapp"
#define MyApplication   MyAppRoot + ".exe"
#define MyHelp          MyAppRoot + ".chm"
#define MyAppName       "My Application"
#define MyAppID         MyAppName
#define MyURL           "http://www.stonefieldquery.com"
#define MyVersion       GetFileVersion(MyApplication)
#define MyCompany       "Stonefield Software Inc."
#define MyCopyright     "Copyright © 2014 Stonefield Software Inc. All rights
   reserved."

[Setup]
```

```
AppName={#MyAppName}
AppID={#MyAppID}
AppVerName={#MyAppName}
AppVersion={#MyVersion}
AppPublisher={#MyCompany}
AppPublisherURL={#MyURL}
AppSupportURL={#MyURL}
AppUpdatesURL={#MyURL}
AppendDefaultDirName=no
DefaultDirName={pf}\{#MyAppName}
DefaultGroupName={#MyAppName}
DisableProgramGroupPage=false
LicenseFile=eula.rtf
Compression=lzma
SolidCompression=true
PrivilegesRequired=admin
UninstallDisplayIcon={#AppFiles}Source\myapp.ico
OutputBaseFilename=setup
AppCopyright={#MyCopyright}
VersionInfoDescription={#MyAppName} Setup
VersionInfoVersion={#MyVersion}
VersionInfoTextVersion={#MyVersion}
VersionInfoCompany={#MyCompany}
VersionInfoCopyright={#MyCopyright}
ShowLanguageDialog=yes
WizardImageFile=InstallLogo.bmp
WizardSmallImageFile=compiler:WizModernSmallImage-IS.bmp
InfoBeforeFile=beforefile.rtf

[Tasks]
Name: desktopicon; Description: Create a &desktop icon; GroupDescription: Icons

[Files]
Source: "{#AppFiles}\{#MyApplication}";  DestDir: "{app}";  Flags: ignoreversion
Source: "{#AppFiles}\{#MyHelp}";         DestDir: "{app}";  Flags: ignoreversion
Source: "{#AppFiles}\ComServer.dll";     DestDir: "{app}";  Flags: ignoreversion
   regserver

; Needed because we're using the TreeView control.
Source: "{#VFPCoreFiles}\MSComCtl.ocx";  DestDir: "{sys}";  Flags: onlyifdoesntexist
   sharedfile regserver noregerror

; Needed because we're using XMLAdapter.
Source: "{#VFPCoreFiles}\MSXML4.dll";    DestDir: "{sys}";  Flags: onlyifdoesntexist
   sharedfile regserver noregerror
Source: "{#VFPCoreFiles}\MSXML4R.dll";   DestDir: "{sys}";  Flags: onlyifdoesntexist
   sharedfile

; Needed because we have a multi-threaded DLL.
Source: "{#VFPCoreFiles}\VFP9T.dll";     DestDir: "{app}";  Flags: ignoreversion

[Icons]
Name: "{group}\{#MyAppName}";                          Filename:
   "{app}\{#MyApplication}";  WorkingDir: "{app}"
Name: "{group}\{#MyAppName} Help";                     Filename: "{app}\{#MyHelp}"
```

```
Name: "{group}\{cm:UninstallProgram,{#MyAppName}}";  Filename: "{uninstallexe}"
Name: "{commondesktop}\{#MyAppName}";                 Filename:
   "{app}\{#MyApplication}";  WorkingDir: "{app}";  Tasks: desktopicon

