Adding Features to Your VFP Applications Using C#

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

You may have seen sessions at previous Southwest Fox conferences on using wwDotNetBridge to call .NET code from VFP. If you haven't, this session will introduce you to the benefits .NET can add to your VFP applications. However, this session is more from the .NET side. When do you need to write a wrapper program in .NET rather than accessing the .NET code directly? How do you write such a wrapper? This session will also look at a new interop library by James Suárez called Kodnet. This library has some features that aren't available in wwDotNetBridge.

Introduction

The Microsoft .NET framework has a lot of powerful features that aren't available in VFP. For example, dealing with Web Services is really ugly from VFP but is simple in .NET. .NET also provides access to most operating system functions, including functions added in newer versions of the OS. While these functions are also available using the Win32 API, many of them can't be called from VFP because they require callbacks and other features VFP doesn't support, and accessing these functions via .NET is easier anyway.

There are various mechanisms that allow you to access .NET code from VFP applications. For example, my "Creating ActiveX Controls for VFP Using .NET" white paper, available at http://doughennig.com/papers.aspx, shows how to create .NET components that can be used in VFP applications. However, these types of controls suffer from a couple of issues: they have to be registered for COM on the customer's system and there are limitations in working with .NET Interop in VFP that prevent many things from working correctly.

Fortunately, Rick Strahl created a utility called wwDotNetBridge which provides an easy way to call .NET code from VFP. It eliminates all of the COM issues because it loads the .NET runtime host into VFP and runs the .NET code from there.

You may have seen sessions on wwDotNetBridge as Rick Strahl and I have done several. Those sessions were focused on how wwDotNetBridge works and examples of its use. This document is different: it focuses more on the C# point-of-view. We'll start covering the basics of wwDotNetBridge but not delve into detail on how it works as that's covered elsewhere. We'll then look at issues such as when is a wrapper required, design considerations in creating wrappers, dealing with multiple assemblies, debugging C# code called from VFP, handling events, and accessing .NET forms. Finally, we'll look at a new alternative to wwDotNetBridge called Kodnet.

I'm not going to discuss the C# language or how to use Visual Studio. If you're new to C#, see my "Introduction to C# for VFP Developers" white paper at http://doughennig.com/papers.aspx.

Introduction to wwDotNetBridge

There are two versions of wwDotNetBridge:

- The commercial version is included with a couple of Rick's products: West Wind Web Connection and West Wind Internet and Client Tools.
- The open source version is available on GitHub at https://github.com/RickStrahl/wwDotnetBridge.

The commercial version has some features the open source version doesn't (although the open source version was updated in late September 2019 to more closely align with the commercial version), including an SMTP client, SFTP support, and encryption and image management utilities through .NET wrappers. This document focuses on the open source version.

There's a lot of documentation available for wwDotNetBridge:

- The main documentation is a white paper at https://tinyurl.com/y54c8j27. It's a little out of date as new features have been added but is mostly accurate.
- The documentation at https://tinyurl.com/yyksq4gq is for the commercial version but is more complete than the open source version and is useful nonetheless.
- Rick has numerous blog articles expanding on uses of wwDotNetBridge, such as "Calling async/await .NET methods with wwDotnetBridge" (https://tinyurl.com/yxodocks).
- I have a couple of white papers on wwDotNetBridge: "Calling .NET Code from VFP the Easy Way" and "Practical Uses of wwDotNetBridge to Extend Your VFP Applications," both available at http://doughennig.com/papers.aspx.

Getting wwDotNetBridge

As I mentioned in the previous section, the open source version of wwDotNetBridge is available on GitHub: https://github.com/RickStrahl/wwDotnetBridge. You can either clone the repository or download the files by clicking the Clone or Download button and choosing the appropriate option.

Whichever approach you take, the files you actually need to work with are in the Distribution folder:

- ClrHost.dll: a loader for the .NET runtime
- wwDotNetBridge.dll: the .NET assembly proxy and helper
- wwDotNetBridge.prg: a VFP wrapper for wwDotNetBridge.dll

Since wwDotNetBridge.dll is downloaded, you may have to unblock it to prevent an "unable to load Clr instance" error when using wwDotNetBridge. Newer versions of wwDotNetBridge, including the one on GitHub, make this unnecessary; see Rick's blog post at https://tinyurl.com/y2nk7g5g for details.

Other problems may occur using wwDotNetBridge: see http://tinyurl.com/yabovc3k for a discussion of these issues and their solutions.

Deploying wwDotNetBridge

Deploying wwDotNetBridge with your application is straightforward: include wwDotNetBridge.prg in your project so it's built into the EXE and include ClrHost.dll and wwDotNetBridge.dll in the list of files to deploy with your application. There's no need to unblock the DLLs on the user's machine, even if they're from an older version of wwDotNetBridge, if you use an installer as most applications do.

One other file you should deploy is a manifest file, especially if there's a chance the application may be installed on a server. By default, .NET prevents a DLL from being loaded

from a network share, but it can be configured to allow it with a manifest file, which is just a text file with the same name as your top-level EXE but with a .config extension. For example, the manifest file I use with sfquery.exe is named sfquery.exe.config. This file contains the following, which tells .NET to allow loading DLLs from a network share:

```
<configuration>
  <runtime>
      <loadFromRemoteSources enabled="true"/>
  </runtime>
</configuration>
```

Note that by default, wwDotNetBridge requires .NET 4.5.2 or later. This means it works with Windows 7, Windows Server 2008 R2, and later. If you need to support earlier versions of Windows, such as Vista and XP, see the wwDotNetBridge GitHub page for a link to download a version that works with them. Also, although Windows 8 and later include .NET 4.5 (or later), with Windows 7 and Windows Server 2008 R2 you'll need to explicitly install .NET 4.5. You can download the .NET framework from https://tinyurl.com/yxbcnp3f but if you use Inno Setup to create a setup executable for your applications, you can make installing .NET 4.5 part of your installation process. To do so, put DotNetInstall.iss and isxdl.dll from the Deployment folder accompanying this document into your deployment folder (the one where your ISS script is located) and add this to your ISS script:

```
#include "DotNetInstall.iss"
```

When your setup executable runs, the installer checks whether .NET 4.5.2 is already installed or not. If not, it downloads it from the Microsoft web site and installs it on the user's system.

Using wwDotNetBridge

With earlier versions of wwDotNetBridge, I used to instantiate it with:

```
goBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V4')
```

The last parameter tells wwDotNetBridge which version of the .NET runtime to load; this example specifies version 4 (which includes 4.5.2 and later versions). goBridge is a global object (actually defined as PRIVATE in my top-level program) so it's available throughout the application. When I needed to call a wwDotNetBridge method, I used goBridge.SomeMethod().

However, newer versions of wwDotNetBridge have a better way: DO wwDotNetBridge.prg at startup and then call GetwwDotNetBridge() whenever you need to call a wwDotNetBridge method:

```
loBridge = GetwwDotNetBridge()
loBridge.SomeMethod()
```

GetwwDotNetBridge uses a public variable named __DOTNETBRIDGE to cache the instantiated wwDotNetBridge object, only instantiating it if the variable doesn't exist or doesn't contain an object. So, there's no longer a need for a global goBridge object.

.NET assemblies

Microsoft defines assemblies as "the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions" (https://tinyurl.com/y4azyq6o). Basically, a .NET assembly is an EXE or DLL containing one or more classes. Assemblies may be unsigned or signed (sometimes referred to as "strong-named); see https://tinyurl.com/y7l6338b for details.

The .NET framework consists of thousands of classes in hundreds of assemblies. The framework assemblies are installed in a common location called the Global Assembly Cache or GAC. One advantage of the GAC is that it's installed on every machine that has .NET so there's no need to distribute those files to your users, making installation executables much smaller.

In order to use a .NET class, you have to load the assembly the class is located in. wwDotNetBridge automatically loads two assemblies when it starts: mscorlib, the common .NET object runtime library, and System, which contains a lot of the .NET base classes. To load other assemblies, call the LoadAssembly method of wwDotNetBridge, passing it one of the following:

- The full path to the assembly, such as "C:\MyFolder\MyAssembly.dll."
- The assembly name without an extension, such as "System.Windows.Forms." This works in two cases: when the assembly is unsigned and located in the same folder as your application's EXE, and when the assembly is one of the following: System.Xml, System.Data, System.Web, and System.Windows.Forms. In my testing, some other assemblies in the GAC also work, such as System.Drawing.
- The fully qualified assembly name, such as "System.Windows.Forms, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089." This is used for any assemblies loaded from the GAC (other than those wwDotNetBridge already loads or supports) or for strongly signed local assemblies.

  One way to find out the value for PublicKeyToken is use the SN tool that comes with Visual Studio (VS): open a command prompt in VS (see the tip below to add that to your VS menu) and type SN –T *AssemblyName*, where *AssemblyName* is the path to the assembly (note the "-T" must be uppercase). For example:

  ```
  sn –T C:\Windows\Microsoft.NET\Framework64\v4.0.30319\System.Threading.dll
  ```

  displays a public key token of b03f5f7f11d50a3a, so use "System.Threading, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" to load the System.Threading assembly from the GAC.

Another way to get the fully qualified assembly name is to use a .NET disassembling tool: see Rick's blog post at https://tinyurl.com/y3hvdzgn for details.

See https://tinyurl.com/yd99372g for details on adding a command prompt menu item to the Visual Studio menu. If you copy and paste from Luc's blog post, be sure to change the curly quotes to regular ones or you'll get an error like I did until I fixed it.

Rick's blog post at https://tinyurl.com/y35eh8ax has even more options, including specifying Console2 as a replacement for Cmd.exe.

