This paper was originally presented at the Southwest Fox conference in Tempe, Arizona in October, 2006.

Automating the Build

Rick Borup
Information Technology Associates
701 Devonshire Drive, Suite 127
Champaign, IL 61820
Email: rborup@ita-software.com
Blog: http://rickborup.com/blog/

Overview

Compiling your VFP project into an EXE is only the first step in the deployment process. How do you get from there to a complete setup package that's ready to deploy? Do you have an organized and repeatable process to ensure a successful build? To what extent is that process automated? This session explores ways to automate the build, from partially automated methods using popular utilities to more fully automated methods using specialized software such as FinalBuilder and Visual Build Pro.

What is a build?

A build is the process of compiling the source code, gathering up all the required distributable files, and creating the deployment package[1] for an application.

The process generally begins with the building of the application's main executable file and other files that require compilation from source code. It usually finishes with the building of a distributable deployment package using a setup authoring tool such as Inno Setup, InstallShield, or Wise. If your development environment includes the use of version control software[2], your build process also encompasses any necessary interactions with the source code repository.

A build is the process of creating the deployment package for an application, starting from its source code.

The specifics of the build depend on how your development environment is configured. In the simplest case, all the files might reside on a single machine (even if source control is being used) and all the steps in the build process might take place on that same machine. In a team development environment or even a more sophisticated environment for an individual developer, the source code repository is likely to be on a server, each developer likely has her own machine, and there may even be a separate machine dedicated do doing nothing but the build.

Every developer probably has his or her own preferred way of configuring the development and deployment directories on their own machine(s) and/or network. This may be a matter of individual preference, or it may be determined by company or team requirements. Regardless of personal preferences or company standards, though, some things are common to all Visual FoxPro development environments.

In Visual FoxPro, each application has its own project directory. This is where the VFP project file and other project-specific source code files reside, and is typically located on your local machine. If you're not using version control software, this may be the only place (other than backups) where the project file and source code files reside. If you are using version control software, the project directory on your local machine is probably the "working directory" where you do your development work after checking source code files out of the repository.

When it comes time to compile the application into an EXE, you can of course do so directly from your local project directory. This is typically the case when source control software is not being used. If source control software is being used, you can still build the EXE from your local project directory, but you would first want to check out all source code files into your local working directory to be sure you had the latest copy, especially if you're working with other

[1] The term 'deployment package' is used to mean the file or files that get distributed to the end user to install the product. The deployment package could be a single Setup.exe, a CD-ROM containing several files, or even an installation package designed for the Web. Regardless of its physical nature, conceptually it's all the same thing.

[2] I tend to use the terms "version control software" and "source control software" interchangeably. In both cases, I'm referring to software such as Visual SourceSafe, SourceGear Vault, Subversion, and other software packages whose purpose is to manage source code across multiple versions and developers.

developers on the same application. Another approach is to check out all the source code files out into a temporary directory and compile the project from there.

However you approach it, though, compiling the application's main executable and its other distributable files such as DLL's is just the first step in the overall build process.

The EXE itself is usually only one of several files you need to distribute to the end user. You also need to distribute any other files required at runtime but not included (embedded) in the EXE. This can include a readme file, a Help file, an INI file, the VFP report files if they're external to the EXE, and so on.

The EXE and other distributable files mentioned above usually start out in your development directory, but your development directory also has lots of other stuff in it that you don't need to distribute. A good example is the source code files themselves.

The set of distributable files from your development directory represent one set of the total files required for building a release, but other files are required, too.

In *Best Practices for Deployment*[3], I suggest one best practice is to create a deployment directory from which you build the deployment package. The idea of a deployment directory is to create a place that contains the latest versions of all distributable files, and only the distributable files.

Some but not all of the files in the deployment directory change from build to build. Files that typically change for each build include the application's EXE, readme (version history) file, and possibly other files from the application's project directory. These files need to be copied from the development directory to the deployment directory before building the deployment package.

Other files typically do not change from build to build, or at least not for every build. Examples include empty data files, meta data files, etc. These files can be placed in the deployment directory on a semi-permanent basis, and are updated only occasionally, if ever.

What is an automated build?

An automated build is a standardized, reliable, and repeatable way of running the build process with little or no manual intervention.

It's important to understand that an automated build performs the same steps as a manual build. The difference is that it automates some or all of the steps you would otherwise have to perform manually. Before you can begin to automate your build process, you need to start by enumerating the steps you use in your manual build process.

Steps in the build process

As you begin the process of automating your builds, the first requirement is to formally enumerate the steps you use to perform a build. This may sound simplistic, but until you have documented the steps you perform and the order in which you do them, you are not ready to tackle any form of automation.

[3] Visual FoxPro Best Practices for the Next Ten Years, Hentzenwerke Publishing, 2006 – Chapter 13

If you're an independent developer who works primarily on straightforward VFP apps, you may be tempted to say, "There are only two steps: compile the EXE and build the Setup! What's to write down?"

While those are in fact the two primary tasks in any build process, it's usually not all that simple. Take time to think about all the things you actually do within those two tasks: setting build options in the VFP project, updating version numbers, tweaking the setup script, managing GUIDs for MSI setups, etc. Beyond that, do you use source control software? If so, how do you handle checking in and checking out files from the repository for a build? What tool or tools do you use to create the deployment package? And once it's built, how do you deploy the completed setup package? Do you create a CD or copy it to an FTP site? Do you make a backup of all source and deployment files for each build, and if so, how?

In other words, even a simple build process usually involves more than first meets the eye, and each of these steps is a candidate for automation.

Without version control

For purposes of this paper, I'm going to discuss a simplified build process. If version control software is not being used, that process involves only three main steps:

1. Build the EXE and any other components such as DLLs that require compilation from the project source code
2. Copy the new EXE, DLLs, etc. into the deployment directory
3. Build the deployment package from the deployment directory.

There are or course sub-tasks to perform within these three steps, such as updating a Help file or a Readme file, updating the setup script for the new version, etc. But if you're not using version control software, these three steps are essentially all there is to it.

With version control

If you are using version control software, there are a couple of additional steps to perform on the front and back end of the three steps above:

1. Check in your latest changes to the source code repository
2. Retrieve all the source code files from the repository[4]
3. Build the EXE from the project source code
4. Copy the new EXE, DLLs, etc. into a deployment directory along with the other distributable files
5. Build the deployment package from the deployment directory

[4] You might choose to fetch source code files from the repository directly into your development directory, or you might prefer to fetch them into a temporary directory whose only purpose is to support the building of the application's EXE. The process is functionally equivalent either way.

6. Check in the updated setup script or project file, along with any other files that belong in the repository

Steps 1 and 2 comprise whatever you need to do to make sure all the most recent source code changes are included in the build. If you're working as a solo developer and you're building your EXE from your working (development) directory, you may not need to perform step 2 because your working directory is already up to date. On the other hand, if you're a member of a team—or even if you are a solo developer using a more formalized build process—then step 2 may involve fetching all current source code files from the repository into a temporary directory that's used only for compiling the EXE. This approach keeps the files used in the build separate from the files in your working directory, and for that reason may be considered a safer way of doing things.

Steps 3, 4, and 5 are the same three core steps mentioned above. Again, they also comprise whatever sub-tasks are required to produce all the necessary distributable files.

Step 6 is optional, but when the build process is complete it's likely you're going to want to check in to the repository at least some of the updated files produced by or used in the build process. I recommend at a minimum checking in the current setup script (Inno Setup script, InstallShield or Wise project file, or whatever) because you're going to need it if you ever want to re-do the same build. You may also want to check in a copy of the new EXE, DLLs, compiled Help file and its source files, and so on, but it's a matter of some debate whether or not binary files and generated files should be included in a version control repository.

The sample app

In order to have something to work with, I created a sample Visual FoxPro application called myVFPApp. It's an overly simplistic app—in fact, all the EXE does is display "Hello, Southwest Fox!"—but it has enough parts to serve as an example for automating the build.

The distributable files of the sample application are:

- myVFPApp.EXE – the program executable file
- myVFPApp.CHM – the Help file
- Readme.TXT – the version Readme file
- rptCustomers.FRX/FRT – an external report (not embedded in the EXE)
- myDatabase.DBD/DCT/DCX – the VFP database container files
- Customers.DBF/CDX/FPT – a Customers table, initialized for first use

The sample app's directory structure

The build process I'm using as an example here involves only a development directory and a deployment directory. I'm deliberately leaving version control and a source code repository out of the picture for the sake of simplicity.

The development directory is where the VFP project file and source code files reside. This is where you do all your development work, and also where you compile the EXE.

The deployment directory, on the other hand, is a place to put the updated distributable files. It is the source for the files the setup authoring tool uses to build the deployment package. Use of a deployment directory is optional, but I believe it represents a best practice because it keeps the distributable files separate from the development files and thereby helps avoid unintentional changes or corruption that might occur during development work.

The physical directory names I'm using for the sample app are shown below. These directories are referenced in the examples to follow, and are presented here so you'll know what they represent when you see them later on.

- The development directory is `C:\SWFox2006\VFP9Apps\myVFPApp`. It has a conventional set of sub-directories like Data, Help, and Reports.
- The deployment directory is `C:\SWFox2006\VFP9Distrib\myVFPApp`. It has sub-directories identical to the development directory, but only those containing files needed for deployment.
- Other files pulled in by the setup compiler come from `C:\SWFox2006\VFP9Distrib\System32` and `C:\SWFox2006\VFP9Distrib\ActiveX`.

The VFP 9 runtime support library files come from their installed location in `\Program Files\Common Files\Microsoft Shared\VFP`.

The build process illustrated

The build process I'm using here follows the three steps enumerated above when version control is not being used. The illustration in **Figure 1** should help you conceptualize the process.

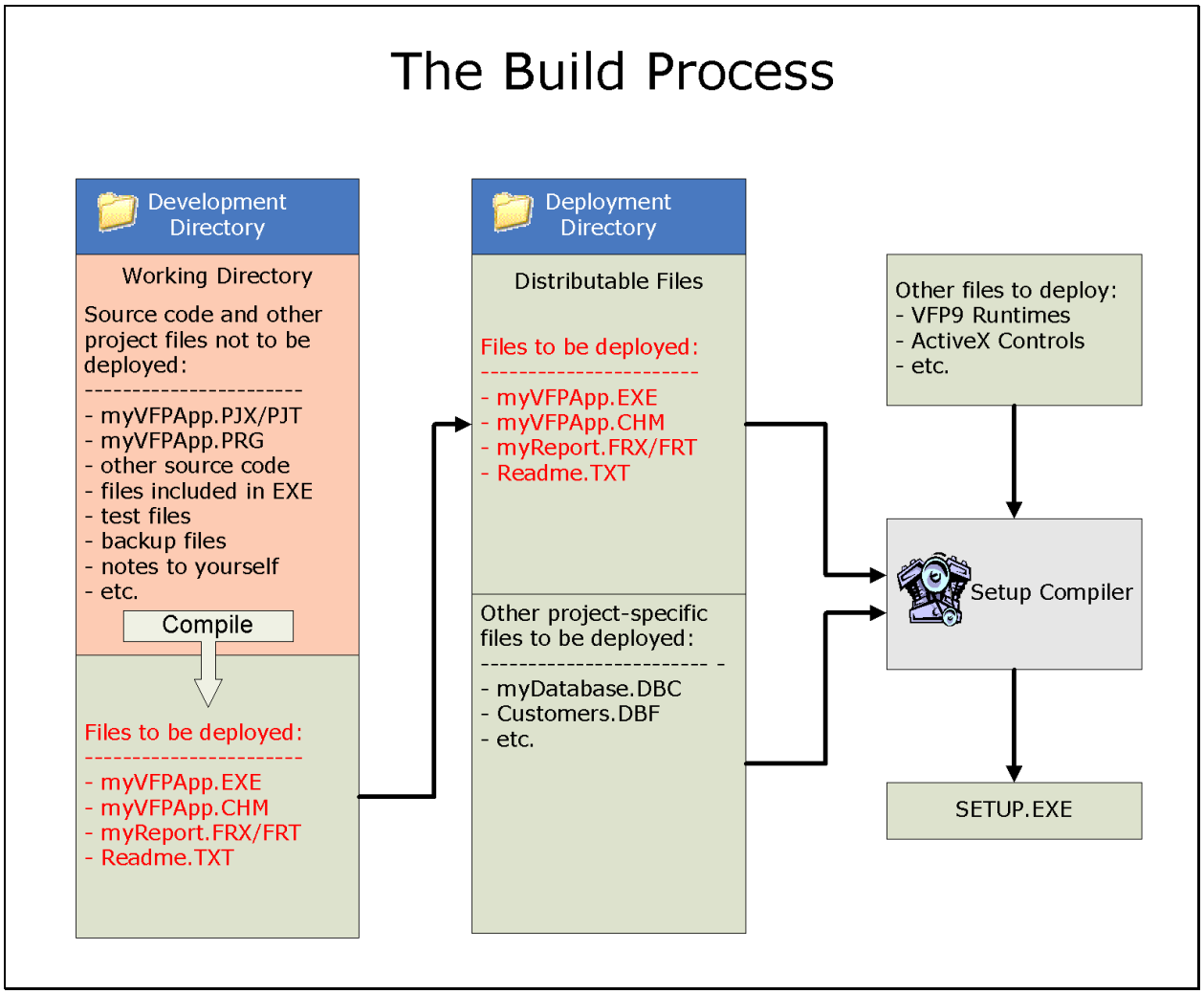

***Figure 1**: The build process copies the updated EXE and other files from the development directory to the deployment directory, then runs the setup compiler to produce Setup.EXE.*

Looking at Figure 1, you can see that the development directory is where the EXE and other distributable files are first created. This directory also includes other files that do not get distributed to the end user, such as the VFP project file, source code files, test files, backup files, temporary files, and so on. Some but not all of the files in the development directory are distributable. Those that are get copied to the deployment directory as part of the build process.

After being updated for a new release, the deployment directory contains a copy of the current distributable files from the development directory. These file are typically updated for each and every build. The deployment directory also contains other project-specific distributable files that may not need to be updated for every build, such as the database container file and the initialized copy of the Customers table.