[Run]
Filename: "{app}\{#MyApplication}";  Description: "{cm:LaunchProgram,{#MyAppName}}";
   Flags: nowait postinstall skipifsilent
```

Here are some things to note about this script:

- It's as generic as possible. Note the use of #DEFINEs to define those things that change from application to application. That means that other than the [Files] section, most of this script can be reused for other applications by changing the #DEFINE statements as necessary.
- This script expects a directory structure in which the installer-related files (BeforeFile.rtf, EULA.rtf, InstallLogo.bmp, and MyApp.ISS) are in a subdirectory of the folder containing the application files. **Figure 1** shows the directory structure for the samples accompanying this application (there are actually additional folders, but they're left out of this image for simplicity). The MyApp folder contains all of the application files (EXE, DLL, CHM, etc.) plus the PJX files. Source contains the source code for the EXE and DLL and HTMLHelp contains the source for the CHM. Installer contains the installer-related files.

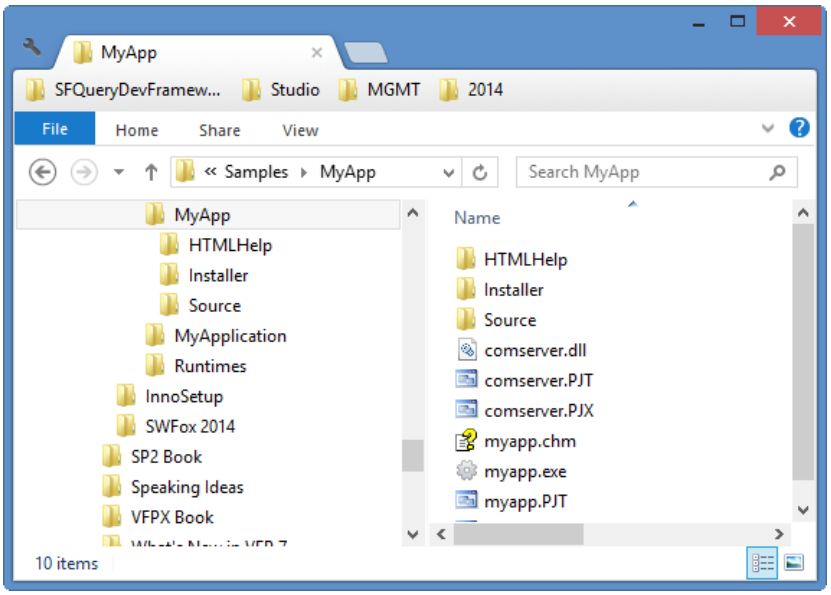

Figure 1. The installer script expects a directory structure with the installer-related files in a subdirectory of the application files.

- The setup executable created by this script displays a custom image on the first page of the wizard (the WizardImageFile setting), a license agreement on the second page (the LicenseFile setting), information about the application (the InfoBeforeFile setting), allows the user to choose where to install the program, allows them to choose the program group, has an option to create a desktop icon (the Tasks section

and last item in the Icon section), includes an option to run the application after installation is complete (the item in the Run section), and creates shortcuts for the application, its help file, and the uninstaller in the program group (the first three items in the Icons section).

- The AppPublisher, AppSupportURL, and AppUpdatesURL settings allow the Add or Remove Programs dialog to display the correct information for the application as shown in **Figure 2**.

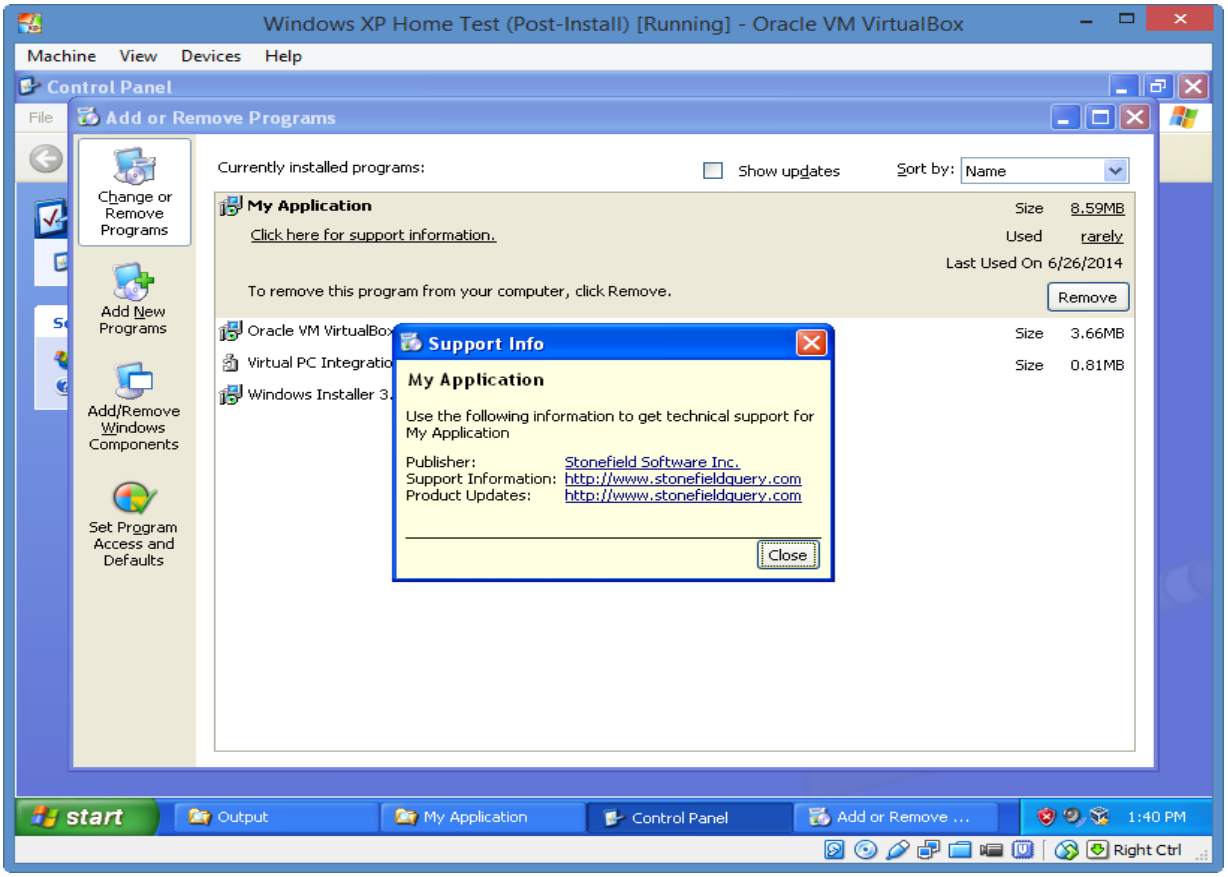

Figure 2. The settings in MyApp.ISS ensure the correct information about the application is displayed.

- The setup installs the files shown in **Figure 3** in the application folder. In addition to these files, it installs and registers MSComCtl.OCX and MSXML4*.* in the Windows system folder and registers those files as well as ComServer.DLL in the application folder.

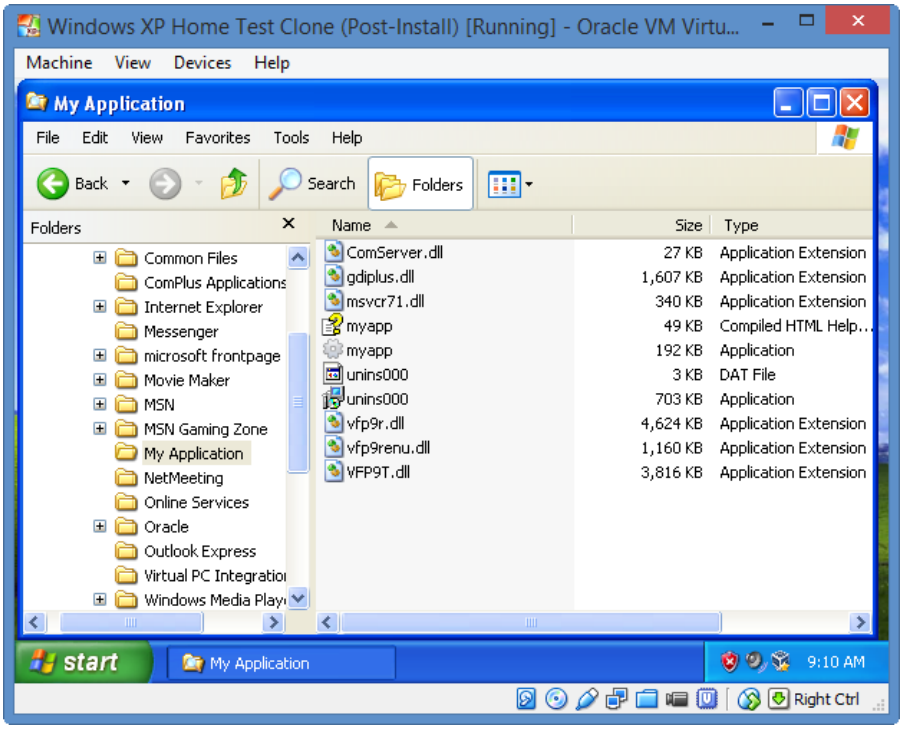

Figure 3. The file installed by the setup executable created by MyApp.ISS.

- If you instantiate an ActiveX control programmatically (that is, using CREATEOBJECT) rather than dropping it on a form, you may run into a licensing issue. For example, the ActiveX controls that ship with VFP, such as the TreeView and Date and Time Picker controls, require a license. Fortunately, there's a simple way to deal with this, at least with the Microsoft controls: install the license as part of the installation process. Add the following to the ISS file to install the license for the controls in MSComCtl.OCX:

```
[Registry]
; License for MSCOMCTL.OCX classes
Root: HKCR; Subkey: "Licenses\ED4B87C4-9F76-11D1-8BF7-0000F8754DA1";
   ValueType: string; ValueData: "knlggnmntgggrninthpgmnngrhqhnnjnslsh"
```

The net result is a professional installer with all of the features expected by users.

Well, almost all. **Figure 4** shows what happens when we install it on Windows Vista and later (in this case, Windows 7). (The left image appears when the installer is run from a remote location.) The "unknown publisher" message doesn't exactly inspire confidence. To make those go away, we need to digitally sign our setup executable.

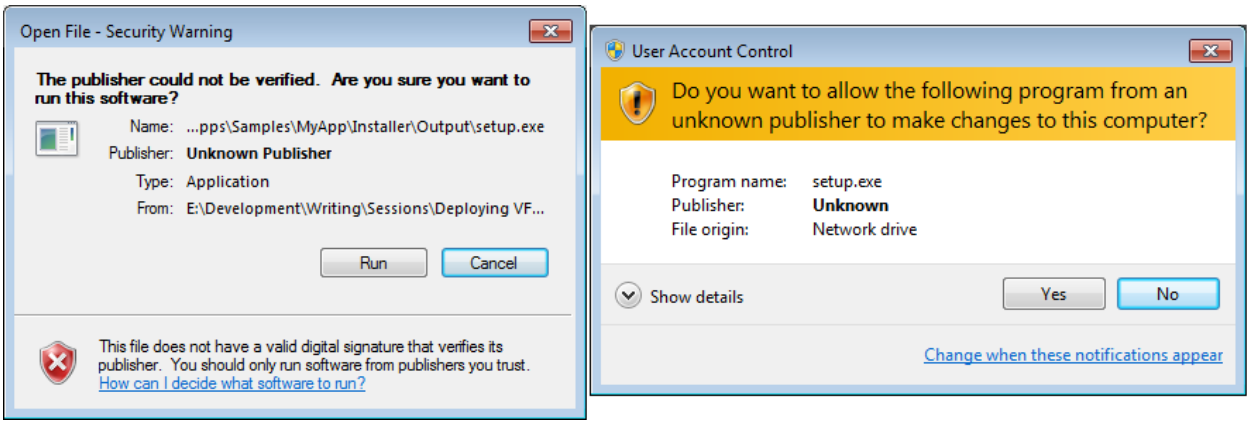

Figure 4. Running Setup.EXE results in these "unknown publisher" dialogs.

Digital signatures

Wikipedia defines a digital signature as "a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, such that the sender cannot deny having sent the message (authentication and non-repudiation) and that the message was not altered in transit (integrity)." One white paper I found useful for understanding digital signatures is "Securing Your Private Keys as Best Practice for Code Signing Certificates" by Larry Seltzer, http://tinyurl.com/nu35py3.

You apply a digital signature to something using a certificate generated by a trusted authority. There are numerous certificate authorities you can deal with: Symantec (which owns VeriSign, Thawte, and GeoTrust), Comodo, GoDaddy, and others. To obtain a certificate from one of these companies, you contact the company and provide them the information they require to confirm that you are who you say you are. This means a time-consuming process of jumping through numerous hoops, such as providing incorporation documentation, but it's for a good reason and you only have to do it once. You then purchase the type of certificate you need. In the case of signing Windows executables, it's a Microsoft Authenticode certificate. Certificates aren't cheap; for example, Thawte charges $499 US for a two-year certificate.

Once you've purchased the certificate, you'll likely get an email with a link to download the certificate into the certificate store on your machine. Normally, you'll want the certificate in file form to make signing easier, so you need to get it out of the certificate store after you've downloaded it. To do so, do the following:

- Start the Certificate Manager snap-in by running CertMgr.msc.
- The downloaded certificate is in the Personal\Certificates section. Right-click the certificate and select All Tasks, Export to bring up the Certificate Export Wizard.
- In the Export Private Key step, choose *Yes, export the private key*.
- In the Export File Format step, *Personal Information Exchange (.PFX)* is already selected; turn on the *Export all extended properties* options.

- In the Security step, turn on *Password* and enter a password.
- In the File to Export step, enter the name of the PFX file to create.

You can then use SignTool to sign EXEs using the PFX file. SignTool comes with Microsoft Visual Studio and the Microsoft Windows SDK. If you don't have Visual Studio, you can download the SDK from Microsoft's web site; search Microsoft.com for "download Windows SDK" to find the appropriate download page. You can typically find SignTool.EXE in C:\Program Files\Microsoft SDKs\Windows\v7.1a\Bin (the version number may be different depending on what version of the SDK you download and it may be in C:\Program Files (x86) instead). Here's an example of a command to sign an EXE:

```
"C:\Program Files\Microsoft SDKs\Windows\v7.1a\Bin\signtool.exe" sign
   /f certificatePath /p password /d "description" fileToSign
```

where *certificatePath* is the path to the PFX file, *password* is the password for the PFX, *description* is the description for the digital signature, and *fileToSign* is the path to the EXE to sign. Remember to add quotes around any path containing spaces.

It's actually even easier than that to sign the installer generated by Inno Setup. You can do it one of two ways:

- In the Inno Setup Compiler, choose Configure Sign Tools from the Tools menu, click Add, specify a name (I use "Standard"), and enter something like the following for the command:

  ```
  C:\Program Files\Microsoft SDKs\Windows\v7.1A\Bin\signtool.exe sign
     /f certificatePath /p password $p
  ```

  Note there are no quotes in the path and that $p is a placeholder for the SignTool parameters specified in the [Setup] section. Then add the following to the [Setup] section of your ISS file:

  ```
  SignedUninstaller=yes
  SignTool=Standard /d $q{#MyAppName}$q $f
  ```

  $q is a special symbol meaning insert a quote (you can't use actual quotes here) and $f is a placeholder for the name of the file to be signed. This assumes you have a constant defined for MyAppName. If not, use whatever you wish for the description.

- If you compile your ISS file using the Inno command line compiler, pass the following parameter to the compiler:

  ```
  "/sStandard=C:\Program Files\Microsoft SDKs\Windows\v7.1A\Bin\signtool.exe
     sign /f certificatePath /p password"
  ```

Be sure to put the certificate in a path with no spaces in any of the folder names or you'll get unhelpful error messages when you build the setup.

Figure 5 shows the results when running a digitally signed installer.

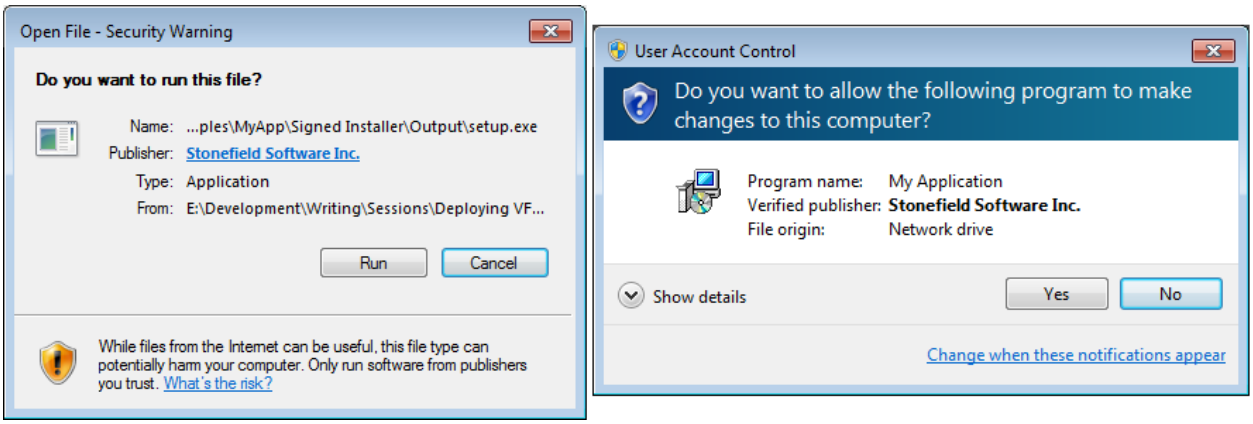

Figure 5. These dialogs appear when running a digitally signed installer.

Running the application on a workstation

Often applications are installed on a file server and run on workstations from the server folder. With a simple VFP application, it's possible to do that without installing anything on the workstation. For example, the main form in the sample MyApp.EXE that comes with the file accompanying this document works just fine in that case. However, there are several gotchas for this type of setup:

- If the application uses any ActiveX controls or COM servers, they have to be registered on the workstation. Registering them on the server isn't good enough.
- CHM-based help doesn't appear properly due to Window security for remote CHM files (**Figure 6**).

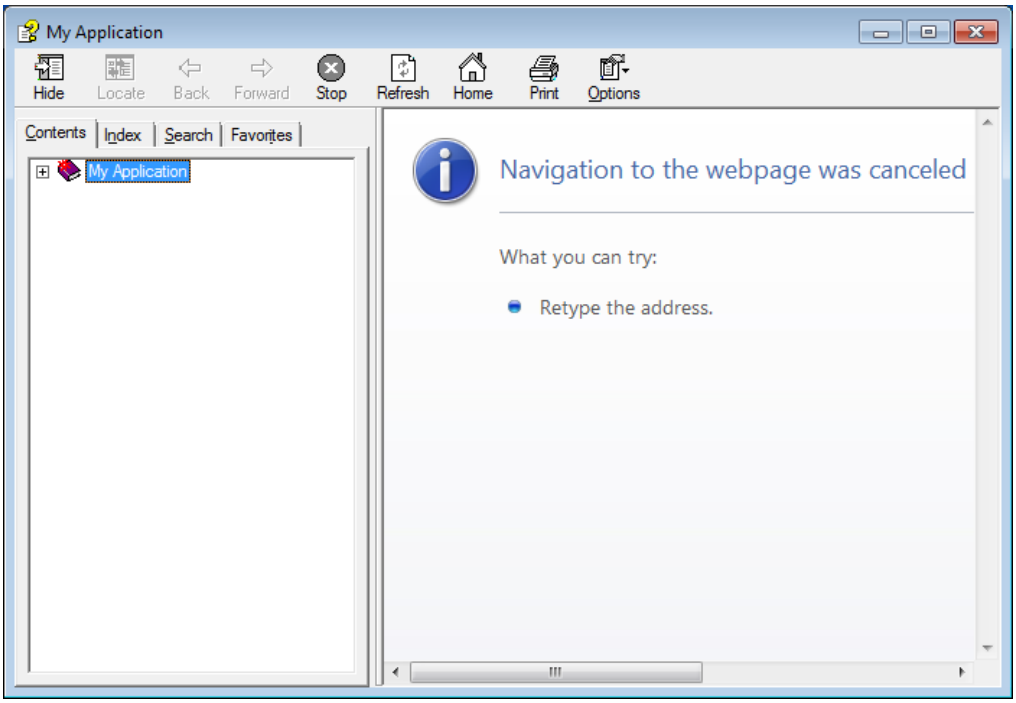

Figure 6. CHM-based help doesn't appear properly when the CHM is on a remote machine.

- The XMLAdapter class can't be used if MSXML4 isn't installed and registered on the workstation.
- The user has to manually create a shortcut to the application's executable on the server.

The workaround for these issues is to create a workstation installer. This installer is installed by the main installer on the file server along with the other application files. To use the application on a workstation, the user navigates Windows Explorer to the server folder and runs the workstation installer (WSSetup.EXE). This installer does the following:

- Installs and registers any ActiveX controls or COM servers used by the application.
- Adds to the Windows Registry the values necessary to allow CHM files located on a file server to display properly on a workstation; see http://tinyurl.com/3798vq for details.
- Creates shortcuts in a program group.
- Optionally creates a shortcut on their desktop to the main executable on the file server.

Listing 3 shows the script for WSSetup.ISS, a mostly generic workstation installer. WSSetup reads the application and executable names from Setup.INI, which is installed (along with WSSetup.EXE generated by WSSetup.ISS) as part of the full installer. It assumes the help file is the same name as the main executable but with a CHM extension. To create a workstation installer specific for an application, edit the [Files] section of WSSetup.ISS.

Listing 3. WSSetup.ISS is a mostly generic workstation installer.