To find where the GAC is located on your machine, open RegEdit and find the InstallPath value in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full.

How do you know what assembly to use? Obviously, if it's a wrapper you or someone on your team created, you'll know the name. Otherwise, check the documentation for the class you want to use. For example, suppose you want to do something with processes. Googling ".net process" finds documentation for the Process class at docs.microsoft.com, shown in **Figure 1**. Since it's in the System.dll assembly, which wwDotNetBridge loads automatically, there's no need to load anything to use the Process class. Otherwise, load the specified assembly.

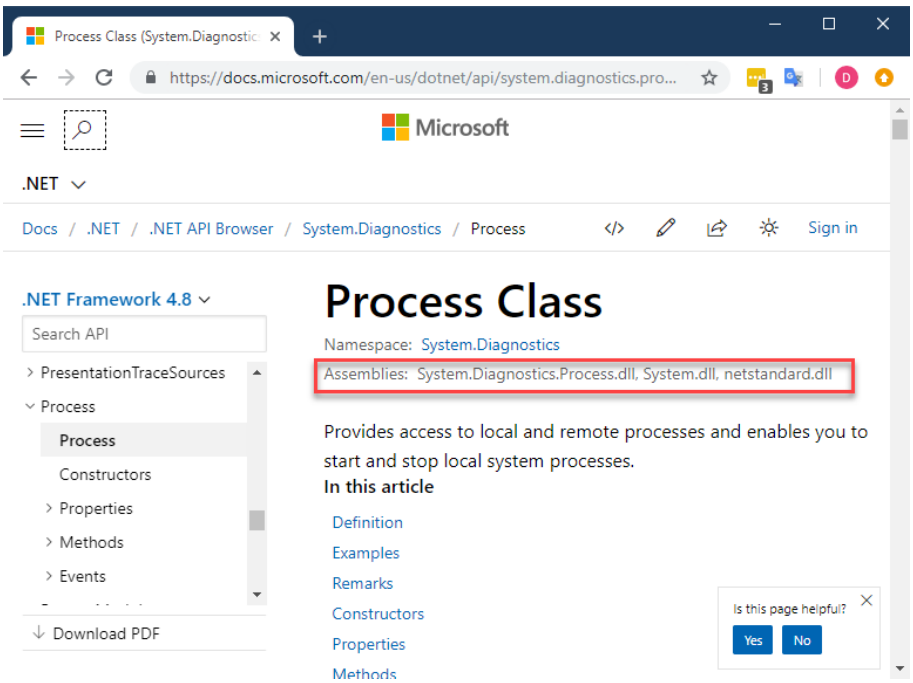

Figure 1. The documentation for the Process class shows the assembly it's located in.

A couple of other points about assemblies:

- You don't have to load assemblies the class you're using has dependencies on; they're loaded automatically. We'll see an example of that later when we create a wrapper that uses the System.Windows.Forms assembly. We have to load the wrapper DLL but not System.Windows.Forms.dll as the latter is loaded automatically because the former has a dependency on it.
- There isn't a way to get a list of loaded assemblies so just call LoadAssembly as required. If an assembly was loaded by an earlier call, it isn't re-loaded so no harm done.

Our first attempt

Although I prefer to log diagnostic information and errors to text and/or DBF files, some people like to log to the Windows Event Log, as system administrators are used to looking there for issues. Doing that from a VFP application is ugly because the Win32 API calls are messy. Let's figure out how to do it using .NET.

Start by Googling ".net windows event log." That leads to the documentation for the EventLog class (https://tinyurl.com/y5cugfet). From there, we learn a few things:

- Since it's in System.dll, we don't have to load any assembly.
- EventLog is in the System.Diagnostics namespace so its full name is System.Diagnostics.EventLog.
- The C# sample shows accessing the class directly for some operations; that is, it uses code like EventLog.SourceExists("MySource") rather than instantiating the class. That means the class has some static methods so we'll call those using wwDotNetBridge's InvokeStaticMethod.
- The sample also shows instantiating the class and using the instance for other operations. We'll use CreateInstance and try to access the members directly if we can.

That's the first lesson: find out what class or classes you need to accomplish a task and find the documentation and hopefully some sample code. That gives you the information you need to get started writing VFP code to use the classes.

The code in **Listing 1**, taken from WindowsEventLog.prg that accompanies this document, shows how to use the EventLog class in VFP.

Listing 1. Writing to the Windows Event Log from VFP.

```
local loBridge, ;
   lcSource, ;
   lcLogType, ;
   lcClass, ;
   loValue, ;
   loEventLog, ;
```

```
    lcEvents, ;
    loEvents, ;
    lnI, ;
    loEvent

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* Define the source and log types.

lcSource  = 'MyApplication'
lcLogType = 'Application'

* Put the name of the .NET class we'll use into a variable so we don't
* have to type it on every method call.

lcClass = 'System.Diagnostics.EventLog'

* See if the source already exists; create it if not.

if not loBridge.InvokeStaticMethod(lcClass, 'SourceExists', lcSource)
    loBridge.InvokeStaticMethod(lcClass, 'CreateEventSource', lcSource, ;
       lcLogType)
endif not loBridge.InvokeStaticMethod ...

* Create an information message.

loBridge.InvokeStaticMethod(lcClass, 'WriteEntry', lcSource, ;
    'Some application event that I want to log')

* For an error message, we need to use an enum.

loValue = loBridge.GetEnumValue('System.Diagnostics.EventLogEntryType.Error')
loBridge.InvokeStaticMethod(lcClass, 'WriteEntry', lcSource, ;
    'Error #1234 occurred', loValue, 4)
    && 4 is a user-defined event ID

* However, that doesn't work in this case because WriteEntry is overloaded, so
* so we'll use this method instead.

loValue = loBridge.CreateComValue()
loValue.SetEnum('System.Diagnostics.EventLogEntryType.Error')
loBridge.InvokeStaticMethod(lcClass, 'WriteEntry', lcSource, ;
    'Error #1234 occurred', loValue, 4)

* Display the last 10 logged events.

loEventLog        = loBridge.CreateInstance(lcClass)
loEventLog.Source = lcSource
loEventLog.Log    = lcLogType
loEvents          = loEventLog.Entries
*loEvents = loBridge.GetPropertyEx(loEventLog, 'Entries')
    && If the access to the collection didn't work, we could use this instead
lcEvents          = ''
for lnI = loEvents.Count - 1 to loEvents.Count - 10 step -1
    loEvent  = loEvents.Item(lnI)
```

```
    lcEvents = lcEvents + transform(lnI) + ': ' + loEvent.Message + chr(13)
next lnI
messagebox('There are ' + transform(loEvents.Count) + ' events:' + ;
    chr(13) + chr(13) + lcEvents)

* Dispose the object (the documentation says to do so).

loEventLog.Dispose()
```

This code in Listing 1 starts by checking whether the event source, which is usually an application name or something equally descriptive, already exists or not; if not, it's created using the specified type (Application, Security, or System; Application in this case). It then calls the WriteEntry method to write to the log. The first call to WriteEntry uses one of the overloads of that method: the one expecting the name of the source and the message. The second call uses a different overload: the one expecting the name of the source, the message, an enum representing the type of entry, and a user-defined event ID (4 in this case). As the comment notes, there's a wrinkle here: passing an enum to WriteEntry fails with an "OLE IDispatch exception code 0 from wwDotNetBridge: Method 'System.Diagnostics.EventLog.WriteEntry' not found..." error. Obviously the method does exist, so in this case, "method not found" means we aren't passing the correct parameters. As the documentation for GetEnumValue (https://tinyurl.com/y3spxqof) points out, when a method is overloaded, passing an enum can fail because GetEnumValue creates an integer value rather than a type of the actual enum. Most times that works but not in this case. Instead, the code creates an enum a different way: using CreateComValue and then SetEnum.

Figure 2 shows how the log entries appear in the Windows Event Viewer.

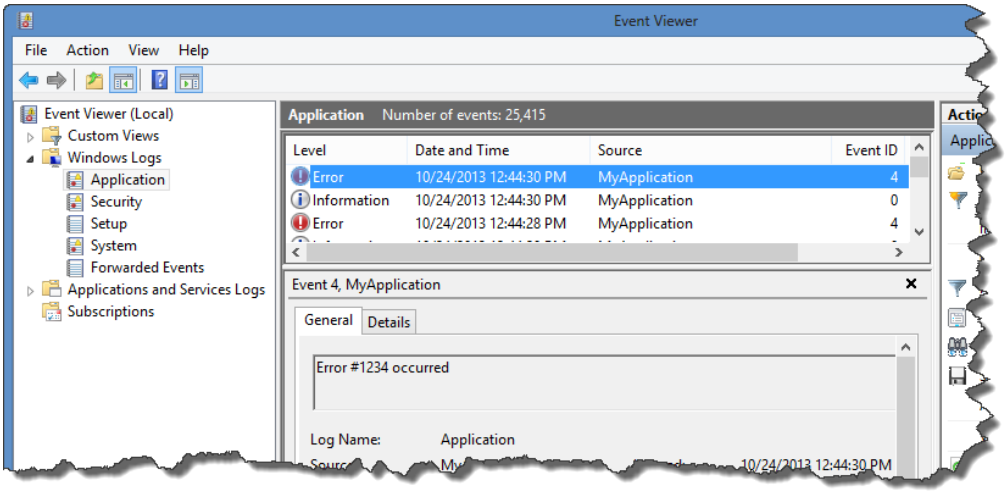

Figure 2. The results of running WindowsEventLog.prg.