Finally, Figure 1 also shows a third set of directories from which other non-project specific distributable files are pulled. These include the VFP 9 runtime support library files, ActiveX controls, etc.

The steps in the build process can be traced from left to right in Figure 1, starting with compiling the EXE, continuing with copying the updated files to the deployment directory, and finishing by running the setup authoring tool to generate the deployment package.

Building the VFP EXE

One of the first steps in the build process is to compile the main EXE from the project source code. Among the things you typically need to do before compiling the new EXE for distribution are:

- Set VFP to recompile all files from their source code
- Clean out any printer-specific information from the first record of all VFP report files
- Turn off the debug code flag (unless you intentionally distribute your executable with debug code included)
- Increment the version number of the EXE

VFP does not lend itself to automating these tasks very well. You can write a script to run the compiler from the command line, and VFP 9.0 does a much better job of keeping printer-specific information out of report files than earlier versions did, but the process of building the EXE, although not difficult, is likely to be at least partially manual regardless of how fully automated your build process is.

Fortunately there is at least one tool that makes building the EXE easier. The free Project Builder from White Light Computing, Inc. gives you access to everything you need in one convenient window, as illustrated in **Figure 2**. You can download the WLC Project Builder from www.whitelightcomputing.com/prodprojectbuilder.htm.

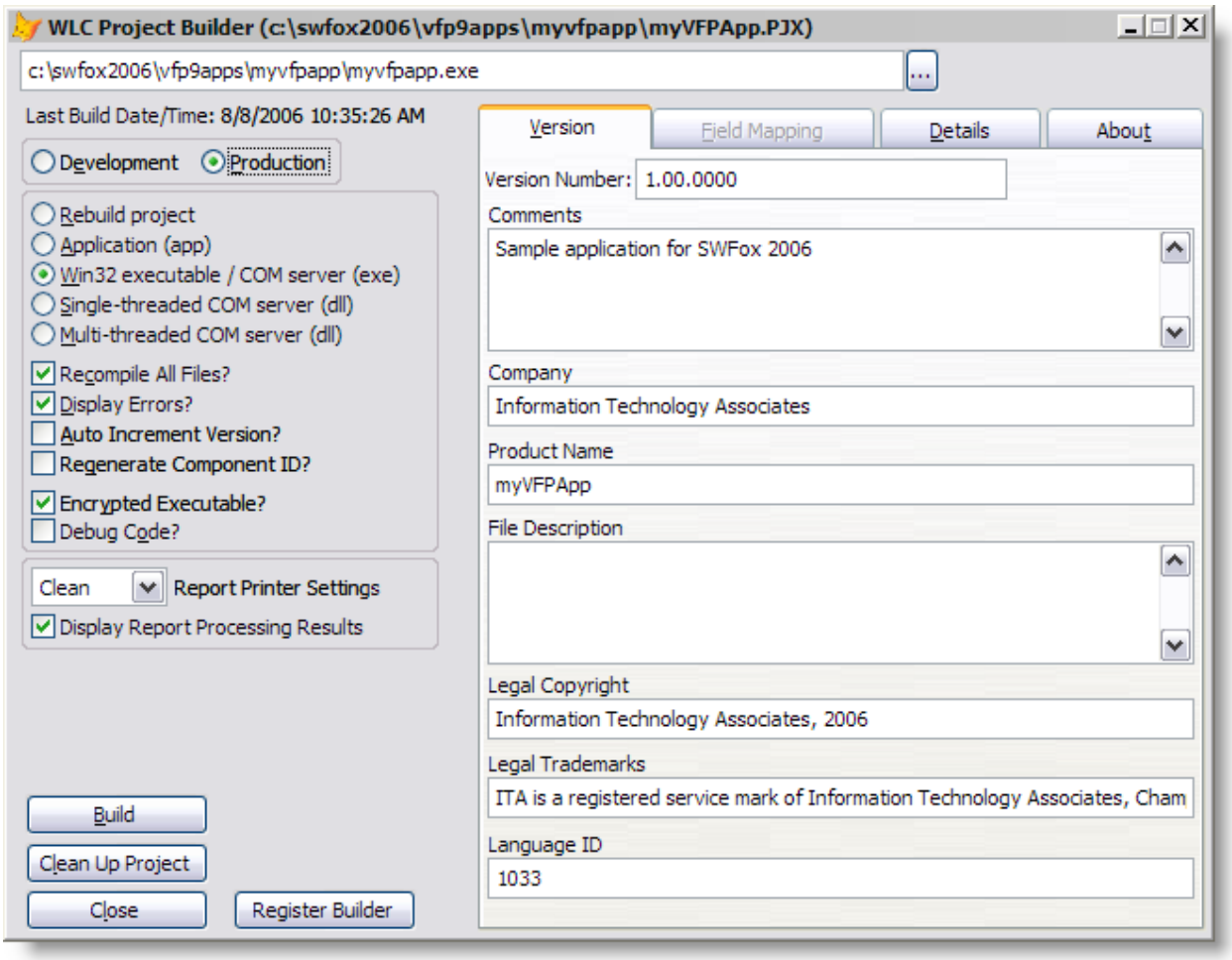

***Figure 2**: The WLC Project Builder from White Light Computing, Inc. provides convenient access to the options and settings for building the EXE in one convenient window.*

Manual builds

The defining characteristic of a manual build process, of course, is that all steps are performed manually. If you haven't done anything to automate your build, you are by definition using a manual build process.

A manual build process for the sample app used in this session would look something like this:

- If source control is used, manually check in all your most recent changes, then manually check out all the files required for compilation.
- Build the VFP EXE using the VFP Project Manager or a tool like the WLC Project Builder
- Use Windows Explorer or another file manager to individually copy the new EXE, DLLs, Readme file, Help file, and other updated distributable files from the working directory to the deployment directory
- Use your setup authoring tool to make any necessary changes to the setup script

- Tell the setup compiler to build the deployment package
- Copy the deployment package to a CD or upload it to an FTP site
- If source control is used, manually check in any newly created or updated files you want to store in the repository
- Manually make any other backups you may want to capture at this point

Pros and cons

Probably the most attractive characteristic of a manual build is that it's simple: no special tools or software are required. This makes it the least expensive solution, if you don't count the value of your time.

By its very nature, a manual build requires direct personal involvement in each step. Some developers might consider this a positive thing because they get to observe and control each step along the way. Other developers might feel this is a strong negative because it's time consuming and prone to error.

Either way, a manual build is tedious because it requires you to manually perform the same steps in the same order over and over again for each build. Human nature being what it is, it's easy to forget a step or make a mistake when handling things by hand.

Another disadvantage of manual builds is that there is no log of what was done, unless the developer creates one manually.

Tools

The best way to improve the reliability of a manual build process is to use a checklist. All the time. For every build. A simple checklist in Notepad or Word works just fine, but software tools that specialize in creating and using checklists can be helpful. Two of these are TaskTracker (www.positive-g.com) and ListPro (www.iliumsoft.com).

I've used ListPro for a number of years and for a variety of purposes. Although primarily intended for use on a Palm® PDA, there is also a Windows® version that works well for the kind of checklist you need for a manual build. **Figure 3** shows a sample of what such a list might look like in ListPro.

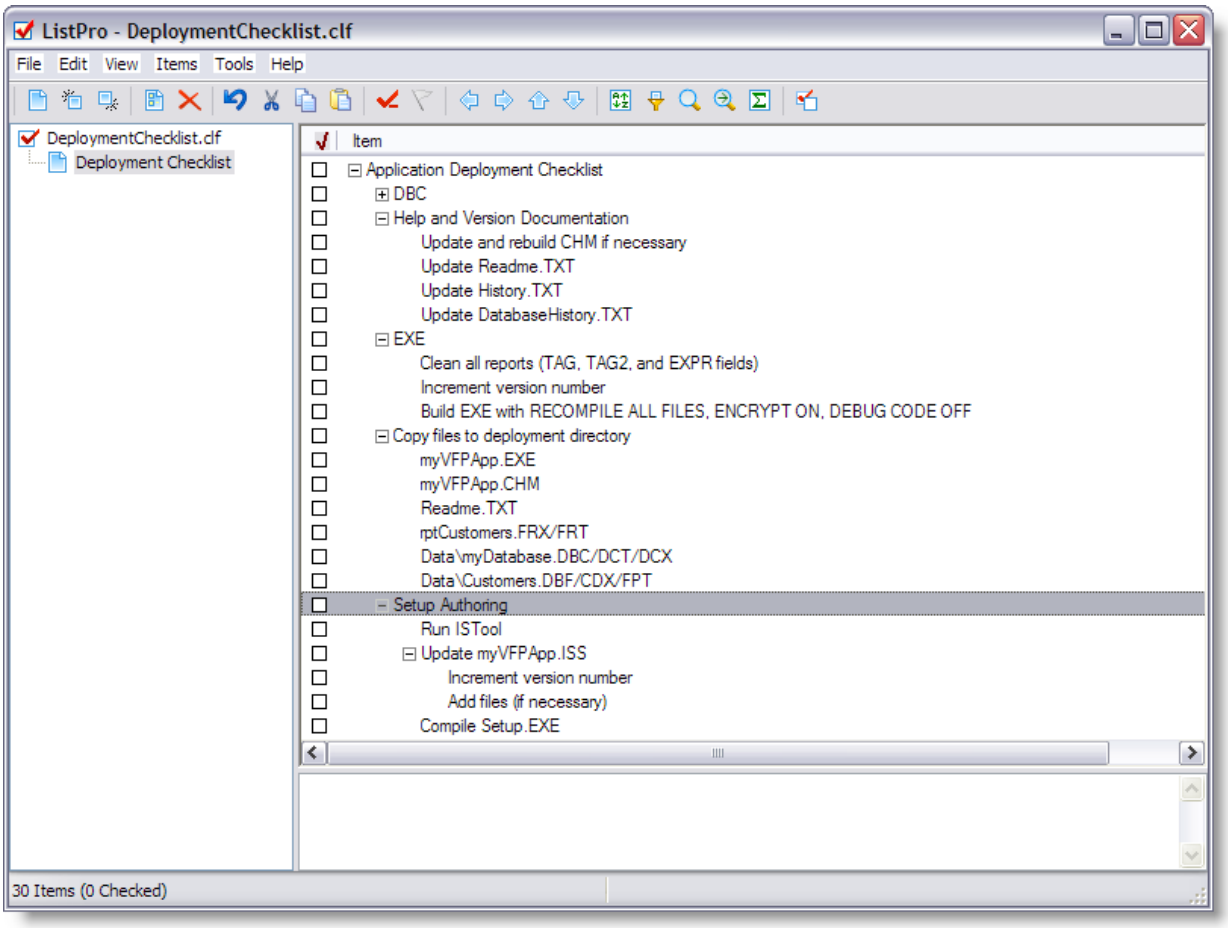

***Figure 3**: ListPro helps you create and use checklists, such as one you might want to use for a manual build process*

Reasons to automate

While a checklist makes it easier to reliably manage a manual build process, there are many compelling reasons to automate the build.

- An automated build process runs faster, saving valuable time in the development and deployment process.
- An automated build requires less manual involvement, reducing the developer's work load.
- An automated build is more reliable because it reduces the number of opportunities for human error.
- In an automated build process all actions are scripted, which creates intrinsic documentation of the steps involved in the build.

The benefits of automating the build are apparent even in a simplistic build process like I'm using here. The more complicated the build process, the more important these benefits become.

Semi-automated builds

The primary characteristic of a semi-automated build is that some of the steps are partially or perhaps even completely automated, but overall the build process still requires a good deal of manual involvement.

In a semi-automated build, individual steps may be grouped together and run as a unit, but each group still needs to be initiated manually.

Semi-automated builds can take advantage of low-cost, general-purpose utilities to make the process less labor-intensive than a manual build.

Pros and cons

The most important advantage of a semi-automated build, in my view, is that it reduces the amount of manual involvement required. Reducing the number of times the developer has to manually perform or initiate a step in the build process automatically reduces the number of opportunities for manual error.

Another benefit of a semi-automated build is that you automatically get some documentation of at least part of the build process. For example, some of the popular file compression utilities can read a file that contains a list of the files to be included in a zip file. That list of files not only automates the zip process but also serves as documentation of that portion of the build.

While a semi-automated build is a step in the right direction compared to a manual build, it is still has a few disadvantages. The primary disadvantage is that the steps in the build process are not all chained together, so each step or group of steps must still be launched manually. This of course means there are still opportunities for manual errors or omissions.

Another disadvantage is that semi-automated builds typically do not generate any kind of log of what was done. While perhaps not as important in small development shops as in larger ones, it is still a good idea to generate some kind of log of the build process as part of the documentation for each release.

Tools

Some common utilities I've found useful for semi-automated builds include WinZip®, PicoZip™, Beyond Compare®, and the often-overlooked and perhaps long forgotten Windows batch or command files.

Referring back to Figure 1, consider the different ways you could copy the updated EXE and other files required for a new release from the development directory to the deployment directory. In a manual build, you need do this by hand for each individual file using Windows Explorer or another file manager. In a semi-automated build, however, you can use a utility like WinZip or PicoZip to create a zip archive as an intermediate storage location, and then use a folder comparison utility like Beyond Compare to synchronize the latest zip archive with the deployment directory. This reduces a multi-step process to just two steps, regardless of how many files are involved.

The following examples illustrate how these tools can be employed to help achieve a semi-automated build.

Updating the deployment directory

The build process illustrated in the examples that follow is patterned after the three steps mentioned earlier, when version control is not in use. The first step, building the new EXE, is done from within the VFP IDE. The second step, copying the new EXE and other updated distributable files from the development directory to the deployment directory, can be accomplished in a number of ways, but I like to do it in two parts:

a) Use a file compression utility like WinZip or PicoZip to pull the updated distributable files from the development directory into a temporary zip archive; and

b) Use a folder synchronization utility like Beyond Compare to update the contents of the deployment directory with the new files from the temporary zip archive.

The advantages of using a zip file as an intermediate step are:

- you can create a simple text file containing a list of the files to be deployed, rather than having to handle each file individually;
- you isolate the files to be deployed from the other files in the development directory; and
- if you choose to use the folder synchronization utility with its user interface visible, you get visual confirmation of what's being copied into the deployment directory.