```
; Define the location of the runtime files.
#define VFPCoreFiles    "..\..\Runtimes"

; Define the location of the application files.
#define AppFiles        "..\"

[Setup]
OutputBaseFilename=WSSetup
OutputDir=.

AppName={code:GetAppName}
AppVerName={code:GetAppName}
AppVersion={code:GetAppVersion}
AppendDefaultDirName=no
DefaultDirName={pf}\{code:GetAppName}
UsePreviousAppDir=no
DefaultGroupName={code:GetAppName}
Compression=lzma
SolidCompression=true
PrivilegesRequired=admin
VersionInfoDescription=Workstation Installer
; VersionInfoProductName and VersionInfoProductTextVersion are just here to avoid
```

```
; compiler warnings
VersionInfoProductName=.
VersionInfoProductTextVersion=.
UsePreviousLanguage=no
SignTool=Standard /d $qWorkstation Installer$q $f

[Messages]
WelcomeLabel2=This setup will install the necessary runtime files on your
computer.%n%nIt is recommended that you close all other applications before
continuing.
FinishedLabel=Setup has finished the workstation-only install on your computer. The
application may be launched by running one of the installed icons.
SelectTasksLabel2=Select any additional tasks that you would like to perform during
setup.
ReadyLabel1=Setup is now ready to begin the workstation-only install on your
computer.

[Tasks]
Name: desktopicon; Description: Create a &desktop icon; GroupDescription: Icons

[Code]
var
   SourceDirectoryPage: TOutputMsgWizardPage ;

function GetEXEName(Dummy: String): String;
begin
  Result := GetINIString('Setup', 'EXEName', '', ExpandConstant('{src}') +
       '\setup.ini') + '.exe';
end;

function GetCHMName(Dummy: String): String;
begin
  Result := GetINIString('Setup', 'EXEName', '', ExpandConstant('{src}') +
       '\setup.ini') + '.chm';
end;

function GetAppName(Dummy: String): String;
begin
  Result := GetINIString('Setup', 'AppName', '', ExpandConstant('{src}') +
       '\setup.ini');
end;

function GetAppVersion(Dummy: String): String;
var
  MyVersion: String;
begin
  GetVersionNumbersString(ExpandConstant('{src}') + '\' + GetEXEName(''), MyVersion);
  Result := MyVersion
end;

procedure InitializeWizard();
begin
  SourceDirectoryPage := CreateOutputMsgPage(wpInfoBefore,
    'Information', 'Setup will create a shortcut to the file listed below.',
    'Target Application: ' + ExpandConstant('{src}') + '\' + GetEXEName(''));
```

```
end;

function NextButtonClick(CurPageID: Integer): Boolean;
var
  bValidTarget: Boolean;

begin
  bValidTarget := true;
  if (CurPageID = SourceDirectoryPage.ID) and (FileExists(ExpandConstant('{src}') +
       '\' + GetEXEName('')) = false) then
  begin
    MsgBox('The application file does not exist in the ' + ExpandConstant('{src}') +
       ' directory. Please run this workstation-only setup file from the server
      directory where the program is already installed.', mbCriticalError, MB_OK);
    bValidTarget := false;
  end;
  Result := bValidTarget;
end;

[Files]
; This lists any ActiveX controls or COM servers that need to be registered to work.
Source: "{#VFPCoreFiles}\MSXML4.dll";   DestDir: "{sys}";  Flags: onlyifdoesntexist
   sharedfile regserver noregerror
Source: "{#VFPCoreFiles}\MSXML4R.dll";  DestDir: "{sys}";  Flags: onlyifdoesntexist
   sharedfile
Source: "{#VFPCoreFiles}\MSComCtl.ocx"; DestDir: "{sys}";  Flags: onlyifdoesntexist
   sharedfile regserver noregerror

; We need these files for CHM help to work.
Source: "{#VFPCoreFiles}\foxhhelp9.exe";    DestDir: "{cf}\Microsoft Shared\VFP";
   Flags: sharedfile uninsneveruninstall
Source: "{#VFPCoreFiles}\foxhhelpps9.dll";  DestDir: "{cf}\Microsoft Shared\VFP";
   Flags: sharedfile uninsneveruninstall

; A VFP COM server needs to be installed and registered locally, but that requires
; the VFP 9 runtimes too.
Source: "{#VFPCoreFiles}\MSVCR71.dll";   DestDir: "{pf}\My Application";
   Flags: ignoreversion deleteafterinstall
Source: "{#VFPCoreFiles}\VFP9R.dll";     DestDir: "{pf}\My Application";
   Flags: ignoreversion
Source: "{#VFPCoreFiles}\VFP9T.dll";     DestDir: "{pf}\My Application";
   Flags: ignoreversion
Source: "{#VFPCoreFiles}\VFP9REnu.dll";  DestDir: "{pf}\My Application";
   Flags: ignoreversion
Source: "{#AppFiles}COMServer.dll";      DestDir: "{pf}\My Application";
   Flags: ignoreversion regserver

[Registry]
; These keys allow remote CHM files to be viewed properly.
Root: HKLM; Subkey: "SOFTWARE\Microsoft\HTMLHelp\1.x\ItssRestrictions";
   ValueType: dword; ValueName: "MaxAllowedZone"; ValueData: 3
Root: HKLM; Subkey: "SOFTWARE\Wow6432Node\Microsoft\HTMLHelp\1.x\ItssRestrictions";
   ValueType: dword; ValueName: "MaxAllowedZone"; ValueData: 3

[Run]
```

```
Filename: "{src}\{code:GetEXEName}"; Description: "{cm:LaunchProgram,application}";
   Flags: nowait postinstall skipifsilent

[Icons]
Name: "{group}\{code:GetAppName}";         Filename: "{src}\{code:GetEXEName}";
   WorkingDir: "{src}"
Name: "{group}\{code:GetAppName} Help";    Filename: "{src}\{code:GetCHMName}"
Name: "{commondesktop}\{code:GetAppName}"; Filename: "{src}\{code:GetEXEName}";
   WorkingDir: "{src}"; Tasks: desktopicon
```

The following line in the [Files] section of the main installer's ISS file installs the WSSetup.EXE generated by WSSetup.ISS into the application's folder:

```
; This is the workstation installer.
Source: "WSSetup.exe";  DestDir: "{app}";  Flags: ignoreversion
```

The following lines generate the Setup.INI used by WSSetup.EXE:

```
; Generate SETUP.INI so the workstation installer has information it needs
[INI]
Filename: "{app}\Setup.ini"; Section: "Setup"; Key: "AppName";
   String: "{#MyAppName}"
Filename: "{app}\Setup.ini"; Section: "Setup"; Key: "EXEName";
   String: "{#MyApplication}"
Filename: "{app}\Setup.ini"; Section: "Setup"; Key: "Company";
   String: "{#MyCompany}"
```

Keeping the footprint on the workstation as light as possible has the benefit that when a new version of the application is installed, you may not have to update anything on the workstation at all. Only if the ActiveX controls or COM servers have changed would the user need to re-run WSSetup.EXE.

Registration-free installations

The two main things WSSetup.EXE does are update the Windows Registry so a remote CHM file is displayed properly and install and register the ActiveX controls and COM servers used by the application. If we could somehow deal with those two issues differently, there'd be nothing to do on the workstation at all except creating a shortcut to the application on the server.

We can deal with the help issue by using actual HTML help rather a CHM file. West Wind HTML Help Builder, the tool I've used for over a decade to create help files for my applications, can generate both CHM and HTML-based help. The latter is handy for posting the documentation on a web site but we can also use it to resolve the remote CHM issue: we can include both the CHM file and the HTML files when the application is installed and use which ever one is appropriate. When the CHM file is local, use it; otherwise, use the HTML files.

A West Wind HTML Help Builder source file is a VFP table with an HBP extension. Although you could deploy the entire table, all we really need is the mapping from the help ID to the

name of the HTML file for that topic. The following code creates a table named Help.dbf included in the PJX for MyApp:

```
select helpid, pk from myapp into table source\help
index on helpid tag helpid
```

The code shown in **Listing 4**, taken from ShowHelp.PRG, displays either the specified help topic in the application's CHM file or the appropriate HTML file for that topic. To make it generic, it uses a global variable named plUseHTMLHelp to determine whether CHM or HTML help is used; in this example, that variable is set to .T. in Main.PRG if the application's help file doesn't exist. To use it, put something like ShowHelp(Thisform.HelpContextID) into the Click method of a help button. Because this code launches the browser with the specified help file, there's no security issue with the CHM file.

Listing 4. ShowHelp.PRG displays either the specified topic in the CHM file or the HTML file for that topic.

```
lparameters tnTopic
local lcTopic

* If we're supposed to use HTML help, open the Help mapping table if necessary,
* look up the specified ID, and get the topic to display. Use ShellExecute to
* display the help file.

if plUseHTMLHelp
   if not used('HELP')
      use HELP in 0 shared again
   endif not used('HELP')
   if seek(tnTopic, 'HELP', 'HELPID')
      lcTopic = trim(HELP.PK) + '.htm'
   else
      lcTopic = ''
   endif seek(tnTopic, 'HELP', 'HELPID')
   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

* This is how you display a particular topic in a CHM file without needing
* FoxHHelp. However, due to Windows security, that won't work on a remote system.

*  ShellExecute(0, '', 'hh.exe', 'mk:@MSITStore:' + set('HELP', 1) + ;
      '::' + lcTopic, '', 1)

* Let's display the HTML help files generated by West Wind HTML Help Builder.
* You can, of course, specify "http://url" instead of "file://file" if the
* files are on a web site.

   ShellExecute(0, 'open', 'iexplore', 'file://' + ;
      fullpath('htmlhelp\index.htm') + '?page=' + lcTopic, '', 1)
```

```
* We're using CHM help so display the specified topic.

else
   help id tnTopic
endif plUseHTMLHelp
```

The second issue, ActiveX controls and COM servers, is trickier. Because of the way COM works, VFP expects to look in the Windows Registry to find out where an ActiveX control or COM server is located. Since we want a zero-footprint install, we want to avoid having to register the control. There's actually another reason why a registration-free installation is a good thing: version-itis. There can only be one file registered with a particular ClsID (class ID) value in the Windows Registry. If your application needs one version of the file and another application needs a different one, and they have the same ClsID, one of you will be unhappy. However, if your application can find the files it needs without looking in the Windows Registry, that issue goes away and the application always uses the correct file.

It turns out that there's a way to get VFP to not look in the Windows Registry for an ActiveX control or COM server. There are several articles online that discuss how to do this:

- Rick Strahl's "Custom Manifest Files in Visual FoxPro EXEs," http://tinyurl.com/pvbzdg8
- Craig Boyd's "PE Files, UAC, Reg-Free COM, and Other Crazy Stuff - Part 2," http://tinyurl.com/l6esm9m
- DBI Technologies' "How to Implement Reg Free COM," http://tinyurl.com/m55ayey
- "Registration-Free Activation of COM Components: A Walkthrough" by Steve White, http://tinyurl.com/yg5czbl (this isn't VFP-specific but discusses registration-free COM in general).

The mechanism discussed in all of these involves using a manifest file. A manifest file can do a number of things for your application, such as telling Windows that it should automatically elevate to administrator when running the application, but the thing we're interested in is specifying which ActiveX controls and COM servers the application uses so they don't have to be registered.

The bottom line is that you create an XML file named *MyApplication*.exe.manifest, where *MyApplication* is the name of your executable, and put it in the same folder as the PJX file for your application. When you build the executable, this manifest file is embedded in the EXE rather than the one that VFP would normally generate.

Listing 5 is an example of a manifest file.

Listing 5. A manifest file.