To display the log entries, the code creates an instance of System.Diagnostics.EventLog and goes through the entries in the log. There's a comment in the code about the Entries property. It's a collection, which can be an issue in VFP. In this case, it's an implementation of the ICollection interface (I followed the documentation links from the EventLog class

topic to find that out), so its Count and Item methods can be accessed directly. Collections that use generic classes (such as those implementing ICollection<TItem>), however, are problematic because generics aren't marshalled from .NET to VFP. For those, use wwDotNetBridge's GetPropertyEx method to get a reference to the collection; GetPropertyEx is smart enough to return an object which has a Count property and an Item method so you can access the members of the collection.

That's the second lesson: try to access members of the class directly first. If they fail, use indirect access using wwDotNetBridge methods. For properties, use GetPropertyEx to read and SetPropertyEx to write. For methods, use InvokeMethod.

Indirect member access

VFP developers have relied on GETFILE() and PUTFILE() for years to display file selection dialogs. However, these dialogs have several shortcomings:

- You don't have much control over them: you can specify the file extensions, the title bar caption, and a few other options, but these functions hide most of the settings available in the native Windows dialogs. For example, you can't specify a starting folder or default filename for GETFILE().
- There's no access to search in the dialog.
- These functions return file names in uppercase rather than the case the user entered or the file actually uses.
- GETFILE() returns only a single filename even though the native dialogs optionally support selecting multiple files.

The most important issue, however, is that the dialogs displayed are from the Windows XP era and don't support new features available in more modern versions. **Figure 3** shows the dialog presented by GETFILE(). Although this dialog does have quick access buttons at the left, it still looks like a dialog from an older operating system.

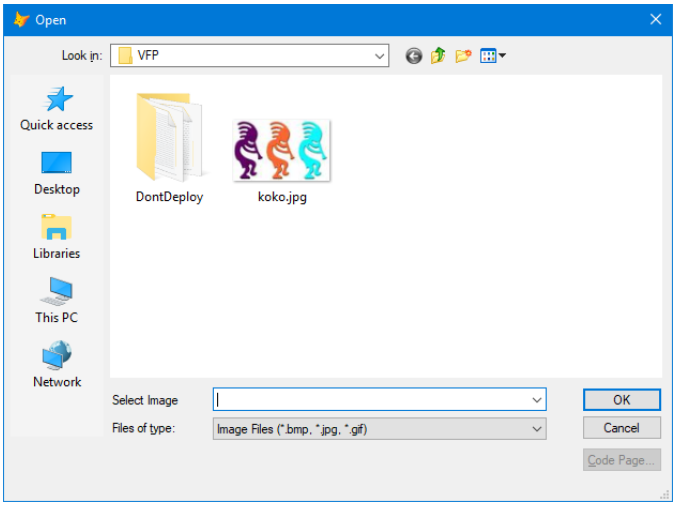

Figure 3. The VFP GETFILE() function is an older-looking dialog.

As you can see in **Figure 4**, the Windows 10 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.

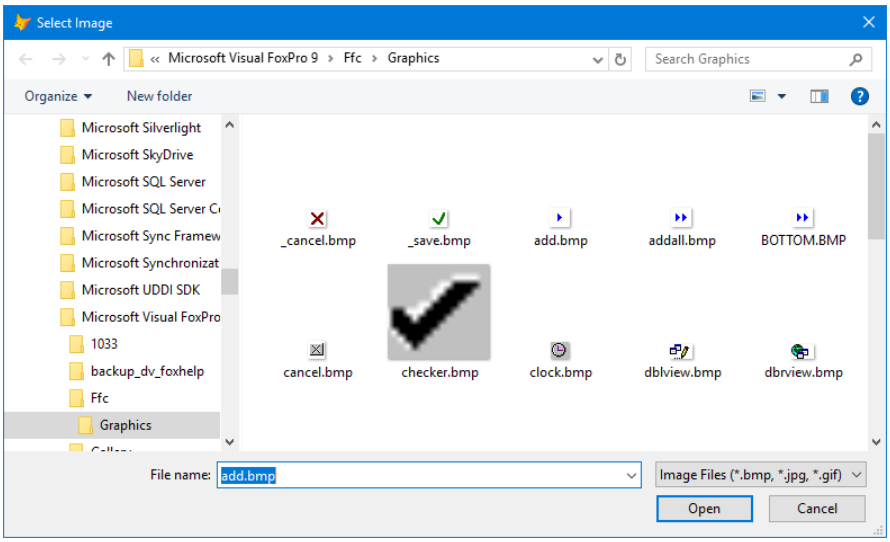

Figure 4. The Windows 10 open file dialog looks modern and has features the older dialog doesn't.

A Google search for ".net open file dialog" leads us to the documentation for the OpenFileDialog class: https://tinyurl.com/y46dhgzf. It looks pretty easy to use, thanks to the sample code provided, so let's create a VFP version that uses it (**Listing 2**).

Listing 2. TestOpenDialog.prg is our first attempt at calling .NET code from VFP using wwDotNetBridge.

```
local lcFile, ;
    loBridge, ;
    loDialog, ;
    lnResult, ;
    lnOK, ;
    lnI

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* Load our assembly and instantiate the OpenFileDialog class. Since
* "System.Windows.Forms" is one of the special assemblies wwDotNetBridge
* recognizes, we don't have to specify its location.

*loBridge.LoadAssembly('C:\Windows\Microsoft.NET\Framework64\v4.0.30319\System.Windows.Fo
rms.dll')
loBridge.LoadAssembly('System.Windows.Forms')
loDialog = loBridge.CreateInstance('System.Windows.Forms.OpenFileDialog')

* Set the dialog properties.

loDialog.FileName         = 'add.bmp'
loDialog.InitialDirectory = home() + 'FFC\Graphics'
```

```
loDialog.Filter           = 'Image Files (*.bmp, *.jpg, *.gif)|*.bmp;' + ;
   '*.jpg;*.gif|All files (*.*)|*.*'
loDialog.Title            = 'Select Image'
loDialog.Multiselect      = .T.

* Display the dialog and get the results. ShowDialog returns an enumeration of
* DialogResult, but that's basically an integer, so we could check for 1
* meaning the user clicked OK. However, it's better to use the actual
* enumeration so we'll get its value using GetEnumValue.

lnResult = loDialog.ShowDialog()
*lnOK     = 1
lnOK     = loBridge.GetEnumValue('System.Windows.Forms.DialogResult', 'OK')
if lnResult = lnOK
   lcFile = ''
   for lnI = 0 to loDialog.FileNames.Length - 1
      lcFile = lcFile + loDialog.FileNames(lnI) + chr(13)
   next lnI
   messagebox('You selected:' + chr(13) + chr(13) + lcFile)
endif lnResult = lnOK
```

Running this code works until it hits the assignment for the FileName property; that results in an "OLE error code 0x80131509: Unknown COM status code." I really don't know why that fails: FileName is a string property so we should be able to write to it directly. However, we also get an error accessing any property and calling any method, so it must be an issue with the class itself.

So, we'll use indirect access. **Listing 3** shows the updated version that works.

Listing 3. TestOpenDialog2.prg is the working version.

```
local loBridge, ;
   loDialog, ;
   lnResult, ;
   lnOK, ;
   lcFile, ;
   loFileNames, ;
   lnI

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* Load our assembly and instantiate the OpenFileDialog class.

loBridge.LoadAssembly('System.Windows.Forms')
loDialog = loBridge.CreateInstance('System.Windows.Forms.OpenFileDialog')

* Set the dialog properties.

loBridge.SetPropertyEx(loDialog, 'FileName', 'add.bmp')
loBridge.SetPropertyEx(loDialog, 'InitialDirectory', home() + 'FFC\Graphics')
loBridge.SetPropertyEx(loDialog, 'Filter', ;
   'Image Files (*.bmp, *.jpg, *.gif)|*.bmp;*.jpg;*.gif|All files (*.*)|*.*')
loBridge.SetPropertyEx(loDialog, 'Title', 'Select Image')
```

```
loBridge.SetPropertyEx(loDialog, 'Multiselect', .T.)

* Display the dialog and get the results.

lnResult = loBridge.InvokeMethod(loDialog, 'ShowDialog')
lnOK     = loBridge.GetEnumValue('System.Windows.Forms.DialogResult', 'OK')
if lnResult = lnOK
    lcFile      = ''
    loFileNames = loBridge.GetPropertyEx(loDialog, 'FileNames')
    for lnI = 0 to loFileNames.Count - 1
        lcFile = lcFile + loFileNames.Item(lnI) + chr(13)
    next lnI
    messagebox('You selected:' + chr(13) + chr(13) + lcFile)
endif lnResult = lnOK
```

The dialog displayed when you run this program is based on the operating system. Under Windows XP, it'll look like an XP dialog. Under Windows 10, it'll look like a Windows 10 dialog.

Creating a .NET wrapper

Using GetPropertyEx or SetPropertyEx for one or two properties isn't bad but it makes for tedious, ugly code if you have to use them a lot like we did in Listing 3. That's one reason for creating a .NET wrapper: making the VFP code easier to write and to read.

Let's create a wrapper in .NET that makes it easy to display a file dialog from VFP. Here are some design decisions:

- It should have properties that represent the same properties of OpenFileDialog: FileName, InitialDirectory, Title, and so on.
- Since we want to replace both GETFILE() and PUTFILE(), let's have ShowOpenDialog and ShowSaveDialog methods rather than having different yet essentially identical classes with ShowDialog methods.
- Rather than having the return value of the Show methods be an indication of whether the user clicked OK or not like the .NET classes, instead let's return the filename (or names) chosen by the user if they click OK or a blank string if not.