It's probably easiest to understand these advantages if we look at the folder synchronization step first. Beyond Compare presents a two-panel interface and uses color to show what's different on each side. Beyond Compare treats a zip file just like a folder, so we can set it up with the zip file (source) on left and the deployment directory (destination) on the right, as illustrated in **Figure 4**.

There are several things to note about what you see in Figure 4. First of all, the directory structure within the zip file on the left is identical to the directory structure of the development directory from which the files are taken, and therefore also identical to the structure of the target deployment directory on the right. This is important because it enables Beyond Compare to line up each file with its equivalent on the other side, making it easy for you to spot similarities and differences.

Next, notice that Beyond Compare's use of color to identify newer files makes them stand out from the others. You can see at a glance that the only newer file in the zip archive on the left is myVFPApp.exe, which is shown in red. Assuming the EXE is the only thing you updated in this release, this gives you visual confirmation that everything is in order.

You can also quickly compare the size of the new EXE on the left with the older one on the right. In most cases they should be roughly equivalent, at least for minor updates. One advantage of visually comparing the sizes of the two files is you can easily tell if something's out of whack. For example, if you expect the size of the newer EXE to be roughly the same as the older one but notice that it's significantly larger, that's a clue you may have forgotten to turn off debug code when you compiled the EXE.

Once satisfied the files you're about to copy into the deployment directory are complete and correct, simply click the Synchronize to Right button on the Beyond Compare toolbar to copy all of the newer files into the deployment directory in one step.

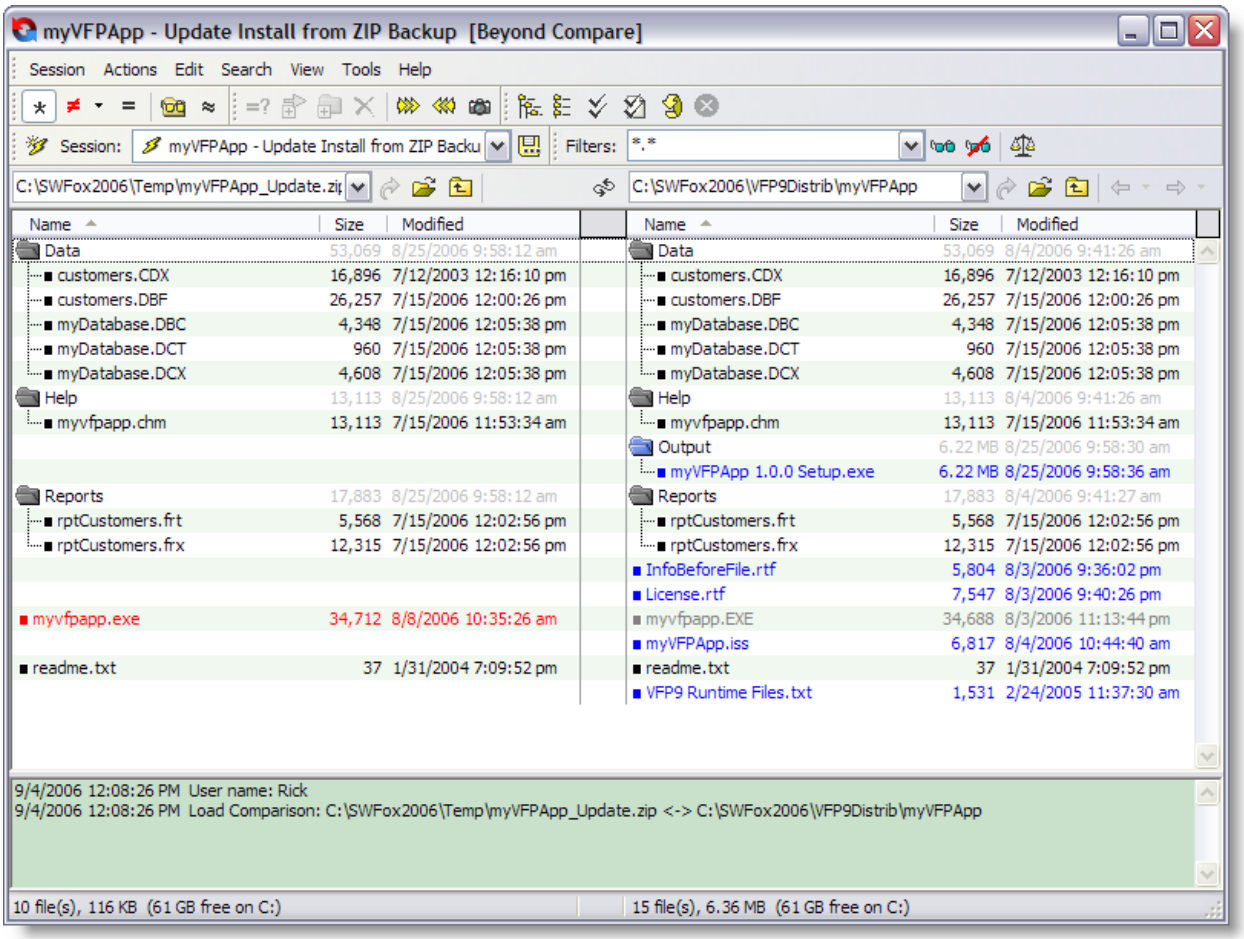

***Figure 4**. Beyond Compare makes it easy to see what's different on one side than on the other. In this example, the executable file myVFPApp.exe is newer in the zip file on the left, as expected.*

You can of course create the zip file manually, but the process can be automated. Doing so is easy, but as it turns out, getting the relative paths to come out as desired is not so easy.

Automating WinZip

There are at least two ways to automate WinZip. One is to download and install wzcline20.exe, the WinZip Command Line Support Add-On for WinZip 10.0. The add-on is free but requires the more expensive Pro version of WinZip 10.0. There is also a version of the Command Line Add-On for WinZip 9.0 and earlier. As of this writing, wzcline11.exe and other older WinZip files are available for download from WinZip's website at www.winzip.com/dprob.htm.

The other way to automate WinZip is to use the "undocumented" command line interface[5] built in to WinZip. The syntax for this interface is

```
winzip32 [-min] action [options] filename[.zip] files
```

[5] This "undocumented" interface used to be documented on the WinZip website. That documentation appears to have been moved or removed from the website, but the interface still works.

where the following options and parameters apply:

```
-min            run minimized

Actions:        -a      add
                -f      freshen
                -u      update
                -m      move

Options:        -r      include subfolders
                -p      store path information
                -ex     extra compression
                -en     normal compression (default)
                -ef     fast compression
                -es     super fast compression
                -e0     no compression
                -hs     include hidden and system files
                -sPassword      case-sensitive password, e.g. –s”myPassword”

Filename.zip    output file name (include drive and path if necessary)

Files           a list of one or more files, or an ampersand followed by the name of a file
                containing a list of files. Can use wildcards, e.g. *.frx, *.frt, etc.
```

The primary feature of interest here is the ability to pass the name of a file containing a list of the files to be zipped. For the sample application, the list of files would look like this:

Listing 1. *Files to be copied from the development directory to the deployment directory.*

```
C:\SWFox2006\VFP9Apps\myVFPApp\readme.txt
C:\SWFox2006\VFP9Apps\myVFPApp\License.rtf
C:\SWFox2006\VFP9Apps\myVFPApp\myvfpapp.EXE
C:\SWFox2006\VFP9Apps\myVFPApp\Data\myDatabase.DCX
C:\SWFox2006\VFP9Apps\myVFPApp\Data\customers.CDX
C:\SWFox2006\VFP9Apps\myVFPApp\Data\customers.DBF
C:\SWFox2006\VFP9Apps\myVFPApp\Data\myDatabase.DBC
C:\SWFox2006\VFP9Apps\myVFPApp\Data\myDatabase.DCT
C:\SWFox2006\VFP9Apps\myVFPApp\Help\myvfpapp.chm
C:\SWFox2006\VFP9Apps\myVFPApp\Reports\rptCustomers.frx
C:\SWFox2006\VFP9Apps\myVFPApp\Reports\rptCustomers.frt
```

If you store this list in a text file named myVFPApp_FileList.txt, you can pull all these files into an archive named winzip_backup.zip by running WinZip from the command line with the syntax shown in **Listing 2**.

Listing 2. *The WinZip command line syntax to create a zip archive from a file containing a list of files.*

```
"c:\program files\winzip\winzip32.exe" -a c:\swfox2006\temp\winzip_backup.zip
@myVFPApp_FileList.txt
```

The problem with this approach is the loss of relative path information in the resulting zip file. As you can see, the source file list gathers files from a root directory and three of its sub-directories, but **Figure 5** shows the zip file does not preserve the relative path information.

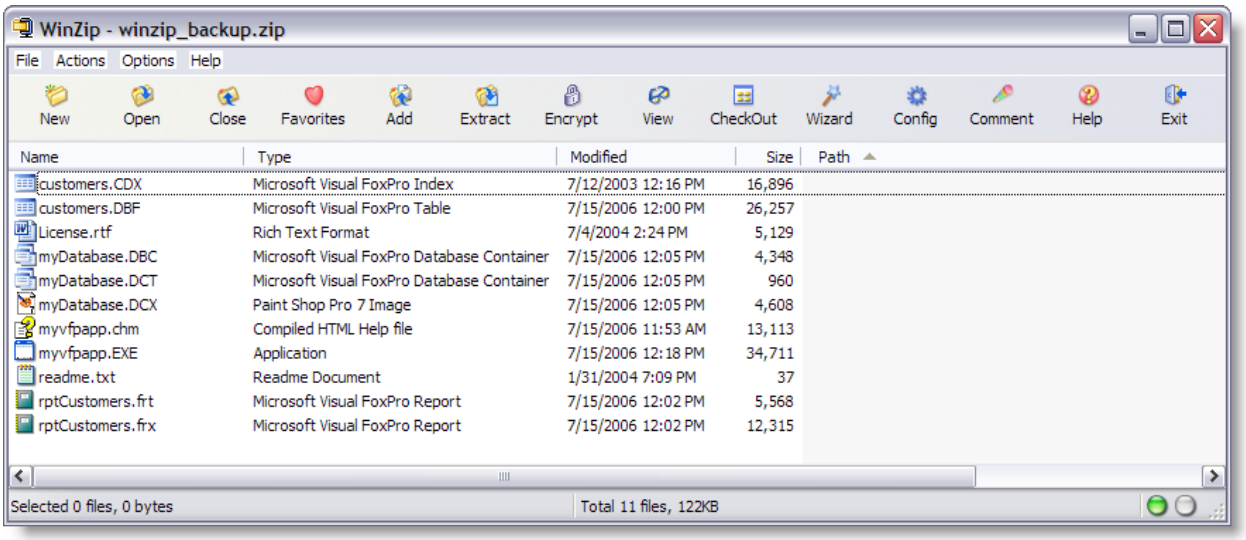

***Figure 5**:Relative path information is lost when the zip archive is created using the command line syntax in Listing 2.*

This would render the comparison in Beyond Compare useless, because the files wouldn't match up and you couldn't easily synchronize the deployment directly with the updated file(s) in the zip archive.

So we need a different solution. The command line syntax can be changed to store path info, as shown in **Listing 3**.

***Listing 3.** The WinZip command line syntax to create a zip archive from a file containing a list of files, including the –p option to include path information.*

```
"c:\program files\winzip\winzip32.exe" -a –p c:\swfox2006\temp\winzip_backup.zip
@myVFPApp_FileList.txt
```

But if you do it that way you end up with full path info instead of relative path info, as shown in **Figure 6**. This is equally unacceptable, because it too renders the comparison between the zip file and the deployment directory useless.

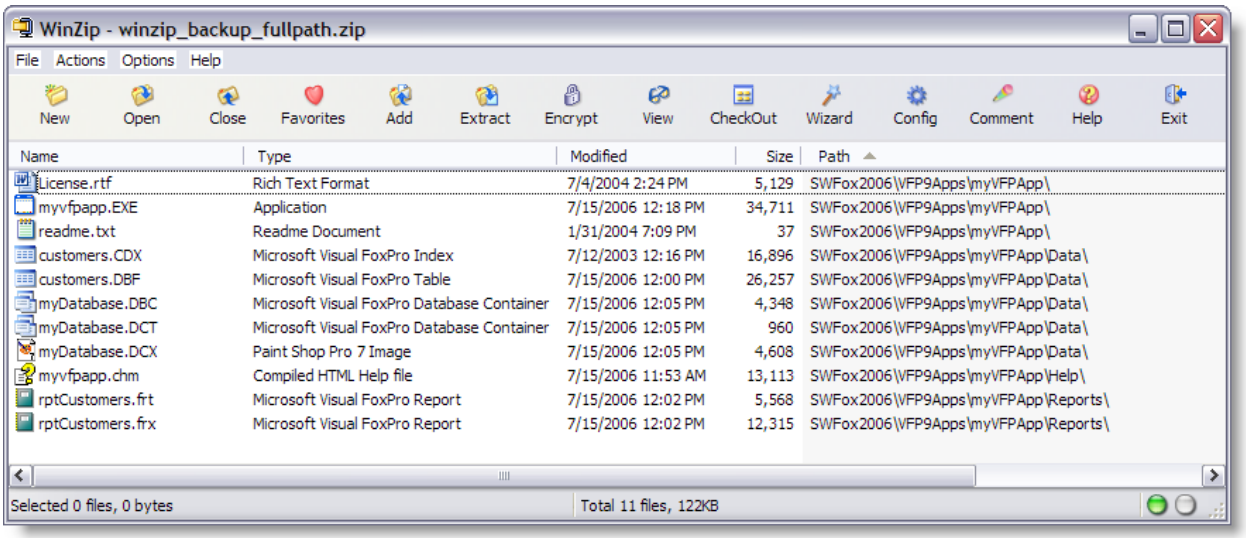

***Figure 6**. Full path information is stored when the zip archive is created using the syntax in Listing 3.*

What is needed is relative path info in the zip file, so the files end up in the same sub-directories relative to the root of the zip file as they are in relative to the root of the development (source) and distribution (target) directories.