```
<?xml version="1.0" encoding="utf-8"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
   manifestVersion="1.0">
   <assemblyIdentity name="myapp.exe"
```

```
      version="1.0.0.0"
      processorArchitecture="x86"
      type="win32" />
   <file name="comserver.dll">
      <comClass clsid="{A0EB9AED-CEAA-41DF-A471-FD3C63FA1214}"
         threadingModel="Apartment"
         progid="comserver.comserver" />
   </file>
   <file name="msxml4.dll">
      <comClass clsid="{88D969C0-F192-11D4-A65F-0040963251E5}"
         threadingModel="Both"
         progid="Msxml2.DOMDocument.4.0" />
   </file>
   <file name="mscomctl.ocx">
      <comClass clsid="{C74190B6-8589-11D1-B16A-00C0F0283628}" />
         threadingModel="Apartment"
         progid="MSComctlLib.TreeCtrl" />
   </file>
</assembly>
```

The name attribute of the assemblyIdentity element contains the name of your executable. The file elements contain information about ActiveX controls or COM servers. You can have as many file elements as you need. The name attribute is the name of the file containing the control; this file, and any of its dependencies, should be in the program folder. COMServer.DLL contains the VFP COM server our sample application uses, MSComCtl.OCX contains the TreeView control, and MSXML4.DLL contains the MS XML 4 DOM class that XMLAdapter needs.

Inside the file element is the comClass element specifying the ActiveX control or COM server used in the application. If you use more than one class from the file, use as many comClass elements as you need. The clsid attribute is the class ID. You can get that by searching the Windows Registry for the class name. For example, the class name of the TreeView control is MSComctlLib.TreeCtrl (you can see that if you drop a TreeView control on a form and look at the OleClass property in the Properties window). The threadingModel attribute is the threading model used by the class. To get that, search the Windows Registry for the class ID and look for the ThreadingModel setting in one of the subnodes of the found node. If you don't know the name of the file containing the ActiveX control, you can get it from the InProcServer32 subnode. The progid attribute contains the class name.

Since nothing is registered on the workstation, you'll have to use HTML help or some other form of help instead of a CHM file.

The only difference in the Inno Setup script for a regular installation and a registration-free one is that everything, including ActiveX controls and COM server, is installed in the application folder and HTML help files are included as well.

The MyApp folder in the files accompanying this document contains the manifest file for the sample application, myapp.exe.manifest. To test this in a registration-free environment, do the following:

- Build MyApp.EXE from the MyApp project.
- Create a folder to hold the deployment files. For example, I created a folder named MyApplication.
- Copy the application executable, any supporting files (data and otherwise), the VFP runtimes, and any ActiveX and COM server files into that folder.
- On another system connected to this system (such as another system on the network or a virtual machine), navigate Windows Explorer to the deployment folder and run MyApp.EXE.

While testing this, I ran into a few issues:

- A VFP COM server works best if built as a multi-threaded DLL, which means the folder must contain VFP9T.DLL. If you build it as an EXE, you'll get an error when you instantiate the COM object. If you build it as a single-threaded DLL, instantiating the COM object creates another DLL, ComServerR1.DLL; see http://tinyurl.com/kywdynw for the reason. In that case, the program folder needs to be writable or you'll get an error because that DLL can't be created. Since it's a bad idea to make the program folder writable, it's best to just use a multi-threaded DLL.
- Although the TreeView control worked fine in a Windows XP client, clicking it caused a C5 error in Windows 7. I found a lead to a solution at http://tinyurl.com/qzu5t68: setting the AutoActivate property of the control to 3-Automatic. The weird thing about this solution is that AutoActivate appears in the VFP help but not in the Properties window or IntelliSense, and it crashes in Windows XP unless you surround it with a TRY structure.

This technique takes a bit of work to get right—using the proper settings in the manifest and architecting things like help differently—but it also means you could put a complex VFP application on a memory stick and it'd work on any workstation you put it into.

Automating the build process

Deployment is often a complex job involving many steps. If any of those steps are forgotten or done incorrectly, the user may end up with a faulty installer. That leads to extra support work and even mistrust of your abilities. Getting this part of the application lifecycle right is crucial.

I use a carefully crafted checklist to make sure I do all of the steps necessary to deploy my applications. As they got more complicated, the list got longer and longer and therefore more likely to be error-prone. Automating as many tasks as possible helps to ensure the deployment process is both accurate and efficient.

Many of the tasks involved in deployment involve files: copying, moving, deleting, creating, changing, digitally signing, and so on. Sounds like a perfect job for Windows PowerShell.

BuildSetup.PS1 (**Listing 6**), included with the sample files accompanying this document, signs the main executable for the sample application (MyApp.EXE), builds the WSSetup.EXE installer, and then builds Setup.EXE.

Listing 6. BuildSetup.PS1 helps automate the deployment process.

```
# Taken from http://stackoverflow.com/questions/495618/how-to-normalize-a-path-in-
powershell
function Get-AbsolutePath($Path)
{
    $Path = [System.IO.Path]::Combine( ((pwd).Path), ($Path) );
    $Path = [System.IO.Path]::GetFullPath($Path);
    return $Path;
}

# Find the location of Signtool.
$regEntry = Get-ItemProperty –Path
   "Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SDKs\Windows\v7.1A"
$value = $regEntry.'InstallationFolder'
$file = $value + "Bin\signtool.exe"
$signTool = Get-ChildItem $file

# Get the password for the certificate.
$password = Get-Content ..\..\DontDeploy\certpw.txt
$parms = 'sign', '/f', "$(Get-AbsolutePath('..\..\DontDeploy\mycert.pfx'))", '/p',
   $password

# Sign the exe.
$signParms = $parms + '/d', 'My Application', '..\myapp.exe'
& $signTool $signParms

# Find the location of the Inno Setup compiler.
$regEntry = Get-ItemProperty –Path
   Registry::HKEY_CLASSES_ROOT\InnoSetupScriptFile\Shell\Compile\Command
$value = $regEntry.'(default)'
$file = $value.Substring(1, $value.IndexOf('"', 2) - 1)
$exe = Get-ChildItem $file
$innoCompiler = $exe.DirectoryName + "\iscc"