A complication with the last point is how do we return multiple filenames if we allow multi-selection and the user chooses more than one file? There are three ways:

- Return a list or collection of names.
- Return an array of names.
- Return a string containing all the names.

As I mentioned earlier, collections can be problematic for VFP, depending on how they're implemented, and arrays can cause similar issues (you need to use GetPropertyEx). The third approach is simple: if we separate the names with carriage returns, we can use

ALINES() in VFP to put the names into an array for processing. Since we want to keep the class simple, let's go with the third approach.

The Dialogs class in Dialogs.cs, shown in **Listing 4**, has ShowOpenDialog and ShowSaveDialog methods that set the properties of the native file dialog based on properties of the class you can set from VFP, such as InitialDirectory and MultiSelect, displays the dialog, and returns the path of the selected files (separated with carriage returns) or blank if the user clicked Cancel.

Listing 4. The Dialogs class provides modern, native file dialogs.

```
using System.Windows.Forms;

namespace FileDialogs
{
    public class Dialogs
    {
        public string DefaultExt { get; set; }
        public string FileName { get; set; }
        public string InitialDirectory { get; set; }
        public string Title { get; set; }
        public string Filter { get; set; }
        public int FilterIndex { get; set; }
        public bool MultiSelect { get; set; }

        public string ShowOpenDialog()
        {
            string fileName = "";
            OpenFileDialog dialog = new OpenFileDialog();
            SetDialogProperties(dialog);
            dialog.Multiselect = MultiSelect;
            if (dialog.ShowDialog() == DialogResult.OK)
            {
                if (dialog.FileNames.Length > 0)
                {
                    foreach (string file in dialog.FileNames)
                    {
                        fileName += file + "\n";
                    }
                }
                else
                {
                    fileName = dialog.FileName;
                }
            }
            else
            {
                fileName = "";
            }
            return fileName;
        }

        public string ShowSaveDialog()
        {
            string fileName;
            SaveFileDialog dialog = new SaveFileDialog();
```

```
            SetDialogProperties(dialog);
            if (dialog.ShowDialog() == DialogResult.OK)
                fileName = dialog.FileName;
            else
                fileName = "";
            return fileName;
        }

        private void SetDialogProperties(FileDialog dialog)
        {
            dialog.FileName = FileName;
            dialog.DefaultExt = DefaultExt;
            dialog.InitialDirectory = InitialDirectory;
            dialog.Title = Title;
            dialog.Filter = Filter;
            dialog.FilterIndex = FilterIndex;
        }
    }
}
```

If you're new to C#, you may be wondering about the parameter for the SetDialogProperties method. Since C# is strongly typed and yet we have to pass two different types to the method (instances of OpenFileDialog and SaveFileDialog, depending on the calling method), the method has to accept something they have in common. In this case, OpenFileDialog and SaveFileDialog are both descendants of FileDialog, which has all the properties they have in common, so that's why although the calling methods pass OpenFileDialog and SaveFileDialog, SetDialogProperties accepts FileDialog.

Listing 5 shows TestOpenDialog3.prg, which uses the Dialogs class. It sets the initial folder to the FFC\Graphics folder in the VFP home directory and turns on MultiSelect so you can select multiple files.

Listing 5. TestOpenDialog3.prg uses Dialogs to display a file dialog.

```
local loBridge, ;
    loDialog, ;
    lcFile

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* Load our assembly and instantiate the Dialogs class.

loBridge.LoadAssembly('FileDialogs.dll')
loDialog = loBridge.CreateInstance('FileDialogs.Dialogs')

* Set the necessary properties, then display the dialog.

loDialog.FileName    = 'add.bmp'
loDialog.InitialDir  = home() + 'FFC\Graphics'
loDialog.Filter      = 'Image Files (*.bmp, *.jpg, *.gif)|' + ;
    '*.bmp;*.jpg;*.gif|All files (*.*)|*.*'
loDialog.Title       = 'Select Image'
```

```
loDialog.MultiSelect = .T.
lcFile               = loDialog.ShowOpenDialog()
if not empty(lcFile)
    messagebox('You selected:' + chr(13) + chr(13) + lcFile)
endif not empty(lcFile)
```

More about wrappers

Let's discuss some housekeeping details about building wrappers.

The first issue is the location of the C# source. I like to put the solution folder at the same level as the VFP source for the application. That way, all the source files are in one general location, which makes backups and managing source control repositories easier.

Speaking of source control, like your VFP source, the C# source should go in source control, so that means adding the solution (.sln), project (.csproj), C# program (.cs), and project properties (AssemblyInfo.cs) files as well as any files referenced by the solution to the repository. You don't need to include the DLLs or supporting VS files since those are built from the source files, so my .hgignore and .gitignore files, which tell Mercurial and Git what to ignore, include these lines:

```
bin/*
obj/*
.vs/*
```

The next issue is where do the assemblies go and how do they get there. I normally put the DLLs in the same folder as the executable but some people like to have a "bin" or similar folder where DLLs and EXEs go. As you likely know, when you build a project or solution in VS, the DLL is created in the bin*mode* subdirectory of the C# project folder, where *mode* is the build configuration (Debug or Release). Rather than manually copying the DLL from the build folder to the folder when the application expects it, I like to specify a build event for the project to do that for me.

Let's look at an example using the FileDialogs wrapper. **Figure 5** shows the folder structure for the sample files for this document.

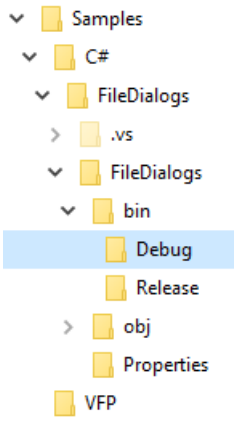

Figure 5. The folder structure for the sample files.

To specify a build event to copy the DLL from Samples\C#\FileDialogs\FileDialogs\bin\Debug to Samples\VFP, double-click Properties in the Solution Explorer, choose the Build Events tab, and enter something like this into *Post-build event command line*:

```
xcopy /y "$(TargetPath)" "$(SolutionDir)..\..\VFP"
```

Rather than hard-coding paths, this command line uses macros. $(TargetPath) contains the path and name of the DLL created by the build process and $(SolutionDir) contains the path for the solution folder. To see the other macros available and their values, click the Edit Post-build button then the Macros >> button. So, "$(SolutionDir)..\..\VFP" means two levels up from the solution folder and then into the VFP folder. All paths are surrounded by quotes in case they contain spaces. Now when I build the solution, the DLL is automatically copied into the VFP folder. Obviously, the final parameter for the XCOPY command will vary depending on your folder structure but the rest of the command is the same.

However, one wrinkle with this is that there can't be any references to the DLL open or the copy fails with a rather cryptic error message: "The command "xcopy (filenames omitted for brevity)" exited with code 4." Since you can't unload an assembly from wwDotNetBridge once it's been loaded, that means exiting VFP or your application before building the solution and then starting it up again afterward.

The final issue is to add the C# DLL to your application's installer. The nice thing about wwDotNetBridge is that no registration is required so it's just another file to include in the installer. Like many VFP developers, I use Inno Setup, so it's as simple as adding a line such as this to the [Files] section of the ISS file:

```
Source: "AssemblyName.dll"; DestDir: "{app}"; Flags: ignoreversion
```

Multiple assemblies

Often, a wrapper is used to wrap the functionality of a .NET library that's hard to use from VFP. In the case of FileDialogs.dll, it wraps some of the functionality in System.Windows.Forms. Since System.Windows.Forms.dll is included in the GAC, there's no need to copy it to the application folder or deploy it to your users. Assemblies that aren't in the GAC, however, do have to be deployed. Let's see an example of that.

Like the VFP GETFILE() and PUTFILE() functions, GETDIR() displays a dialog without a lot of functionality (**Figure 6**). For example, it doesn't provide different views (such as Detail and List) and doesn't allow you to search. However, unlike GETFILE() and PUTFILE(), this isn't the fault of VFP: the .NET FolderBrowserDialog has the same limited functionality.

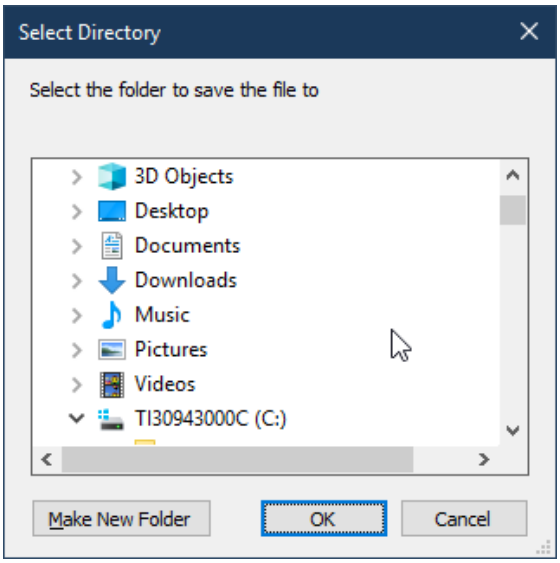

Figure 6. The GETDIR() dialog has limited functionality.

Fortunately, there's a library that has, among a lot of other things, an enhanced folder browser dialog: the Windows API Code Pack. You can read about how to use this dialog in .NET at https://tinyurl.com/y3nlkpja. Rather than creating a separate wrapper for it, it makes sense to include its wrapper in the FileDialogs assembly we created earlier.