The WinZip Command Line Add-On WZZIP.EXE is more powerful than the built-in command line interface. Among other things it has both –p (relative path) and –P (full path) options. The relative path option sounds promising, but unfortunately it only works the way we want it to when used in conjunction with the –r option to recursively include files from sub-directories. Using the –r option isn't feasible here because we do not want to pull in all the files from all the sub-directories, but only the files we've specified in our list of files in myVFPApp_FileList.txt. So up to this point we're still out of luck with WinZip.

There is a solution, though. WinZip 10.0 has a new feature called Data Backup Jobs. Jobs can be configured to select individual files and to store their relative path information. This new jobs feature is the only way I've found to accomplish the relative path configuration we need using WinZip. The jobs feature is available only in the Pro version of WinZip 10.0.

Creating a backup job is relatively straightforward, and there's a jobs wizard to help you out. Using the wizard is self-explanatory, so the only thing I want to point out is where you specify you want to store relative path information. This is done in step 3 of the wizard, as illustrated in **Figure 7**. This step is the same whether you're creating a new job or editing an existing one.

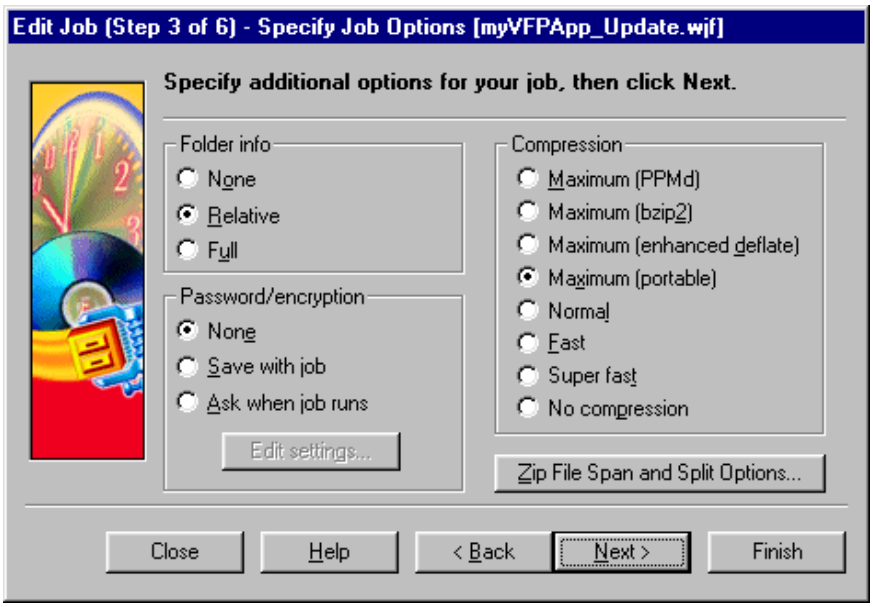

***Figure 7**: Specify you want to save relative path information under Folder Info in step 3 of the WinZip jobs wizard.*

When you run the backup job, the resulting zip file contains the files you selected plus their relative path, as shown in **Figure 8**. This gives us what we need to update the deployment directory using Beyond Compare.

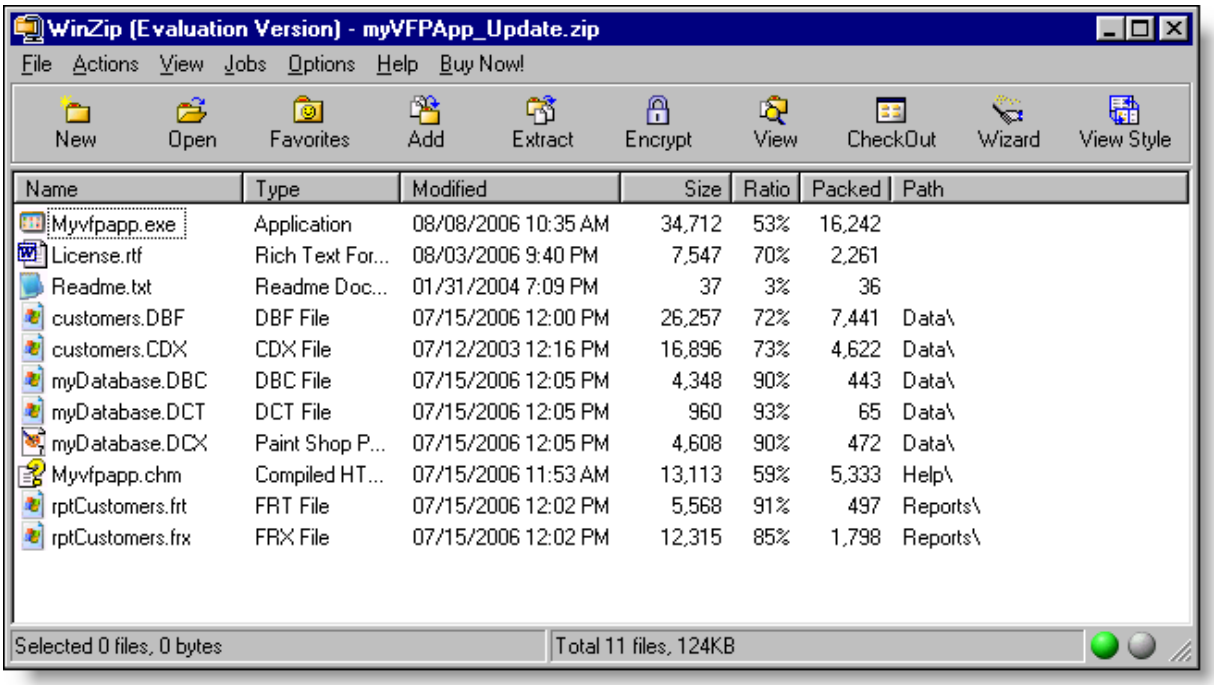

***Figure 8**: Using a WinZip 10.0 backup job enables you to store selected files with their relative path information.*

WinZip jobs are stored in text files with a .wjf file name extension. These are simple text files structured a lot like an INI file. Once you've created a job file using the wizard, you can edit it with any text editor.

WinZip job files are normally stored in My Documents\My WinZip Files. If you need to make minor changes to a job, such as adding a file or two, you might find it easier to simply edit the .wjf file directly rather than going back through the wizard. The .wjf file for the sample job I'm using here is shown in **Listing 4**.

Listing 4. *A WinZip job is stored in a text file with a .wjf file name extension.*

```
[version]
WJF-version=1

[output]
base=myVFPApp_Update.zip
baseappend=0
folder=C:\SWFox2006\Temp\
folderappend=0
log=none
logfolder=
logoverwrite=1
logtojobfolder=1

[options]
jobflags=0001
compression=0
span=1
split=0
splitunit=bytes
pwmode=0
cryptmode=0
pathmode=1
CdWriteSpeed=-1
CdFinalize=1

[wizard]
UseVars=0

[files]
0=FI-C:\SWFox2006\VFP9Apps\myVFPApp\myvfpapp.exe
1=FI-C:\SWFox2006\VFP9Apps\myVFPApp\license.rtf
2=FI-C:\SWFox2006\VFP9Apps\myVFPApp\readme.txt
3=FI-C:\SWFox2006\VFP9Apps\myVFPApp\Data\customers.dbf
4=FI-C:\SWFox2006\VFP9Apps\myVFPApp\Data\customers.cdx
5=FI-C:\SWFox2006\VFP9Apps\myVFPApp\Data\mydatabase.dbc
6=FI-C:\SWFox2006\VFP9Apps\myVFPApp\Data\mydatabase.dct
7=FI-C:\SWFox2006\VFP9Apps\myVFPApp\Data\mydatabase.dcx
8=FI-C:\SWFox2006\VFP9Apps\myVFPApp\Help\myvfpapp.chm
9=FI-C:\SWFox2006\VFP9Apps\myVFPApp\Reports\rptcustomers.frt
10=FI-C:\SWFox2006\VFP9Apps\myVFPApp\Reports\rptcustomers.frx

[schedule]
type=99
```

Once the WinZip job has been configured and stored as a .wjf file, the job can be run from within WinZip itself or from the command line using the /autorunjobfile option as shown in **Listing 5**. Path and file names that include spaces should be enclosed in quotes.

Listing 5. *A WinZip job can be run from the command line using the /autorunjobfile option.*

```
"c:\program files\winzip\winzip32.exe" /autorunjobfile myVFPApp_Update.wjf
```

You can incorporate this command into a .bat or .cmd file to run the WinZip job as part of an automated build process.

Automating PicoZip

PicoZip is a file compression utility from Acubix. PicoZip has the ability to create what it calls a backup set, which is a list of selected files along with the other attributes of the zip file you want to create. Backup sets enable you to store relative path information in the zip file.

PicoZip includes a tool for visually creating and editing backup sets. On the Backup Options page of that tool, you can specify you want to store relative path information by choosing "Relative Path" in the *Store Path Info* drop-down list, as shown in **Figure 9**.

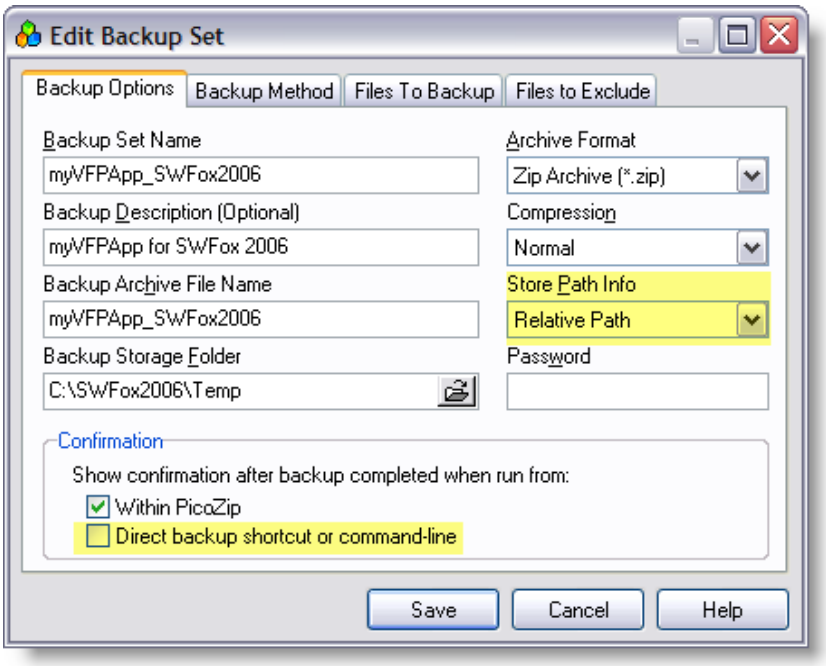

***Figure 9**. PicoZip comes with a tool to create backup sets. You can easily set it up to store relative path information.*

When running a PicoZip backup set as part of an automated build process, you may want to run it without any user intervention required. To do so, unmark the "Direct backup shortcut or command line" check box under *Show confirmation after backup completed when run from*, as illustrated in Figure 9. Otherwise, when the backup is complete PicoZip displays a confirmation dialog the user must dispatch by clicking OK.

Selecting Relative Path in the Backup Options dialog produces a zip file that contains the files selected for backup along with their path relative to the root directory, as shown in **Figure 10**.

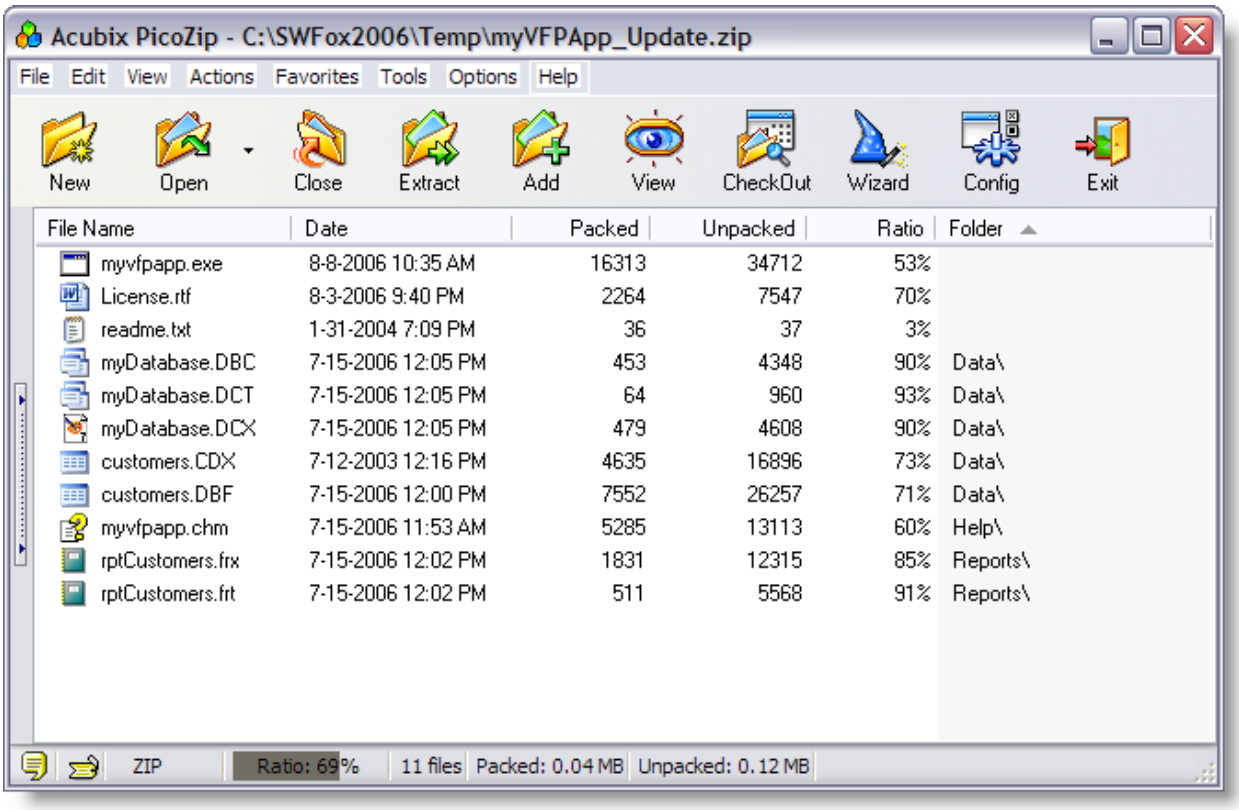

***Figure 10**: PicoZip stores the selected Files with their relative path information.*

PicoZip backup sets are stored as simple text files with a .zfb file name extension. These files look a lot like INI files, and because they are simple text files they can be edited with any text editor. These files are normally stored in C:\Program Files\PicoZip\Backups\. The .zfb file for the sample backup set is shown in **Listing 6**.