# Build WSSetup.exe and Setup.exe
$signParms = "/sStandard=$signTool $parms $" + "p"
& $innoCompiler $signParms WSSetup.iss | Out-Null
& $innoCompiler $signParms MyAppSignedWithWS.iss
```

See my “Windows PowerShell: Batch Files on Steroids” white paper for a more extensive script that, in addition to building the setup, uploads it to an FTP site.

You may also want to look at a VFPX tool called Automated Build (http://tinyurl.com/npa95pp). It’s an extension to another tool called CruiseControl.NET that allows you to create a build server for VFP applications. I haven’t tried it but it’s

discussed in detail in the book "VFPX: Open Source Treasure for the VFP Developer," available from http://www.foxrockx.com/GetVFPX.htm.

Installing the .NET framework

Since we're VFP developers, why would we care whether the .NET framework is installed on the user's machine or not? One very good reason is wwDotNetBridge. wwDotNetBridge is a free utility from Rick Strahl that makes calling .NET code from VFP easy. It provides a simple way to add new features to your VFP applications by leveraging the power of the .NET framework. I discussed wwDotNetBridge in detail in a white paper called "Calling .NET Code from VFP the Easy Way" you can download from my web site (http://doughennig.com/papers/default.html).

Obviously, wwDotNetBridge requires that .NET be installed on the user's system. Suppose you want to use a feature in .NET 4 but you don't know whether that version of the framework is installed or not, and certainly don't want to explain to the user how to install it. Instead, we want to make installing the correct version of the .NET framework as seamless as installing any other component.

DotNet2Install.ISS and DotNet45Install.ISS are two very similar Inno Setup scripts. They check whether .NET 2 or 4.5, respectively, is installed and if not, download the installer from the Microsoft web site and run it. They use ISXDL.DLL, included with the download accompanying this document, to perform the .NET download. To use either of these scripts, #INCLUDE it in your application's ISS file.

Automatically updating your application

Some applications detect when a newer version is available and give the user the option to download and install the new version. One technique for doing this is discussed in Rick Strahl's white paper "Automatic Code Updates for Visual FoxPro Applications" (http://tinyurl.com/3grdvt7). Unfortunately, it doesn't work in Windows Vista and later 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:

- 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, which is part of West Wind Web Connection and West Wind Client Tools, for file download. You can use a different FTP library if you wish but unfortunately not Craig Boyd's free VFPConnection.FLL (http://tinyurl.com/333d4b5) because it doesn't support passive FTP which is required to function properly on Windows Server 2008 and later.

- 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 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 an updater application using "RunAs" with ShellExecute. That brings up the User Access Control (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. After using ShellExecute, we terminate the current application so it isn't running anymore.
- The updater 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 the updater application itself or the VFP runtimes, since they're in use, but any other files can be updated.
- After the copying is done, the updater 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.

The source code for all of this except wwFTP.PRG (because that's part of West Wind's commercial tools) is included with this document. Here's how to use this in your own applications:

- Copy CheckUpdate.EXE and UpdateApp.EXE into the application folder. CheckUpdate.EXE checks for an updated version and UpdateApp.EXE is the program that does the actually copying of files into the program folder. Note that UpdateApp.EXE should be digitally signed because otherwise the UAC dialog displays "unknown publisher" when that EXE is run.
- If the files to be downloaded are zipped, also copy VFPCompression.FLL (Craig Boyd's free zipping tool for VFP) into the folder.
- Create a file named Files.XML (you can give it a different name if you wish; pass the name to CheckUpdate as described below) with content like the following:

```
<update>
<version>Version number of update</version>
<minversion>The minimum version number that can be updated</minversion>
<text>Information as formatted HTML for the user about the update e.g. new
features
```

```
</text>
<files>
<file minversion="The minimum version number that can be updated"
maxversion="The maximum version number that can be updated">File name</file>
...
</files>
</update>
```

- The minversion element allows you to specify the minimum version that can be upgraded using this mechanism. If the user has a version older than the minimum, they'll have to download the current installer for the application manually.
- The minversion and maxversion attributes for files allow you to provide different update mechanisms for different versions. For example, suppose version 1.0 is the original application, version 1.1 uses a new ActiveX control, and version 1.2 just has some minor new features. To upgrade from version 1.0 to 1.2 requires installing and registering the new ActiveX control, which is best left to an installer, so if the user is at 1.0, we'll download and run Setup.EXE to install version 1.2. However, to upgrade from version 1.1 to 1.2 just needs a new application EXE, so we'll download and copy MyApp.EXE into the application folder. The following content for Files.xml specifies that:

```
<update>
<version>1.2</version>
<minversion>1.0</minversion>
<text>&lt;base target="_blank"&gt;
&lt;p&gt;&lt;font face="Tahoma" size="2"&gt;&lt;b&gt;My Application Version
1.2&lt;/b&gt;&lt;/p&gt;
There are several new features in this release. Yada yada.
&lt;/font&gt;
</text>
<files>
<file minversion="1.0" maxversion="1.0">setup.exe</file>
<file minversion="1.1">myapp.exe</file>
</files>
</update>
```

- Three types of files are supported in the list of files. If there's only one file to download and it's an EXE file containing "setup" or "update" in the name, that file is executed. For example, because of the minversion and maxversion attributes in the listing above, if the user is running version 1.0, only Setup.EXE is downloaded. If the file has a ZIP extension, its contents are extracted in the application folder. Otherwise, the file is simply copied into the application folder.
- Upload Files.XML and any files listed in it to the appropriate folder on the FTP site used for updates.
- To check if a new version is available, use code like this in your application:

```
do CheckUpdate with main EXE path, Registry key, FTP URL, user name, ;
   password, 'files.xml', .T. to always check
```

The parameters are:

- The full path to the main executable for the application. This is the EXE that is re-run after an update is installed. You can either hard-code it or use something like _vfp.ServerName to make it generic.
- A key for the Windows Registry where the date of the last time an update check was done and the number of days to wait between checks (the default is 1) is stored. If you want to change these values, update the LastUpdate and UpdateDays values in the desired key.
- The URL for the FTP site where update files can be downloaded from, including the folder where the files are located, such as www.mysite.com/Updates.
- The user name and password for the FTP site.
- The name of the file to download containing update information.
- .F. to only check for an update if a check was not done recently and to not display a "connecting to site" dialog (typically used when this code is executed as part of application startup) or .T. to always check and to display such a dialog (usually the case when called from a Check for Updates menu item).

Listing 7 shows the code in CheckForUpdate.PRG, which is called from the main program for the sample application that accompanies this document.

Listing 7. CheckForUpdate.PRG calls CheckUpdate.EXE to see if an update is available.

```
lparameters tlForceCheck
lcSettings = filetostr('..\DontDeploy\Update.txt')
alines(laSettings, lcSettings)
lcKey = 'Software\My Application'
do CheckUpdate with _vfp.ServerName, lcKey, laSettings[1], laSettings[2], ;
   laSettings[3], 'files.xml', tlForceCheck
```

Summary

Given how important deployment is in the application lifecycle, planning for deployment needs to part of the overall development process. This document describes several techniques for deployment that you may not have considered before so hopefully you can use some of these ideas, or even code, in your own application efforts.

Biography

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

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

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox and Southwest Xbase++ conferences (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).