After downloading and unzipping the library and adding references to the two DLLs as described in Rod's blog post, add the following code to Dialogs.cs:

```
using Microsoft.WindowsAPICodePack.Dialogs;

public string ShowFolderDialog()
{
    string folder;
    CommonOpenFileDialog dialog = new CommonOpenFileDialog();
    dialog.IsFolderPicker = true;
    if (dialog.ShowDialog() == CommonFileDialogResult.Ok)
    {
        folder = dialog.FileName;
    }
    else
    {
        folder = "";
    }
    return folder;
}
```

Build the solution, then run the program shown in **Listing 6**.

Listing 6. TestFolderBrowser.prg shows how to use the CommonOpenFileDialog class in the Windows API Code Pack.

```
local lcFolder, ;
    loBridge, ;
    loDialog
```

```
* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* Load our assembly and instantiate the Dialogs class.

loBridge.LoadAssembly('FileDialogs.dll')
loDialog = loBridge.CreateInstance('FileDialogs.Dialogs')

* Set the necessary properties, then display the dialog.

loDialog.InitialDirectory = home()
lcFolder                  = loDialog.ShowFolderDialog()
if not empty(lcFolder)
   messagebox('You selected:' + chr(13) + chr(13) + lcFolder)
endif not empty(lcFolder)
```

The following error occurs on the call to loDialog.ShowFolderDialog: "OLE IDispatch exception code 0 from FileDialogs: Could not load file or assembly 'Microsoft.WindowsAPICodePack.Shell (text omitted for brevity). The system cannot find the file specified." Oops, forgot to copy the assemblies FileDialog requires. You could do that manually but adding the following two lines to the post-build event command line takes care of that:

```
xcopy /y "$(TargetDir)Microsoft.WindowsAPICodePack.dll" "$(SolutionDir)..\..\VFP"
xcopy /y "$(TargetDir)Microsoft.WindowsAPICodePack.Shell.dll" "$(SolutionDir)..\..\VFP"
```

Quit VFP, rebuild the solution, and run the VFP code again. This time, you get a much better dialog, shown in **Figure 7**.

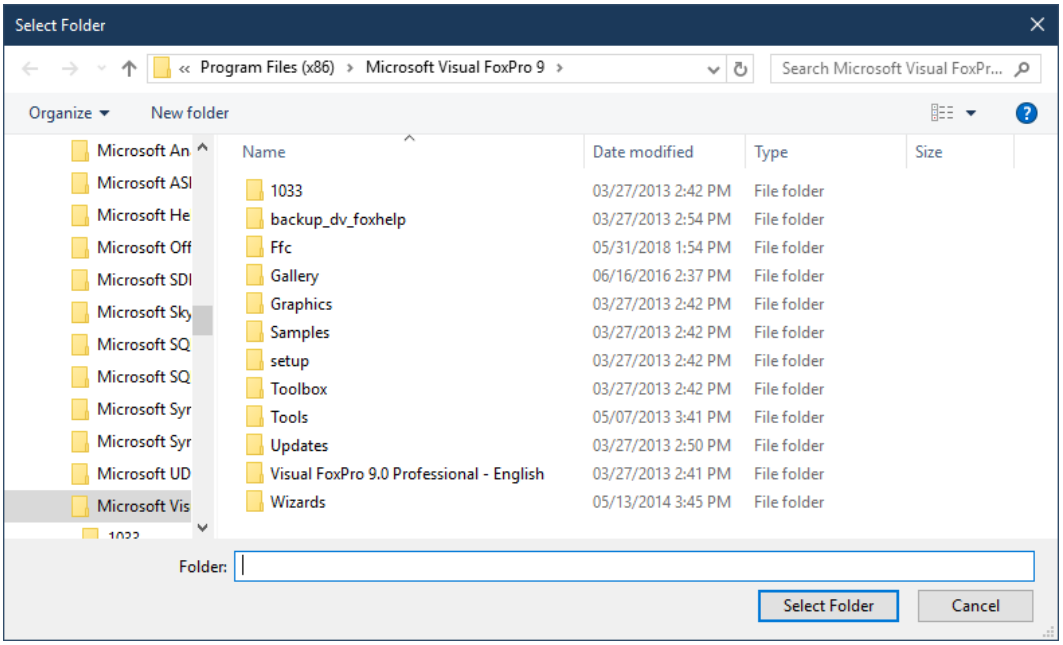

Figure 7. The CommonOpenFileDialog class in the Windows API Code Pack provides a much better folder browser dialog.

Of course, you have to also add the three DLLs—FileDialogs.dll, Microsoft.WindowsAPICodePack.dll, and Microsoft.WindowsAPICodePack.Shell.dll—to your installer so those files are deployed to your users as well.

Debugging .NET code

While the folder browser dialog is much better than GETDIR(), one thing that doesn't appear to work is the initially selected folder: rather than being the VFP HOME() directory as the code instructs, it's some other folder. That means we need to some debugging to find out why.

Debugging COM objects is hard. Typically, you do it the old fashioned way: add statements to display the current value of variables or properties, run the code, see what the values are, fix the code, run it again, and so on.

Debugging .NET code called from VFP via wwDotNetBridge is easy (assuming you have the source code, which we do in this case):

- Open the .NET solution in Visual Studio.
- Set a breakpoint where desired.
- Start VFP.
- Choose Attach to Process from the Visual Studio Debug menu and choose vfp9.exe.
- Execute the VFP code that calls the desired .NET method.

The first time you do this, you may see breakpoint appear as a hollow circle with an exclamation mark and a tooltip of "The breakpoint will not currently be hit." That happens when the DLL hasn't been loaded by wwDotNetBridge yet. The solution is to set a breakpoint in your VFP code just before the call you want to debug, run the VFP code until the breakpoint is hit, then use Attach to Process to attach the VS debugger.

When the breakpoint in the .NET code is hit, the Visual Studio debugger kicks in. You can do all the usual debugging things: step through the code, examine and change the values of properties and variables, skip over code, etc. When you're finished debugging, choose Detach All from the Debug menu.

Note that for debugging to work, you have to do a couple of other things:

- You have to build the solution in Debug configuration.
- In addition to copying the DLL from the bin folder to the application folder, you also have to copy the associated PDB (debugging information) file. You can do this with a post-event command line and simply not deploy the PDF file to your users or you can copy the file manually when needed.

Let's debug the issue with our wrapper. Follow the steps outlined earlier, then run TestFolderBrowser.prg. When the breakpoint is hit in VS, check the value of InitialDirectory by hovering the mouse pointer over it. Note that it contains the correct value: the folder for the VFP home directory. Then hover the mouse pointer over the dialog variable, expand it to see its members, and check the value of its InitialDirectory property. Hmm, it's null. Aha: we didn't set the value of that property to the value of the wrapper's property. Press F5 to finish running the method, exit out of the VFP code, quit VFP, and add this line of code after instantiating CommonOpenFileDialog:

```
dialog.InitialDirectory = InitialDirectory;
```

Rebuild the solution, rerun the VFP code, and notice that the dialog now has the correct initial directory.

Handling events

You can send email from VFP without using a C# wrapper, as **Listing 7** shows. There are a couple of wrinkles: you have to use wwDotNetBridge's InvokeMethod to attach a file and to call the Send method. Other than that, it works well, supporting HTML messages and handling services that require SSL such as Gmail.

Listing 7. TestEmail.prg shows how to send an email from VFP without a C# wrapper.

```
local loBridge, ;
    loCredential, ;
    loMail, ;
    loFrom, ;
    loTo, ;
    loMessage, ;
    loAttachment

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* Set up an SmtpClient object.

loCredential       = loBridge.CreateInstance('System.Net.NetworkCredential', ;
    'doug.o.hennig@gmail.com', filetostr('DontDeploy\password.txt'))
loMail             = loBridge.CreateInstance('System.Net.Mail.SmtpClient')
loMail.Host        = 'smtp.gmail.com'
loMail.Credentials = loCredential
loMail.Port        = 587
loMail.EnableSsl   = .T.

* Set up the message.

loFrom               = loBridge.CreateInstance('System.Net.Mail.MailAddress', ;
    'doug.o.hennig@gmail.com', 'Doug Hennig')
loTo                 = loBridge.CreateInstance('System.Net.Mail.MailAddress', ;
    'doug@doughennig.com')
loMessage            = loBridge.CreateInstance('System.Net.Mail.MailMessage', ;
    loFrom, loTo)
```

```
loMessage.Sender     = loMessage.From
loMessage.IsBodyHtml = .T.
loMessage.Body       = 'This is a test message. ' + ;
    '<strong>This is bold text</strong>. ' + ;
    '<font color="red">This is red text</font>'
loMessage.Subject    = 'Test email'

* Add an attachment.

loAttachment = loBridge.CreateInstance('System.Net.Mail.Attachment', ;
    'koko.jpg')
* This causes VFP to terminate.
*loMessage.Attachments.Add(loAttachment)
loBridge.InvokeMethod(loMessage.Attachments, 'Add', loAttachment)

* Send the message.

* This gives "no such interface supported" error so use InvokeMethod
*loMail.Send(loMessage)
loBridge.InvokeMethod(loMail, 'Send', loMessage)

* Clean up.

loMessage.Dispose()
```

However, what if you want to send a message asynchronously? The SmtpClient object raises an event when the message has been sent but VFP can't handle events from .NET objects, only from COM objects, and the whole point of using wwDotNetBridge is to avoid creating COM objects from .NET code.

So let's create a C# wrapper that makes things simpler for us. Here are the design considerations:

- We don't want to have to create and set properties for six different objects, so let's wrap everything up in one object that has the properties and methods we need to simplify things.
- We'll support multiple To, CC, and BCC addresses by separating them with semi-colons.
- We'll have an AddAttachment method to make it easy to add multiple attachments to the message.
- Rather than using the names of the properties of the .NET objects, let's use names that are a little clearer, such as "MailServer" rather than "Host."
- The wrapper will call methods of a VFP object when the SmtpClient raises its SendCompleted event so our code is notified.