***Listing 6.** The PicoZip backup set file to create a zip archive of the distributable files for the sample application myVFPApp.*

```
BackupSetName=myVFPApp_SWFox2006
Description=myVFPApp for SWFox 2006
BackupFolder=C:\SWFox2006\Temp
BackupFilename=myVFPApp_SWFox2006
ArcType=6
Compression=1
StorePathInfo=0
Password=
ConfirmNormal=1
ConfirmCommandLine=0
SkipReadOnly=0
SkipHidden=0
SkipSystem=0
BackupMethod=1
Generations=3
IncrementalModified=0

[FilesInBackupSet]
C:\SWFox2006\VFP9Apps\myVFPApp\myvfpapp.EXE
C:\SWFox2006\VFP9Apps\myVFPApp\License.rtf
C:\SWFox2006\VFP9Apps\myVFPApp\readme.txt
C:\SWFox2006\VFP9Apps\myVFPApp\Data\customers.CDX
C:\SWFox2006\VFP9Apps\myVFPApp\Data\customers.DBF
```

```
C:\SWFox2006\VFP9Apps\myVFPApp\Data\myDatabase.DCX
C:\SWFox2006\VFP9Apps\myVFPApp\Data\myDatabase.DBC
C:\SWFox2006\VFP9Apps\myVFPApp\Data\myDatabase.DCT
C:\SWFox2006\VFP9Apps\myVFPApp\Help\myvfpapp.chm
C:\SWFox2006\VFP9Apps\myVFPApp\Reports\rptCustomers.frx
C:\SWFox2006\VFP9Apps\myVFPApp\Reports\rptCustomers.frt

[FoldersInBackupSet]

[ExcludeList]
```

This file is initially created by PicoZip when you create a Backup Set from within the program itself, but once created the file can be edited with any text editor, making it easy to add or remove files when necessary.

Once a backup set has been configured and stored as a .zfb file, you can run it from within PicoZip or you run it from the command line using the –B option, as shown in **Listing 7**. Path and file names that include spaces should be enclosed in quotes.

Listing 7. *Run PicoZip with the –B option and pass the name of the backup set as a parameter.*

```
"C:\Program Files\PicoZip\PicoZip.exe" -B myVFPApp_SWFox2006.zfb
```

You can incorporate this command into a .bat or .cmd file to run the PicoZip backup job as part of an automated build process.

Automating BeyondCompare

Once you've configured a folder comparison that way you want it in Beyond Compare, you can save it as a *session*. Sessions can be re-opened and re-used as often as needed.

Figure 4 shows a session named *myVFPApp – Update install from ZIP Backup*, which I created for the sample application. It compares the contents of a zip file named myVFPApp_Update.zip to the deployment folder VFP9Distrib\myVFPApp in preparation for copying the newer files from one to the other.

You can open a session by choosing it from the list of available sessions in Beyond Compare, or you can automate the process using a command line parameter. **Listing 8** shows the command line syntax for opening the sample session *myVFPApp – Update install from ZIP Backup*. As always, strings that include spaces should be enclosed in quotes.

Listing 8. *You can run Beyond Compare from the command line and open a session automatically.*

```
"C:\Program Files\Beyond Compare 2\bc2.exe" "myVFPApp - Update Install from ZIP Backup"
```

This opens the session in the Beyond Compare window, as shown in Figure 4. If you want visual confirmation of what's new in the zip file before copying files to the deployment directory, this is the way to go.

On the other hand, if you're interested in silent automation Beyond Compare also offers a scripting language. Using a script it's possible to update one directory with newer files in the other without the Beyond Compare user interface. I won't go into details here; the Beyond Compare Help file has information and examples on how to do this.

As of v2.4 Build 240 (March, 2006) BeyondCompare supports check in and check out with most version control systems. If you use version controls software, this has the potential to significantly extend the usefulness of Beyond Compare in an automated build process.

Building the deployment package

Once the deployment directory is up to date with the latest files for the new release, you're ready to build the deployment package. Of course you have your choice of setup authoring tools here, but I'm going to discuss two of the popular ones used by VFP developers, Inno Setup and InstallShield Express.

Automating Inno Setup

Lots has been written about how to use Inno Setup to deploy VFP apps and this is not the place to reproduce any of that. If you're interested in but not familiar with Inno Setup, one place to start is the VFP developers page of my website at www.ita-software.com/foxpage.aspx, where you'll find several white papers and a videocast on the subject. A Google search for "Inno Setup AND FoxPro" turns up many other valuable references, too. On the other hand, if you're not interested in Inno Setup at all you can skip this section and go directly to the next section on Automating InstallShield.

If you use Inno Setup to build the deployment package for your VFP app, there are at least three ways to automate that process from the command line. In all three cases it's assumed you have the Inno Setup script (.iss file) for the application already written.

The first way to automate Inno Setup is to launch the compiler directly and pass the name of the script you want to compile, as shown in **Listing 9**.[6] Path and file names that include spaces should be enclosed in quotes.

Listing 9. *Run the Inno Setup compiler with the /cc option and pass the name of the script file.*

```
"C:\Program Files\Inno Setup 5\compil32.exe" /cc "C:\SWFox2006\VFP9Distrib\myVFPApp\myVFPApp.iss"
```

Running the compiler in this way opens the Inno Setup user interface window for the duration of the build. The script is not opened for editing but compiler output messages are displayed as the build progresses. The compiler returns an exit code of 0 if the build was successful.

An alternative method is to use the Inno Setup console mode compiler issc.exe. The syntax for the sample build using this method is shown in **Listing 10**.

Listing 10. *Run the Inno Setup console mode compiler with the /Q option and pass the name of the script file.*

```
"C:\Program Files\Inno Setup 5\iscc.exe" /Q "C:\SWFox2006\VFP9Distrib\myVFPApp\myVFPApp.iss"
```

[6] All Inno Setup examples in this paper are based on version 5.1.7, which is the latest release as of this writing.

The /Q parameter is optional. It tells the compiler to run in quiet mode, displaying only error messages if any occur. Other parameters can be passed to specify an output path and file name, if you want to override the ones specified in the script. See the Inno Setup Help file for more information.

Running the console mode compiler with the /Q option results in a completely silent build. No user interface window is opened and no user response is required unless there is an error.

Many developers who use Inno Setup also use the companion tool ISTool. ISTool provides a graphical user interface to Inno Setup scripts, and it can run the Inno Setup compiler directly from within its own window.

ISTool can also be automated. One way is to open the ISTool user interface with the script loaded for editing. This is useful if you want to update the script before compiling, for example to increment the version number, add or change a file reference, etc. **Listing 11** shows the syntax for doing this from a command line.

Listing 11. *Run ISTool and pass the name of the script file to open for editing.*

```
"C:\Program Files\ISTool 4\istool.exe" "C:\SWFox2006\VFP9Distrib\myVFPApp\myVFPApp.iss"
```

ISTool is a separate product from a different developer, but a new release of ISTool is generally available shortly after each new release of Inno Setup. Even though ISTool is currently at version 5.1.6, it still installs to the ISTool 4 directory under Program Files, as shown in Listing 8.

Another way to automate ISTool is to pass the –compile parameter, which tells ISTool to immediately compile the script using the Inno Setup compiler. The syntax for this is given in **Listing 12**.

Listing 12. *Run ISTool and pass the –compile parameter as well as the name of the script file.*

```
"C:\Program Files\ISTool 4\istool.exe" -compile "C:\SWFox2006\VFP9Distrib\myVFPApp\myVFPApp.iss"
```

When run in this manner, ISTool opens the script in its editing window and immediately launches the Inno Setup compiler. Although the script is visible in the editing window, you do not have a chance to edit it because compilation begins right away. Compiler output is displayed in a child window, the same as if you'd launched the compiler manually from within ISTool. Assuming there were no errors, the ISTool window is automatically closed when compilation is complete.

Automating InstallShield

Ever since VFP 7.0, Visual FoxPro has shipped with a limited edition of InstallShield Express named *InstallShield Express - Visual FoxPro Limited Edition*, or ISX VFP LE for short.

While ISX VFP LE works fine as far as it goes, it lacks several features found in the full versions of InstallShield Express. Because of this, many developers including myself use the more current fully featured versions of InstallShield Express. I no longer have ISX VFP LE installed on my

machine, but I do still have the full version of InstallShield Express 5.0. The examples I use in this paper are based on that version, which is as close as I can come to the version that ships with VFP 9.0. As far as I know these examples also work with the version of ISX VFP LE that ships with VFP 9.0, and quite possibly with older versions as well.

Automating InstallShield Express is simply a matter of being able to tell it what you want it to do from the command line. InstallShield Express 5.0 stores its project files with a .ise file name extension. The examples that follow assume you have already built the InstallShield Express project for the application.

To launch InstallShield Express and open an existing project for editing, you can use the syntax shown in **Listing 13**.

Listing 13. *Launch InstallShield Express and pass the name of the project file to open for editing.*

```
"C:\Program Files\InstallShield\Express 5.0\iside.exe" "C:\SWFox2006\InstallShield Express
Projects\myVFPApp.ise"
```

If the project file requires no editing, you can compile it directly as shown in **Listing 14**.

Listing 14. *Launch InstallShield Express and pass the name of the project file to compile.*

```
"C:\Program Files\InstallShield\Express 5.0\system\IsExpCmdBld.exe" -p
"C:\SWFox2006\InstallShield Express Projects\myVFPApp.ise" -r SingleImage -c COMP -e y
```

Note that several parameters are passed to InstallShield Express in this command string. The –p parameter tells ISX which project file to build. The –r parameter specifies the desired release type, while the –c parameter is the compression option. Passing ‘y’ as the –e parameter tells ISX you want to create a setup.exe file. These and the other available parameters are documented in the Command-Line Build Parameters topic of the ISX Help file.

Running the build using the syntax in Listing 11 does not open the InstallShield Express IDE, but it does display the build progress in the command window. If you don’t want this you can achieve a silent build by adding the –s parameter, as shown in **Listing 15**.

Listing 15. *Perform a silent build by passing the –s parameter.*

```
"C:\Program Files\InstallShield\Express 5.0\system\IsExpCmdBld.exe" -p
"C:\SWFox2006\InstallShield Express Projects\myVFPApp.ise" -r SingleImage -c COMP -s -e y
```

As you can see, the command line can get a bit long. To alleviate this, InstallShield Express allows the build options to be stored in an INI file, which is passed to ISX using the –i parameter. Some developers may find this more convenient because an INI file is easy to edit and the command line required to run the build is shorter.

The build options used in Listing 15 can be incorporated into an INI file like this:

Listing 16. *Build options can be stored in an INI file.*

```
[Project]
Name="C:\SWFox2006\InstallShield Express Projects\myVFPApp.ise"
Product=myVFPApp
```

```
BuildLabel=Express

[Release]
Configuration=COMP
Name=SingleImage
SetupEXE=y

[Mode]
Silent=yes
```

If the INI file in Listing 16 is saved as myVFPApp_ISE.ini, then the command line to run the build is as shown in **Listing 17**.

Listing 17. *InstallShield Express can read its build options from an INI file.*

```
"C:\Program Files\InstallShield\Express 5.0\system\IsExpCmdBld.exe" -i
"C:\SWFox2006\Sessions\Build\myVFPApp_ISE.ini"
```

Putting it all together

The final step in creating a semi-automated build is to create a single script to run each of the steps in the correct sequence. One easy way to do this is to use a Windows batch (.bat) or command (.cmd) file.

The objective is to encapsulate the steps required to perform the build into a single file so that running that file launches each of the required steps in the correct sequence. As you've seen, there are often several different ways to run each step, so you need to decide which alternatives you want to use for your own build process. For the example I'm using here, I'll choose to run PicoZip using the command from Listing 5, Beyond Compare using the command from Listing 6, and ISTool using the command shown in Listing 8 (which gives me a chance to edit the script before compilation). Adding a few comments and a bit of user feedback along the way results in the file shown in **Listing 18**.

Listing 18. *A Windows .cmd file with the desired commands for a semi-automated build.*

```
@echo off
REM Build the deployment package for myVFPApp.

echo Step 1 - Create zip backup of updated files
"C:\Program Files\PicoZip\PicoZip.exe" -B "C:\SWFox2006\Sessions\Build\myVFPApp_Update.zfb"
echo Step 1 complete

echo Step 2 - Update the deployment directory
"C:\Program Files\Beyond Compare 2\bc2.exe" "myVFPApp - Update Install from ZIP Backup"
echo Step 2 complete

echo Step 3 - Edit the setup script using ISTool, then compile manually from the IDE.
"C:\Program Files\ISTool 4\istool.exe" "C:\SWFox2006\VFP9Distrib\myVFPApp\myVFPApp.iss"
echo Step 3 complete

REM Visually confirm the presence, size, and datetime stamp of the setup package.
DIR "C:\SWFox2006\VFP9Distrib\myVFPApp\Output\*.exe"

echo Done!
```

If your memory is bad or you just don't want to be bothered with having to remember to run the Inno Setup compiler from within ISTool, you can add a line in step 3 to run the compiler after

you close ISTool. Use the command from Listing 6 or Listing 9 to do this. The worst case is the compile will run twice, which doesn't hurt anything but is a lot better than forgetting to run it at all!

Save the commands shown in Listing 15 as Build_myVFPApp.cmd or another name of your choosing, and you can now launch your entire build process simply by double-clicking the .cmd file in Windows Explorer.

This begins to look a whole lot like a fully automated build. I'd still consider it a semi-automated build because it requires a certain amount of manual involvement along the way, as well as for other reasons, but it still represents a significant improvement over a manual build.

Fully Automated builds

A fully automated build is probably the Holy Grail of build processes, and, like finding the putative Holy Grail itself, may be impossible to achieve. Nonetheless it's certainly possible to attain significant improvement over a manual or even a semi-automated build process using special purpose software designed specifically to fully automate the build.

A fully automated build displays the following characteristics:

- Each step is fully scripted and automated
- The entire build process is defined in one place
- One touch launches the entire process
- Each step is dependent on the successful completion of the previous one
- There is a provision for handling errors that may occur during the build process
- Little if any user interaction is required
- A log is generated each time the build process is run

Pros and cons

On the positive side, a fully automated build offers several advantages. Because it provides a totally hands-off build process, it presents the least opportunities for error once the scripts have been tested and debugged. Because special purpose software is generally used, all of the steps in a fully automated build are usually defined in one place, namely the file used to run the build.

The software used to accomplish a fully automated build generally offers a higher degree of integration with other parts of the build process than might otherwise be achieved. This includes integration with version control software, integration with source code compilers, and integration with setup authoring tools.

Software for running fully automated builds can also create a log of what was done each time the build is run, and may be able to generate automated notification to team members. They also may offer error handling and the ability to specify actions to be taken if and when an error occurs somewhere in the build process.

Setting up a fully automated build may initially be more complicated and time consuming than setting up a manual or semi-automated build, but once configured and running smoothly you get some of that time back each and every time you run the build.

Tools

A fully automated build is almost certainly going to require special purpose software designed specifically for that purpose. Here you are confronted with the old build or buy decision. While writing your own is certainly possible—you *are* a developer, after all—it's no trivial task and would clearly take a lot of time. On the other hand, professional tools are expensive relative to the cost of the general purpose utilities discussed earlier. What's the best way to go?

Homegrown tools

If you're averse to spending money for a professional tool, you may want to consider rolling your own build automation tool. If you're adept at writing command or script files you could probably come up with something suitable, at least for a simple build. More likely, as a developer, you might look at your favorite tool (VFP, of course!) and consider how much work it would be to write an app to automate your build process.

It's feasible, but is it smart? Both of these approaches may have merit, but like so many other things the question probably ought to be: Why would I want to take time away from billable hours to build something somebody else has already built and which I can buy for a reasonable price? The two professional tools I describe here each cost a few hundred dollars. Divide that by your billing rate. Could you write an equivalent tool in that number of hours? I know I couldn't.

The purpose of saying all this is not to discourage you from coming up with your own solution or to diminish the potential value of a solution you might write for yourself, but rather to suggest that a good 3rd party tool is usually well worth its price when considered in the greater scheme of things.

Professional tools

The two professional tools I discuss here are Final Builder from VSoft Technologies Pty Ltd and Visual Build Pro from Kinook Software, Inc. Both are special purpose software packages designed to help fully automate the build process. Both are also full featured applications in their own right, with enough features and capabilities to fill a session or more likely several sessions of their own.

The intent of this section is therefore not to describe these tools in depth but to introduce you to them in the context of our simplistic little sample application, and hopefully by doing so to give you enough information to evaluate the potential these tools might offer you in your own work.

Final Builder

The descriptions and screenshots in this paper are from Final Builder Professional Edition version 4.2.0.276, which is the latest version available as of this writing. You can download an evaluation copy of Final Builder from the publisher's website at www.finalbuilder.com.

You begin by creating a Final Builder project and populating it with actions. Actions correspond the steps involved in your build process. If you've never automated your build or even given it much thought, you may at first by stymied by this concept. If so it's probably because you've never stopped to consider, much less to write down, the steps you actually perform to create a build. If you've read the first part of this paper, though, you'll see that actions in a Final Builder project correspond directly to the steps enumerated in the examples already given.

Assuming the build process to be automated with Final Builder follows the same three steps I've been describing all along—build the VFP EXE, update the deployment directory, and build the deployment package—you probably want Final Builder to take over starting with the second step.[7] As a starting point, we can set up actions in the Final Builder project to accomplish the same steps we automated from the command line in the previous section of this paper.

The Final Builder IDE has two views, Design and Build Summary. You create and edit your project from the Design view. **Figure 11** shows a sample project open in the Design view.

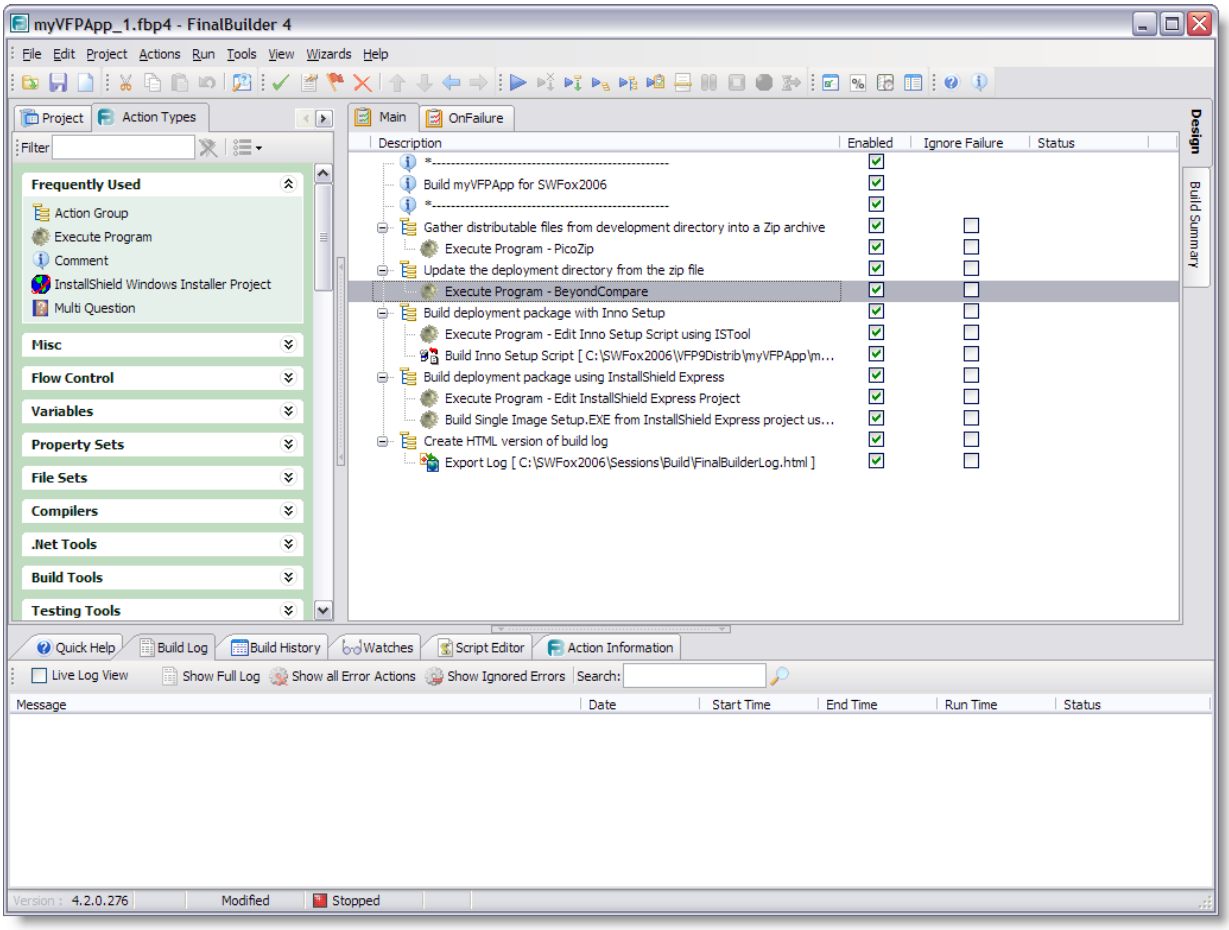

***Figure 11**: A simple project open in Final Builder. Note the Design tab is selected on the far right.*

[7] I've already described the reasons you probably want to build the VFP EXE manually from within the VFP IDE.

The Design view consists of three panels. In the left-most panel, the list of Action Types is selected. To build your project, you add actions to the Main panel from the Action Types list. Final Builder remembers the actions you've used most frequently and adds them to the Frequently Used list, which you can see is expanded in Figure 11.

You can also see in Figure 11 that each action in the Main panel has an Enable check box. Marking or unmarking these check boxes allows you to control which actions are executed when the build is run. One use for this is in testing, where you might want to run only a subset of the actions. There are also icons on the toolbar that enable you to run selected portions of the build.

Another use for the Enabled check boxes is illustrated in Figure 11, where I've set up action groups to build the deployment package using both Inno Setup and InstallShield Express. It's unlikely I'd do both is a real deployment scenario, but it's useful to have both available for demonstration purposes. Rather than having to maintain two copies of the project, I can simply mark and unmark to desired actions to use one setup authoring tool or the other, or both.

The action type I used most frequently, at least at first, is the Execute Program action. This action lets you launch an external program in the same way you would run it manually or from a command file. The Execute Program action is therefore an ideal way to convert the commands we built for use from the command line in the previous section.

Double clicking on an action in the Main panel opens its properties sheet. **Figure 12** shows the properties sheet for the Beyond Compare step in our build process. Note the syntax for launching Beyond Compare here is identical to the command used in the earlier example but broken into two parts, the command itself and the parameter.

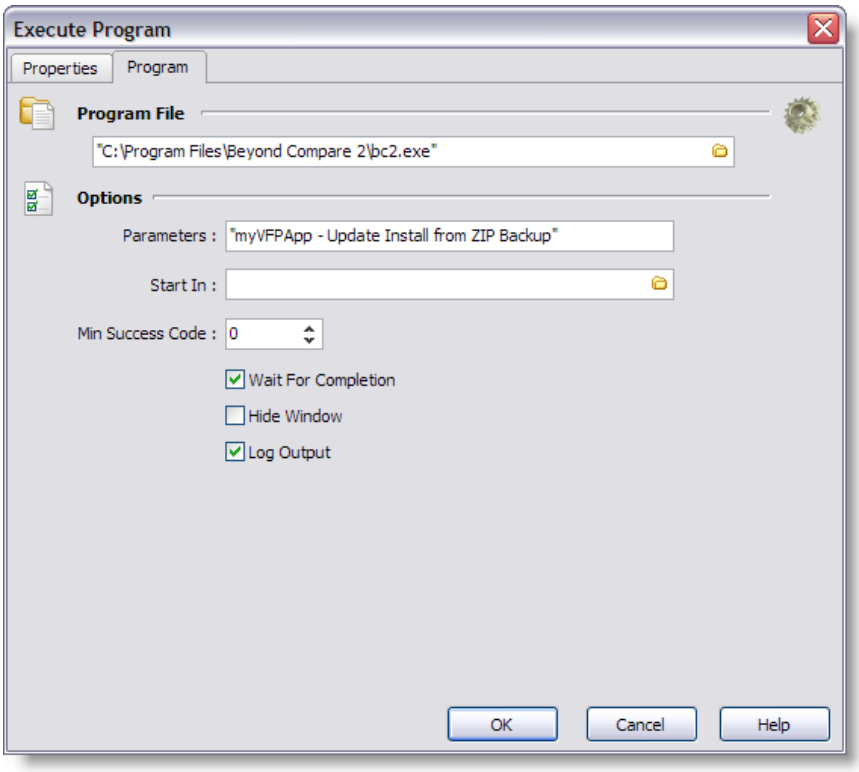

***Figure 12**: In the Execute Program action dialog you specify the program to be run and any parameters to be passed to it.*

In the lower portion of the window shown in Figure 12, note the provision for specifying a minimum success code. This is the return value Final Builder will look for as indication the program completed successfully. Usually this will be zero, but you can change it if necessary.

Also note the three check boxes below the minimum success code. In almost all cases you will want to wait on completion of any action before continuing with the next, so normally you'll leave the Wait For Completion check box marked.

The Hide Window check box enables you to run control whether the program should be run with or without a visible window. For programs that don't require user interaction you'll probably want to leave this check box marked. On the other hand, if the program does require user interaction then be sure to unmark the Hide Window check box or the build will hang indefinitely waiting on a user response to an invisible window! In this example we want the Beyond Compare window to be visible so we can visually compare the two sides before copying the file(s) to the deployment directory, and the action won't terminate until we close the Beyond Compare window, so we unmark the Hide Window check box.

If Log Output is marked, Final Builder will capture all console output from the program being run and write it to the Final Builder log. For example, the output from the Inno Setup compiler can be captured to the Final Builder log in this manner. You'll usually want to leave this check box marked.

One of the nice things about Final Builder is its awareness of and ability to integrate with a wide range of other software. This includes but is by no means limited to utilities such as Beyond

Compare, setup authoring tools such as Inno Setup, InstallShield, and many others, and a wide range of version control software packages.

The pre-configured actions for each of these software package are available in the Action list. Some, like the version control software, are listed individually. Others, like the setup authoring tools, are part of a group. The actions for setup authoring tool, for example, are grouped under Install Builders in the Action list

Look at Figure 11 and find the action that runs the Inno Setup compiler. You'll notice it's not set up as an Execute Program action but rather as a Build Inno Setup Script action. This is possible because Inno Setup is one of the install builders that Final Builder "knows" about and provides a pre-configured action for. **Figure 13** shows the configuration window for the Build Inno Setup Script action.