The code in **Listing 8** shows the finished wrapper code.

Listing 8. Smtp.cs is a wrapper to allow VFP applications to send emails via SMTP.

```
using System;
```

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Net;
using System.Net.Mail;

namespace SMTPLibrary
{
    /// <summary>
    /// Sends email via SMTP.
    /// </summary>
    public class SMTP
    {
        /// <summary>
        /// The mail server address.
        /// </summary>
        public string MailServer { get; set; } = "";

        /// <summary>
        /// The port to use.
        /// </summary>
        public int ServerPort { get; set; } = 25;

        /// <summary>
        /// The email address for the sender.
        /// </summary>
        public string SenderEmail { get; set; } = "";

        /// <summary>
        /// The name of the sender.
        /// </summary>
        public string SenderName { get; set; } = "";

        /// <summary>
        /// A semi-colon separated list of recipients.
        /// </summary>
        public string Recipients { get; set; } = "";

        /// <summary>
        /// A semi-colon separated list of CC recipients.
        /// </summary>
        public string CCRecipients { get; set; } = "";

        /// <summary>
        /// A semi-colon separated list of BCC recipients.
        /// </summary>
        public string BCCRecipients { get; set; } = "";

        /// <summary>
        /// The subject.
        /// </summary>
        public string Subject { get; set; } = "";

        /// <summary>
        /// The body of the message.
        /// </summary>
        public string Message { get; set; } = "";

        /// <summary>
```

```
        /// The user name for the server.
        /// </summary>
        public string UserName { get; set; } = "";

        /// <summary>
        /// The password for the server.
        /// </summary>
        public string Password { get; set; } = "";

        /// <summary>
        /// True if the body contains HTML.
        /// </summary>
        public bool UseHtml { get; set; }

        /// <summary>
        /// True to use SSL.
        /// </summary>
        public bool UseSsl { get; set; }

        /// <summary>
        /// The timeout in seconds.
        /// </summary>
        public int Timeout { get; set; } = 30;

        /// <summary>
        /// The object to call back to when the message was sent or cancelled.
        /// </summary>
        public dynamic CallBack { get; set; }

        /// <summary>
        /// A list of attachments.
        /// </summary>
        private List<string> _attachments = new List<string>();

        /// <summary>
        /// The message we'll send.
        /// </summary>
        private MailMessage _message;

        /// <summary>
        /// Handle the send completion event.
        /// </summary>
        /// <param name="sender"
        /// The sender object.
        /// ></param>
        /// <param name="e">
        /// The parameters for the event.
        /// </param>
        private void SendCompletedCallback(object sender, AsyncCompletedEventArgs e)
        {
            // Get the unique identifier for this asynchronous operation.
            string token = (string)e.UserState;

            // Call the callback object is there is one.
            if (CallBack == null)
            {
            }
            else if (e.Cancelled)
```

```
        {
            CallBack.Cancelled(token);
        }
        else if (e.Error != null)
        {
            CallBack.ErrorOccurred(token, e.Error.ToString());
        }
        else
        {
            CallBack.Sent(token);
        }

        // Clean up.
        _message.Dispose();
    }

    /// <summary>
    /// Send the message.
    /// </summary>
    public void SendMail(string ID)
    {
        // Set up the host.
        NetworkCredential basicCredential = new NetworkCredential(UserName,
            Password);
        SmtpClient smtpClient = new SmtpClient();
        smtpClient.Host = MailServer;
        smtpClient.UseDefaultCredentials = false;
        smtpClient.Credentials = basicCredential;
        smtpClient.Timeout = Timeout * 1000;
        smtpClient.Port = ServerPort;
        smtpClient.EnableSsl = UseSsl;

        // Set up the mail message.
        MailMessage message = new MailMessage();
        message.From = new MailAddress(SenderEmail, SenderName);
        message.Sender = message.From;
        message.IsBodyHtml = UseHtml;
        message.Body = Message;
        message.Subject = Subject;

        // Handle addresses.
        foreach (var address in Recipients.Split(new[] { ";" },
            StringSplitOptions.RemoveEmptyEntries))
        {
            message.To.Add(address);
        }
        foreach (var address in CCRecipients.Split(new[] { ";" },
            StringSplitOptions.RemoveEmptyEntries))
        {
            message.CC.Add(address);
        }
        foreach (var address in BCCRecipients.Split(new[] { ";" },
            StringSplitOptions.RemoveEmptyEntries))
        {
            message.Bcc.Add(address);
        }

        // Handle attachments.
```

```
            foreach (string attachment in _attachments)
            {
                message.Attachments.Add(new Attachment(attachment));
            }

            // Set up a handler for the SendCompleted event.
            smtpClient.SendCompleted +=
                new SendCompletedEventHandler(SendCompletedCallback);

            // Send the message.
            _message = message;
            smtpClient.SendAsync(message, ID);
        }

        /// <summary>
        /// Add an attachment.
        /// </summary>
        /// <param name="fileName">
        /// The file name of the attachment.
        /// </param>
        public void AddAttachment(string fileName)
        {
            _attachments.Add(fileName);
        }
    }
}
```

Note that CallBack is defined as Dynamic. That's because as a VFP object, it isn't static so we're telling the compiler to trust us and if the object doesn't have Cancelled, ErrorOccurred, or Sent methods, an error occurs at runtime.

Notice how much simpler the code is in **Listing 9** than in Listing 7: there's only one .NET object to work with. That, of course, is a common benefit of using a wrapper, but the other benefit we get with this one is the ability to receive event notification from the .NET object via a callback object. When you run this code (replacing my email settings with your own, of course), you'll see a message that the code called SendMail and then some time later, a message indicating the results of the send process.

Listing 9. TestEmail2.prg shows how to use the SMTP wrapper class.

```
local loBridge, ;
   loMail

 * Set up wwDotNetBridge.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()
loBridge.LoadAssembly('SMTPLibrary.dll')

* Set the email properties.

loMail             = loBridge.CreateInstance('SMTPLibrary.SMTP')
loMail.MailServer  = 'smtp.gmail.com'
loMail.Username    = 'doug.o.hennig@gmail.com'
loMail.Password    = filetostr('DontDeploy\password.txt')
```

```
loMail.ServerPort  = 587
loMail.SenderEmail = 'doug.o.hennig@gmail.com'
loMail.SenderName  = 'Doug Hennig'
loMail.Recipients  = 'doug@doughennig.com'
loMail.Subject     = 'Test email'
loMail.Message     = 'This is a test message. ' + ;
    '<strong>This is bold text</strong>. ' + ;
    '<font color="red">This is red text</font>'
loMail.UseSsl      = .T.
loMail.UseHtml     = .T.
loMail.AddAttachment('koko.jpg')

* Set up the callback object, send the email, and display a message that we're
* done.

loMail.CallBack = createobject('CallBack')
loMail.SendMail(sys(2015))
messagebox('Called SendMail')

* The callback object.

define class CallBack as Custom
    procedure Cancelled(tcID)
        messagebox('The email ' + tcID + ' was cancelled')
    endproc

    procedure Sent(tcID)
        messagebox('The email ' + tcID + ' was sent')
    endproc

    procedure ErrorOccurred(tcID, tcMessage)
        messagebox('An error occurred sending email ' + tcID + ': ' + ;
            tcMessage)
    endproc
enddefine
```

Although creating a C# wrapper does a lot for us in this case, the latest version of wwDotNetBridge as of this writing, from September 2019, supports handling events thanks to work done by Edward Brey. So if you really want to avoid creating a wrapper, you can, as **Listing 10** shows. This code is essentially the same as Listing 7 with the addition the call to loBridge.SubscribeToEvents, calling SendAsync rather than Send, and the CallBack class definition. As noted in the wwDotNetBridge GitHub home page, the event handler must have an On* function for each event raised by the .NET class; in this case, it's OnSendCompleted. Also note I added an oMessage property to the CallBack class and set it to the .NET MailMessage object so we can properly dispose it once the message has been sent.

Listing 10. TestEmail3.prg shows how to use the event handling feature in the latest wwDotNetBridge.

```
local loBridge, ;
    loCredential, ;
    loMail, ;
    loFrom, ;
    loTo, ;
    loMessage, ;
```

```
    loAttachment, ;
    loCallBack, ;
    loSmtpEventSubscription

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* Set up an SmtpClient object.

set step on
loCredential       = loBridge.CreateInstance('System.Net.NetworkCredential', ;
    'doug.o.hennig@gmail.com', filetostr('DontDeploy\password.txt'))
loMail             = loBridge.CreateInstance('System.Net.Mail.SmtpClient')
loMail.Host        = 'smtp.gmail.com'
loMail.Credentials = loCredential
loMail.Port        = 587
loMail.EnableSsl   = .T.

* Set up the message.

loFrom               = loBridge.CreateInstance('System.Net.Mail.MailAddress', ;
    'doug.o.hennig@gmail.com', 'Doug Hennig')
loTo                 = loBridge.CreateInstance('System.Net.Mail.MailAddress', ;
    'doug@doughennig.com')
loMessage            = loBridge.CreateInstance('System.Net.Mail.MailMessage', ;
    loFrom, loTo)
loMessage.Sender     = loMessage.From
loMessage.IsBodyHtml = .T.
loMessage.Body       = 'This is a test message. ' + ;
    '<strong>This is bold text</strong>. ' + ;
    '<font color="red">This is red text</font>'
loMessage.Subject    = 'Test email'

* Add an attachment.

loAttachment = loBridge.CreateInstance('System.Net.Mail.Attachment', ;
    'koko.jpg')
loBridge.InvokeMethod(loMessage.Attachments, 'Add', loAttachment)

* Set up the event handler.

loCallBack = createobject('CallBack')
loCallBack.oMessage     = loMessage
loSmtpEventSubscription = loBridge.SubscribeToEvents(loMail, loCallback)

* Send the message. We're using SYS(2015) as the message ID.

loBridge.InvokeMethod(loMail, 'SendAsync', loMessage, sys(2015))
messagebox('Called SendAsync')

* Clean up.