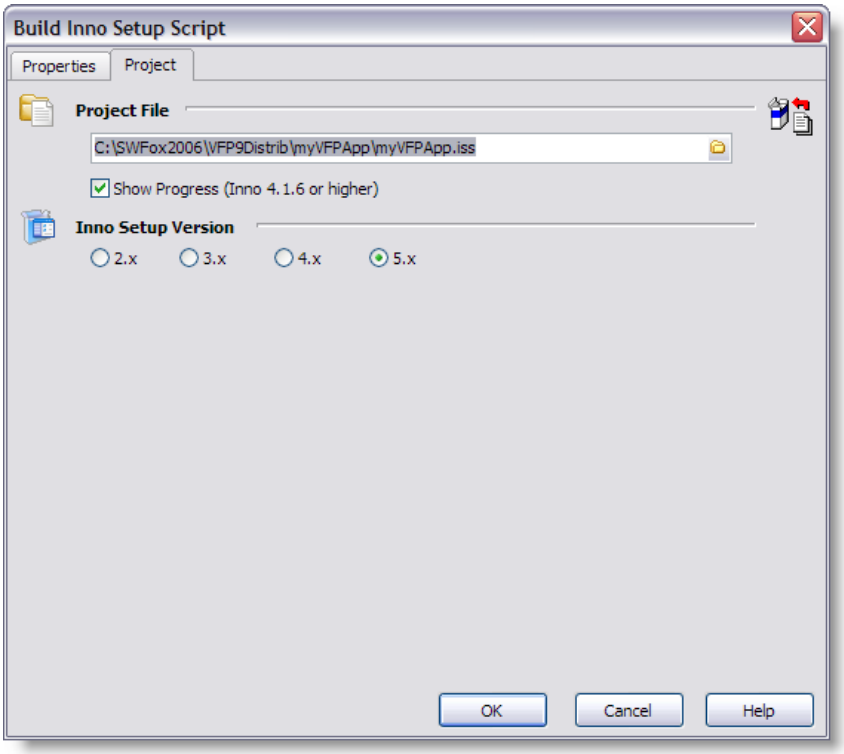

***Figure 13**. Final Builder "knows" about Inno Setup and provides a customized dialog for the Build Inno Setup Script action.*

Note that version 5.x is marked, indicating we want to use Inno Setup version 5. If Inno Setup is installed on your machine when you install Final Builder, Final Builder should recognize and locate it automatically. If not, you can specify the location of Inno Setup manually via the Final Builder Options dialog. **Figure 14** shows the Options dialog for configuring Final Builder to be aware of Inno Setup. Similar Options dialogs exist for the other software Final Builder can be made aware of.

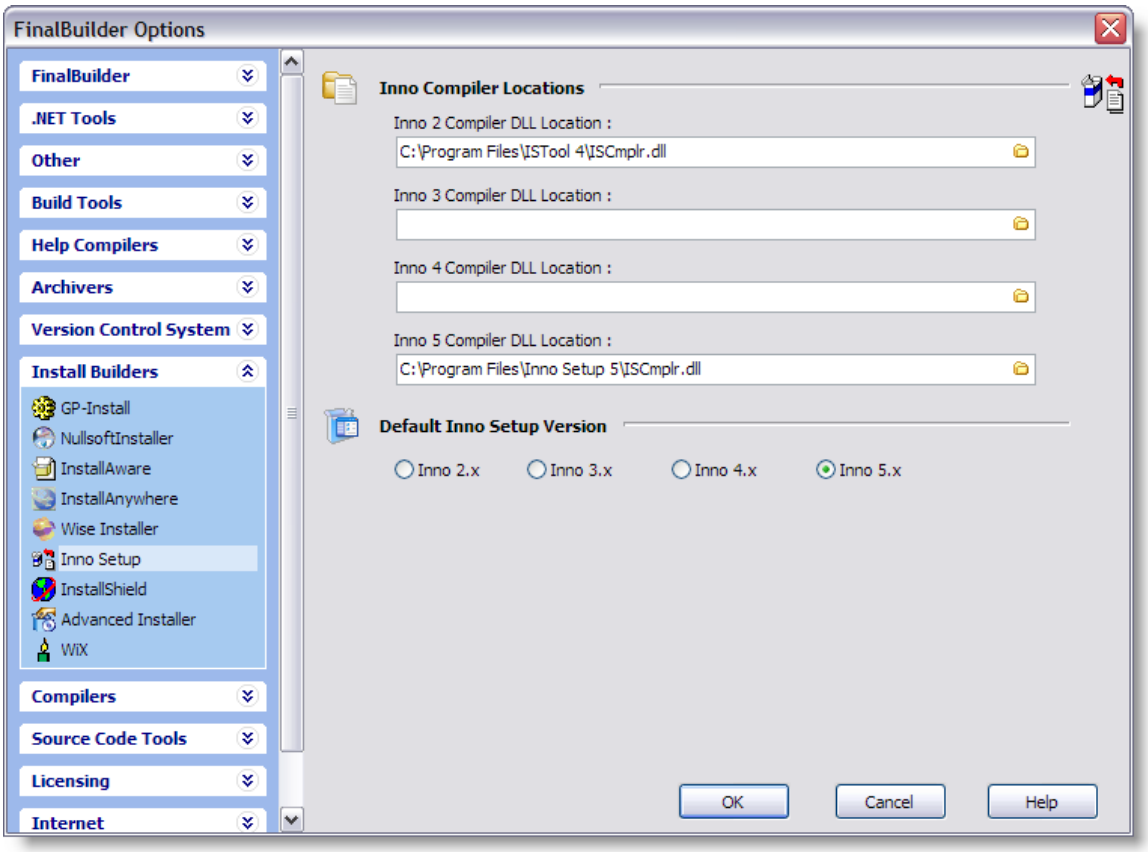

***Figure 14**. The Options dialog is where you specify the location and attributes of other software packages Final Builder is aware of.*

Final Builder offers a number of ways to interact with the developer during the build process. For example, you can make it ask a question. This action and others like it are found under the Interactive action types.

In out example, one question we might want to ask is whether to use Inno Setup or InstallShield. This is easily done by adding an Ask Question action before each of these steps and making the related actions dependent on the answer. **Figure 15** shows a modified Final Builder project with these questions in place, as indicated by the shaded lines in the Main panel.

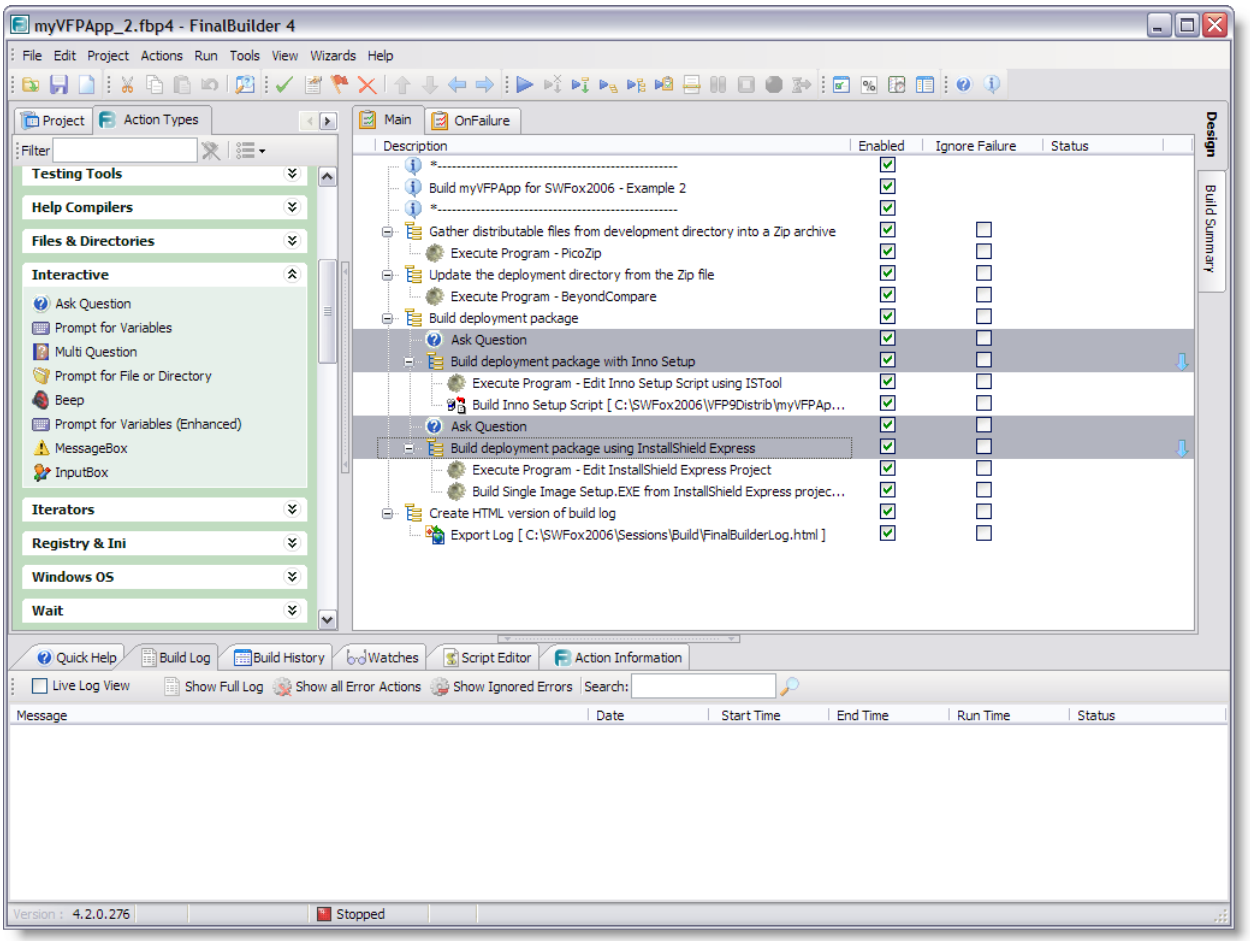

***Figure 15**: Final Builder can interact with the developer during the build process via the Ask Question action.*

In order to make this work, you need to set up a variable. This is a two-step process. The first step is to create a unique variable name and associate it with the answer to the Ask Question action. Do this by opening the Ask Question action's properties sheet and configuring it to put the answer to the question in a variable named DoCompileInno. Because it's a yes/no question, DoCompileInno is a Boolean value.

The second step is make execution of the *Build deployment package with Inno Setup* action group dependent on the value of the DoCompileInno variable. Do this by opening the action group's properties sheet and setting the condition to DoCompileInno.

Although they're actually two separate steps, **Figure 16** shows these two configuration dialogs side by side for convenience.

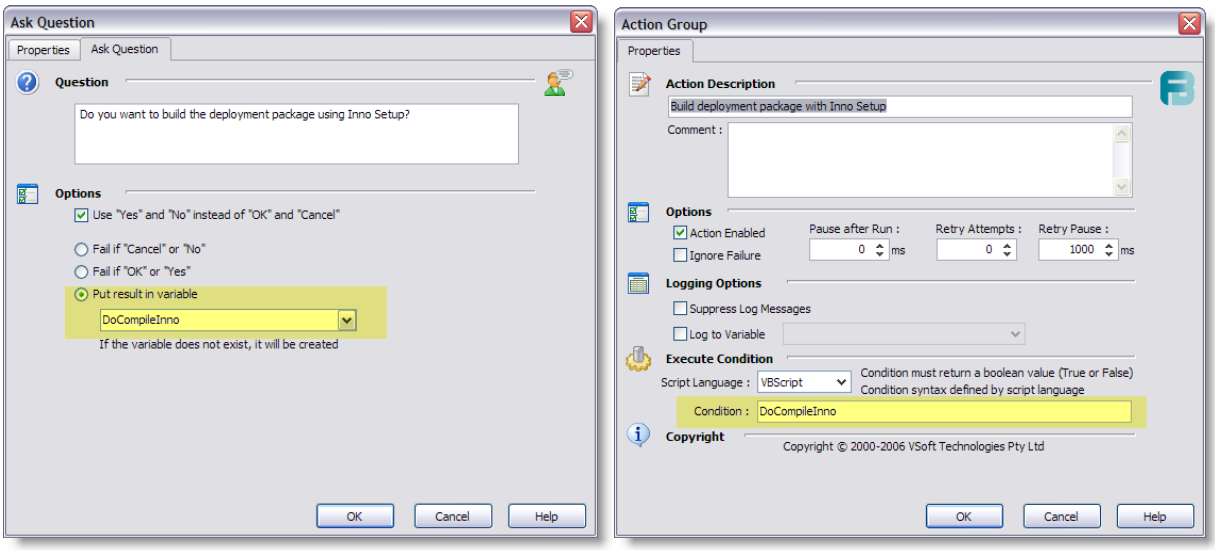

***Figure 16**: Use a variable to make an action group dependent on the answer to a question.*

Doing it the way I just showed requires two questions, one for each install builder. Would you prefer to ask the question only once and be able to capture both answers at the same time? Final Builder's Multi Question action can do that.

Configuring the Multi Question is similar to configuring the individual Ask Question actions, except that you set up both questions and capture both answers in one dialog. **Figure 17** shows the Multi Question action's properties sheet configured for our example.

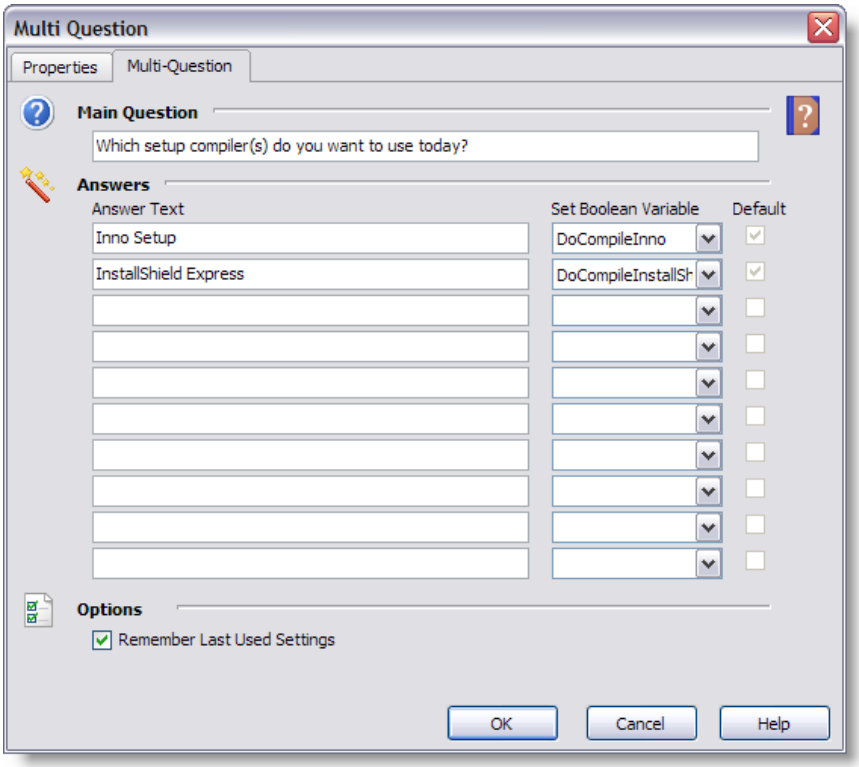

***Figure 17**: Use the Multi Question action to ask two or more questions in the same dialog.*

Figure 18 shows the modified script after replacing the two individual Ask Question actions seen in Figure 15 with a Multi Question action.