* We can't do this here because the message hasn't been sent yet. We'll do it
* in the callback object instead.
* loMessage.Dispose()
```

```
* The callback object.

define class CallBack as Custom
    oMessage = NULL

    procedure OnSendCompleted(toSender, toEventArgs)
        lcID = toEventArgs.UserState
        do case
            case toEventArgs.Cancelled
                messagebox('The email ' + lcID + ' was cancelled')
            case not isnull(toEventArgs.Error)
                messagebox('An error occurred sending email ' + lcID + ': ' + ;
                   toEventArgs.Error.ToString())
            otherwise
                messagebox('The email ' + lcID + ' was sent')
        endcase
        if vartype(This.oMessage) = 'O'
            This.oMessage.Dispose()
        endif vartype(This.oMessage) = 'O'
    endproc
enddefine
```

For another example of event handling, see FileWatcher_Events.prg in the Examples folder of the GitHub repository. One use for this could be an import process: rather than having the user choose an Import File function in your application, they can just drop a file into an Import folder, or it might be an automated process that receives import files via email. The file watcher code could be a separate running EXE that just waits until a new file appears, processes it, and then perhaps displays a popup message using Kevin Ragsdale's Desktop Alerts VFPX project (https://github.com/VFPX/DesktopAlerts) to notify the user that the import is done.

See https://tinyurl.com/yydwl2mu for more information on event handling.

Accessing .NET forms

You may want to display a .NET form from a VFP application. You might, for example, need to use a control that isn't available as an ActiveX control, only as a .NET control, such as the DevExpress Grid control, which has a lot more functionality than the VFP Grid. Or perhaps you're planning to migrate your VFP application to .NET one form at a time so you want the .NET forms to display when the user chooses a menu item.

One of our applications calls a .NET component to retrieve data from a database (we can't use SQL passthrough in VFP since the .NET component contains business rules) and then displays that data in a grid. Since the .NET component returns the data as a DataTable, we have to convert it to a cursor in order to display it in the grid. That process is both slow and problematic because not all .NET data types map to VFP data types. So, we decided to display the DataTable in a .NET form called from our application; no data conversion required.

The sample files accompanying this document contain a .NET solution named Browser. This solution consists of a C# program, BrowseTable.cs, and a Windows Forms class,

Browser, which is just a simple form containing a DataGridView grid. BrowseTable.cs consists of a single method:

```
using System;
using System.Data;

namespace Browser
{
    public class BrowseTable
    {
        public void Show(string caption, DataTable table)
        {
            Browser form = new Browser();
            form.Setup(String.Format("{0} - {1} records", caption, table.Rows.Count),
                table);
            form.Show();
        }
    }
}
```

The Setup method of the form simply sets the Text property (the equivalent of the Caption property of a VFP form) to the passed string and the DataSource property of the grid to the passed DataTable.

Listing 11 shows a program that uses Browser.dll. It retrieves a DataTable containing data from SQL Server (note the comment about how to get a DataTable from a DataSet; this is often necessary when accessing .NET collections) and then passes it to the .NET form (**Figure 8**).

Listing 11. TestBrowser.prg shows how to display a .NET form from VFP.

```
local loBridge, ;
    lcSQL, ;
    lcConnString, ;
    loAdapter, ;
    loDataSet, ;
    loDataTable, ;
    lcTable, ;
    loBrowser

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()
loBridge.LoadAssembly('browser.dll')

* Get the SQL statement and connection string.

lcSQL        = 'select IDCUST, TEXTSNAM, IDGRP, IDNATACCT, DATELASTMN, '+ ;
    'NAMECUST, TEXTSTRE1, TEXTSTRE2, NAMECITY, CODESTTE, CODEPSTL, ' + ;
    'CODECTRY from ARCUS'
lcConnString = 'server=DHENNIGWIN8\LOCALHOST;database=SAMLTD2019;' + ;
    'trusted_connection=yes'

* Retrieve the data.
```

```
loAdapter = loBridge.CreateInstance('System.Data.SqlClient.SqlDataAdapter', ;
   lcSQL, lcConnString)
loDataSet = loBridge.CreateInstance('System.Data.DataSet')
loBridge.InvokeMethod(loAdapter, 'Fill', loDataSet)

* Get the table and display the .NET form.

loDataTable = loBridge.GetPropertyEx(loDataSet, 'Tables[0]')
lcTable     = strextract(lcSQL, 'from ')
loBrowser   = loBridge.CreateInstance('Browser.BrowseTable')
loBridge.InvokeMethod(loBrowser, 'Show', lcTable, loDataTable)
```

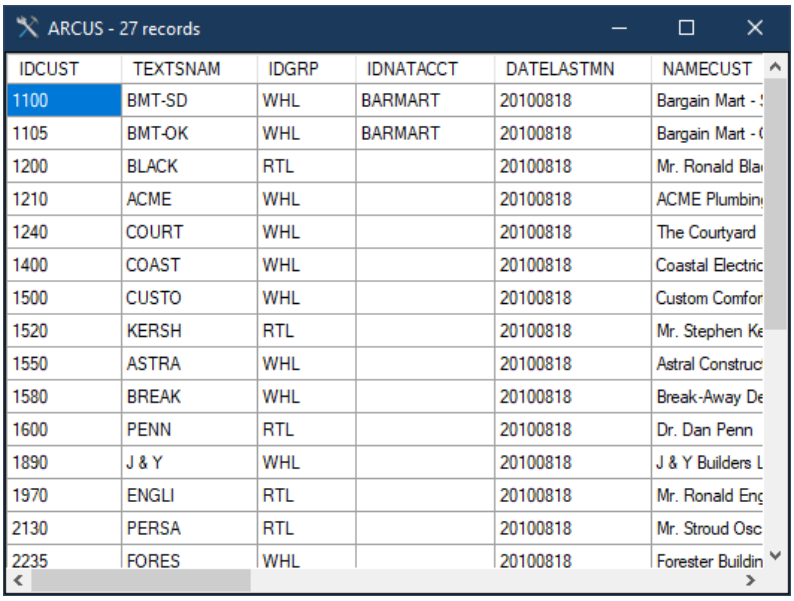

Figure 8. Browser is a .NET form called from VFP.

Something interesting happens when you run TestBrowser.prg: although the variable containing the reference to the object goes out of scope when the code finishes, the form stays open and can't be closed programmatically (not even using CLEAR ALL), only by terminating the application or clicking the close box for the form. I'm not really sure how that happens; the object reference to the .NET form must be stored somewhere but I don't know where.

Given the lack of control over a modeless form and potential conflicts between the event loops in VFP and .NET, you may be better off working with modal forms.

Kodnet

Early in 2019, James Suárez of Ecuador announced the availability of a tool similar to wwDotNetBridge named Kodnet (https://github.com/voxsoftware/kodnet). James claims Kodnet has the following advantages over wwDotNetBridge:

- Code using Kodnet is easier to write because you can access members directly.
- Kodnet supports delegates and events.

- You can create generic class instances.
- It has support for asynchronous .NET methods.
- It can compile C# code dynamically, which can be used for things like runtime scripting.
- It has support for including .NET controls in VFP forms.
- It's up to 8 times faster than wwDotNetBridge.

As of this writing, documentation for Kodnet is sparse but there are a few sample files. Let's look at it in detail and compare it to wwDotNetBridge.

Deploying Kodnet

Like wwDotNetBridge, you include a PRG (Kodnet.prg in this case) in your project so it's built into the EXE. If you want to add .NET controls to your forms, you also need to include DotNet4.vcx in your project.

In addition, there are some other files that come with Kodnet you have to (or not, as you'll see) deploy:

- ClrHost.dll: This is required and appears to be the same file that comes with wwDotNetBridge. It should be in the program folder.
- jxshell.dotnet4.dll and jxshellbase.dll: These are required and must be in the Lib subdirectory of the program folder. However, you can edit Kodnet.prg and change the assignment of dotnet4.libPath to whatever location you want, such as:

  ```
  dotnet4.libPath = dotnet4.path
  ```

  to specify the program folder.

- Microsoft.CSharp.resources.dll: This appears to be optional as deleting it did not cause any problems in my tests. I suspected it's needed for dynamic compilation of C# code but the CompileCSharp.prg sample that comes with Kodnet works correctly without this DLL.
- mscorlib.resources.dll, lib.dll, VFP9resn, and VFP9t.dll: I'm not sure what these are needed for as all samples ran even with these files missing.

Using Kodnet

Like wwDotNetBridge, you start by running a program to set things up: Kodnet.prg. It adds several properties to _screen, but some of them contain the same objects, so they're really pseudonyms. The two objects you'll use are _screen.Kodnet (duplicated in _screen.__DotNet4 and _screen.DotNet4) and _screen.KodnetManager (duplicated in _screen.DotNet4Manager).

Like wwDotNetBridge, Kodnet automatically loads mscorlib and System. Load other assemblies you need using _screen.Kodnet.LoadAssemblyFile. Unlike wwDotNetBridge, you have to specify a full path, even for assemblies in the current folder. This doesn't work:

```
_screen.Kodnet.LoadAssemblyFile('Browser.dll')
```

but this does:

```
_screen.Kodnet.LoadAssemblyFile(fullpath('Browser.dll'))
```

For assemblies in the GAC, you can use LoadAssemblyPartialName without a path or extension:

```
_screen.Kodnet.LoadAssemblyPartialName('System.Data')
```

I'm not sure if this works with all GAC assemblies or just certain ones as is the case with wwDotNetBridge. Also, this doesn't work with unsigned assemblies in the current folder like it does with wwDotNetBridge; the following fails:

```
_screen.Kodnet.LoadAssemblyPartialName('Browser')
```

Unlike wwDotNetBridge, Kodnet doesn't automatically load System.Data.dll. **Figure 9** shows the error message displayed when I used this code:

```
do Kodnet
loAdapterClass = screen.Kodnet.GetStaticWrapper('System.Data.SqlClient.SqlDataAdapter')
```

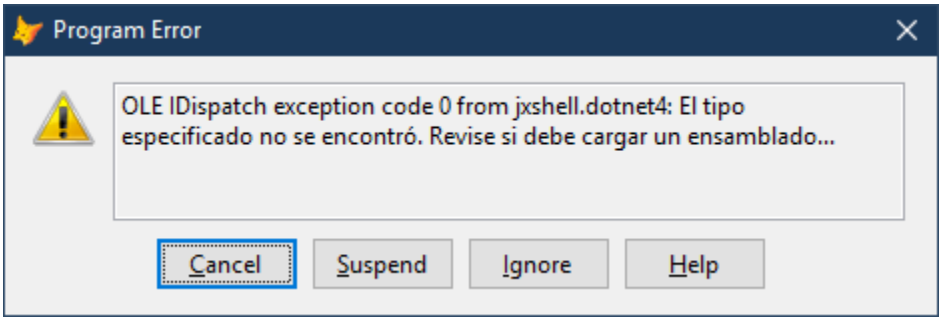

Figure 9. Error messages from Kodnet are in Spanish.

This brings up an issue with Kodnet: error messages are in Spanish. Fortunately, Google Translate is my friend and told me the message in English is "The specified type was not found. Check if you must load an assembly" (which is what I suspected it would be). Loading System.Data takes care of that.

To instantiate a class, use GetStaticWrapper to create a wrapper for the class, then call Construct on the returned object:

```
loDataSetClass = loBridge.GetStaticWrapper('System.Data.DataSet')
loDataSet      = loDataSetClass.Construct()
```

If you don't need a reference to the class for anything else (and there's no documentation about why you would), you can shorten this to:

```
loDataSet = loBridge.GetStaticWrapper('System.Data.DataSet').Construct()
```

Unlike wwDotNetBridge, you can access members of the object returned by Construct directly rather than indirectly using GetPropertyEx, SetPropertyEx, or InvokeMethod. Compare the code in **Listing 12** with that in Listing 11; notice that loAdapter.Fill, loDataSet.Tables.Item, and loBrowser.Show are called directly, so the code is simpler and easier to read.

Listing 12. Code using Kodnet can be simpler than that using wwDotNetBridge.

```
local loBridge, ;
   lcSQL, ;
   lcConnString, ;
   loAdapter, ;
   loDataSet, ;
   loDataTable, ;
   lcTable, ;
   loBrowser

* Create the Kodnet object.

do Kodnet
loBridge = _screen.Kodnet

* Load the assemblies we need.

loBridge.LoadAssemblyPartialName('System.Data')
loBridge.LoadAssemblyFile(fullpath('Browser.dll'))

* Get the SQL statement and connection string.

lcSQL        = 'select IDCUST, TEXTSNAM, IDGRP, IDNATACCT, DATELASTMN, '+ ;
   'NAMECUST, TEXTSTRE1, TEXTSTRE2, NAMECITY, CODESTTE, CODEPSTL, ' + ;
   'CODECTRY from ARCUS'
lcConnString = 'server=DHENNIGWIN8\LOCALHOST;database=SAMLTD2019;' + ;
   'trusted_connection=yes'

* Retrieve the data.

loAdapter = loBridge.GetStaticWrapper('System.Data.SqlClient.SqlDataAdapter')
   .Construct(lcSQL, lcConnString)
loDataSet = loBridge.GetStaticWrapper('System.Data.DataSet').Construct()
loAdapter.Fill(loDataSet)

* Get the table.

loDataTable = loDataSet.Tables.Item(0)

* Display the .NET form.

lcTable   = strextract(lcSQL, 'from ')
loBrowser = loBridge.GetStaticWrapper('Browser.BrowseTable').Construct()
loBrowser.Show(lcTable, loDataTable)
```

That being said, there can be some complications accessing members directly, as the comments in **Listing 13** show.

Listing 13. TestOpenDialog.prg shows that there can be complications accessing members directly.

```
local loBridge, ;
   loDialog, ;
   loResult, ;
   loEnum, ;
   lcFile, ;
   loList, ;
   lnI

* Create the wwDotNetBridge object.

do Kodnet
loBridge = _screen.Kodnet

* Load the assembly we need and instantiate the OpenFileDialog class.

loBridge.LoadAssemblyPartialName('System.Windows.Forms')
loDialog = loBridge.GetStaticWrapper('System.Windows.Forms.OpenFileDialog').Construct()

* Set the dialog properties.

loDialog.FileName         = 'add.bmp'
loDialog.InitialDirectory = home() + 'FFC\Graphics'
loDialog.Filter           = 'Image Files (*.bmp, *.jpg, *.gif)|*.bmp;' + ;
    '*.jpg;*.gif|All files (*.*)|*.*'
loDialog.Title            = 'Select Image'
loDialog.Multiselect      = .T.

* Display the dialog and get the results.

loResult = loDialog.ShowDialog()
loEnum   = loBridge.GetStaticWrapper('System.Windows.Forms.DialogResult')
* Direct comparison of the enums doesn't appear to work but CompareTo does
*if loResult = loResultEnum.OK
*if loResult.Equals(loResultEnum.OK)
if loResult.CompareTo(loEnum.OK) = 0
   lcFile = ''
* We can't use this code: FileNames(lnI) gives "invalid number of parameters"
* while FileNames.Item(lnI) gives "unknown name":
*  for lnI = 0 to loDialog.FileNames.Length - 1
*     lcFile = lcFile + loDialog.FileNames(lnI) + chr(13)
*     lcFile = lcFile + loDialog.FileNames.Item(lnI) + chr(13)
*  next lnI
* We can't use this code either: gives "property FILENAMES is not found"
*  for each lcFileName in loDialog.FileNames
*     lcFile = lcFile + lcFileName + chr(13)
*  next lcFileName
   loList = loBridge.GetStaticWrapper('System.Collections.ArrayList').
      Construct(loDialog.FileNames)
   for lnI = 0 to loList.Count - 1
      lcFile = lcFile + loList.Item(lnI) + chr(13)
   next lnI
   messagebox('You selected:' + chr(13) + chr(13) + lcFile)
```

```
endif loResult.CompareTo(loEnum.OK) = 0
```

Unlike Listing 2, in which setting properties and calling methods of OpenFileDialog directly caused errors, direct access to members using Kodnet works. However, two issues I found were:

- Comparing enums using "=" or the Equals method fails (no error, but the comparison fails). I had to use the Compare method instead.
- Handling arrays is a little complex: you can't reference elements in the array using an index like FileNames[lnI], you can't call an Item method because it doesn't exist, and the VFP FOR EACH syntax doesn't work on a .NET array. Instead, create an ArrayList or some other collection, copy the array to the collection, and then use the Count property and Item method of the collection to access the items in the collection.

To use a .NET control in a VFP form, drop an instance of DotNet4Control in DotNet4.vcx (or a subclass of it) onto a form and set DotNetClassName to the fully qualified name of the control. Webcam.prg and its supporting files (Webcam.scx, Webcam.vcx, Helper.prg, and several .NET DLLs) show how to use a .NET System.Windows.Forms.PictureBox control to display frames from your webcam. See **Figure 10** for the results of running this form.

Figure 10. Kodnet allows you to use .NET controls in a VFP form, such as a PictureBox control displaying frames from a webcam.

Samples

In addition to the webcam sample, Kodnet comes with the following samples:

- CompileCSharp.prg: shows how to compile C# code dynamically at runtime. This may not be something you'll need, but it would certainly be useful to me as we allow

our users to script behaviors in Stonefield Query using C# and getting that to function properly was a lot of work.

- Delegate.prg: shows how to use .NET delegates and generic types in VFP.
- DownloadFileAsync.prg: shows how to call an asynchronous method.
- MD5.prg: shows how to access static members of a class.
- OutlookInterop.prg: shows how to use the Microsoft.Office.Outlook.Interop library.
- X509Certificates.prg: shows how to work with enums and collections.

Summary

Kodnet looks like a very interesting alternative to wwDotNetBridge, especially if you haven't invested time and code in wwDotNetBridge yet. Based on my testing, it looks like there's less need to create C# wrappers with Kodnet. However, until its documentation is more complete, it's hard to recommend this for production applications.

Summary

wwDotNetBridge (and Kodnet once it's a little more complete) makes it easy to call .NET code from VFP. This means you can use .NET to accomplish tasks difficult to do in VFP and easily call them in your applications to add new capabilities. Sometimes you need to write a wrapper in .NET, either to make it easier to write code in VFP or because it's difficult to do what you need with pure VFP code. This document shared some design issues to consider when creating wrappers and looked at several best practices for deployment. I hope you find wwDotNetBridge as important in your VFP development as I have.

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. He wrote over 100 articles in 10 years for FoxRockX and FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe.

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.org). 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).