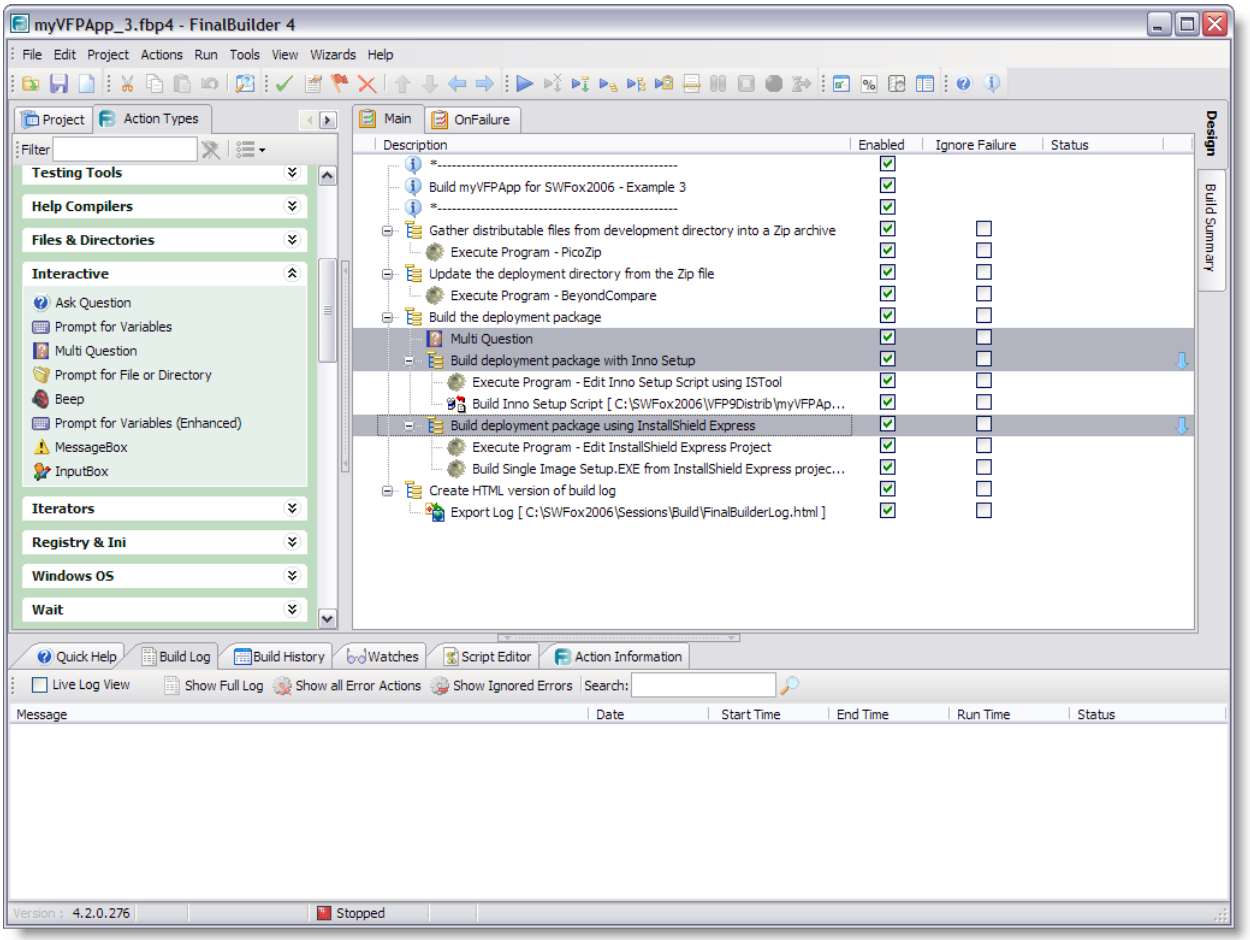

***Figure 18**: Streamline the build by replacing the two individual Ask Question actions with one Multi Question action.*

One of Final Builder's powerful features is its ability to create and use variables, making it possible to construct a truly dynamic build process. Variables are categorized as Project, User, System, or Environment. Final Builder comes with several predefined System and Environment variables such as COMPUTERNAME, SYSDIR, APPDATA, HOMEPATH, ProgramFiles, and CommonProgramFiles, just to pick a few at random. You can create, modify and use your own Project and User variables. System and Environment variables can be referenced but not modified.

Once your Final Builder project is configured the way you want it, you can run it by pressing F9 or clicking the Run button on the toolbar. Running the build process shifts focus to the Build Summary view in the Final Builder IDE. As it runs, progress is displayed in the Build Summary view. **Figure 19** captures the sample project part way through the build process.

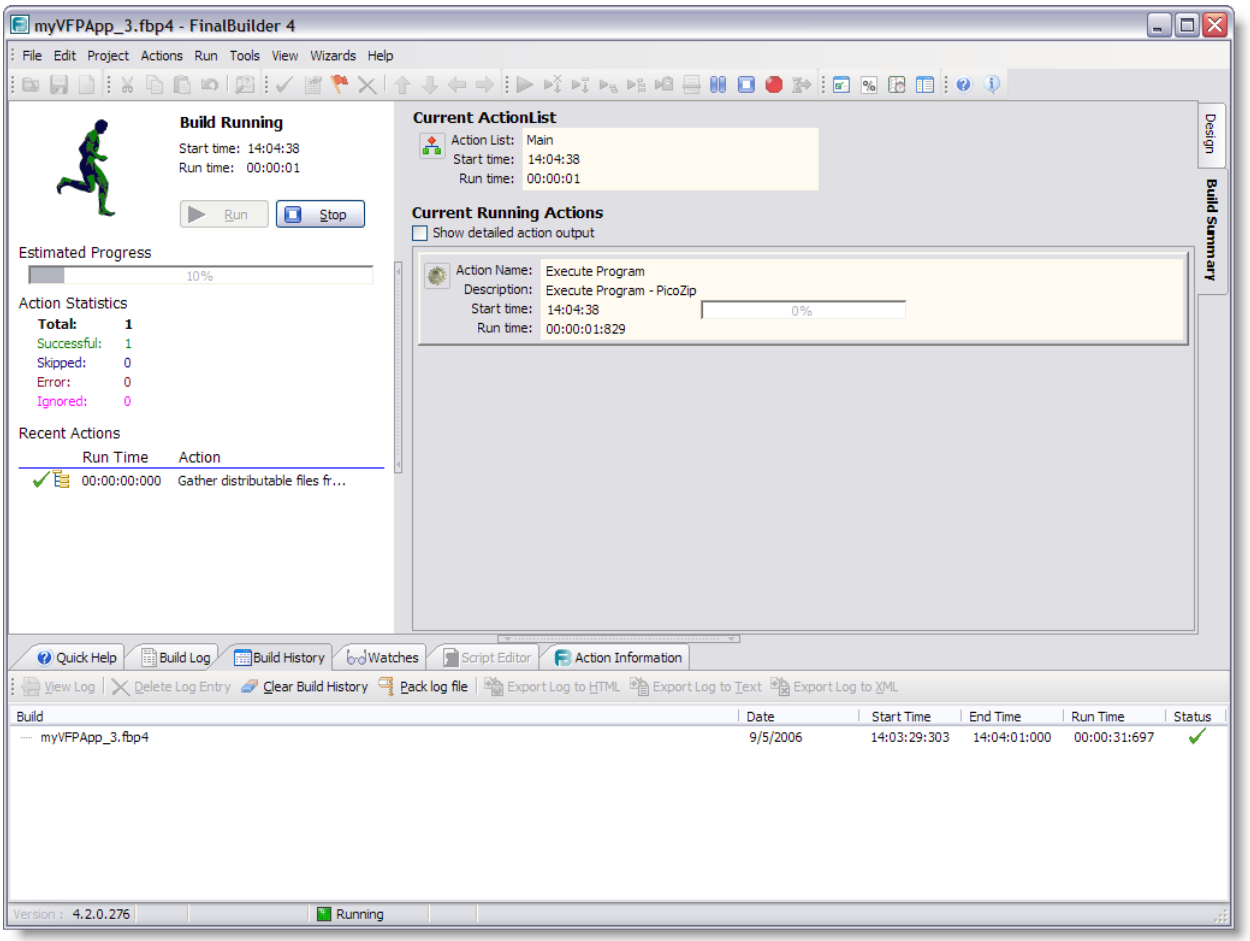

***Figure 19**: The Build Summary view displays progress for each step as the build process runs.*

When complete, the Build Summary view shows a success or failure indication. **Figure 20** shows the result of a successful build. The Action Statistics section summarizes the number of actions that were run successfully, skipped, or had errors. The individual actions and the time they took to run are listed under Recent Actions.

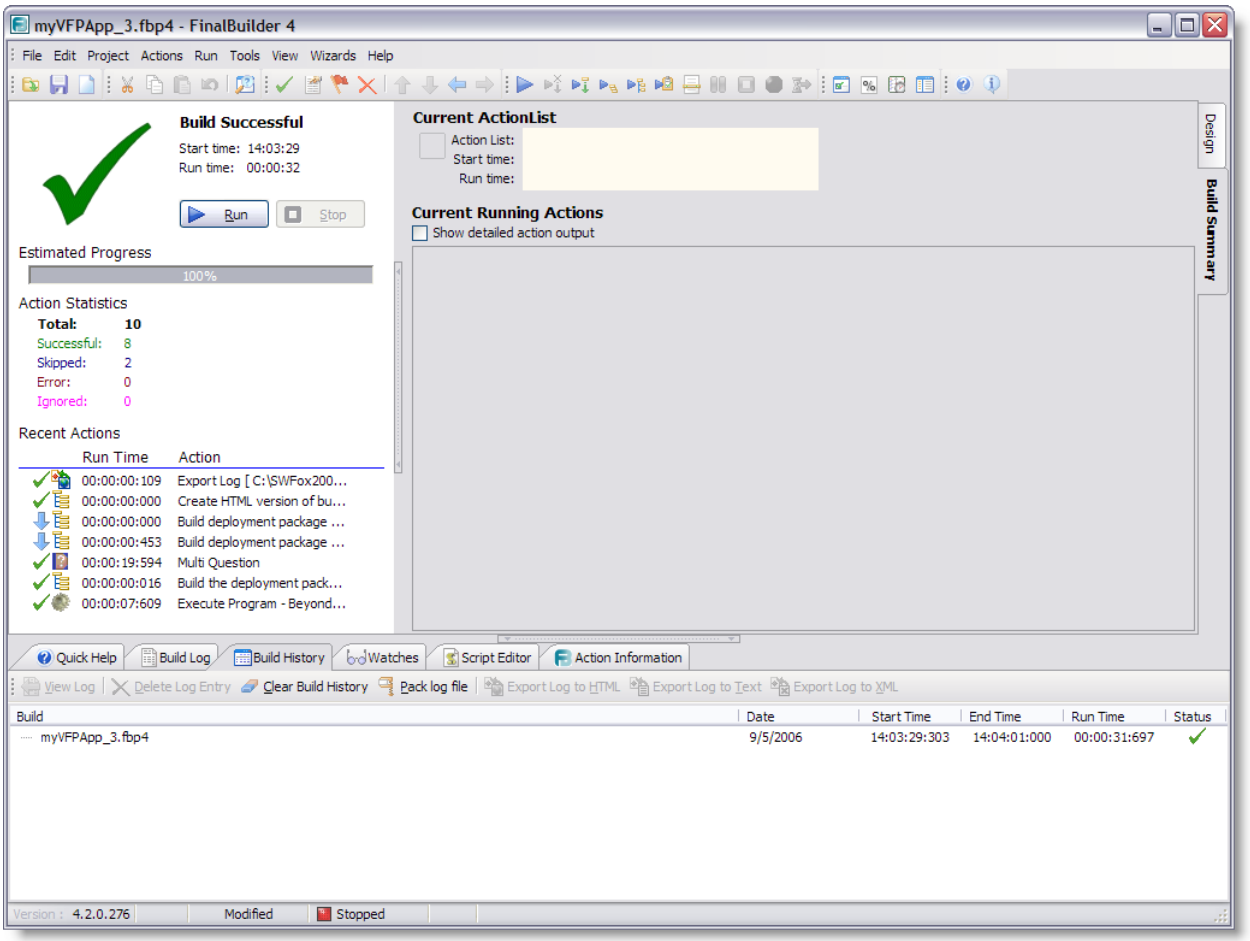

***Figure 20**: Actions are summarized and listed when the build is complete.*

Final Builder also has many other powerful features including Try…Catch error handling, On Failure actions, logging, and much more. I've only scratched the surface of what Final Builder can do here, but I hope this introduction and examples have been enough to get you excited about exploring this tool further on your own.

Visual Build Pro

The descriptions and screenshots here are from Visual Build Pro version 4.2.0.276, which is the latest version available as of this writing. You can download an evaluation copy of Final Builder from the publisher's website at www.finalbuilder.com.

Start by creating a new Visual Build Pro project files and adding actions to it. Visual Build Pro project files are stored with a .bld file name extension. I created a simple Visual Build Pro project to build the deployment package for myVFPApp. The build process follows the same steps used in the other examples in this paper. **Figure 21** shows this project open in the main Visual Build Pro window.

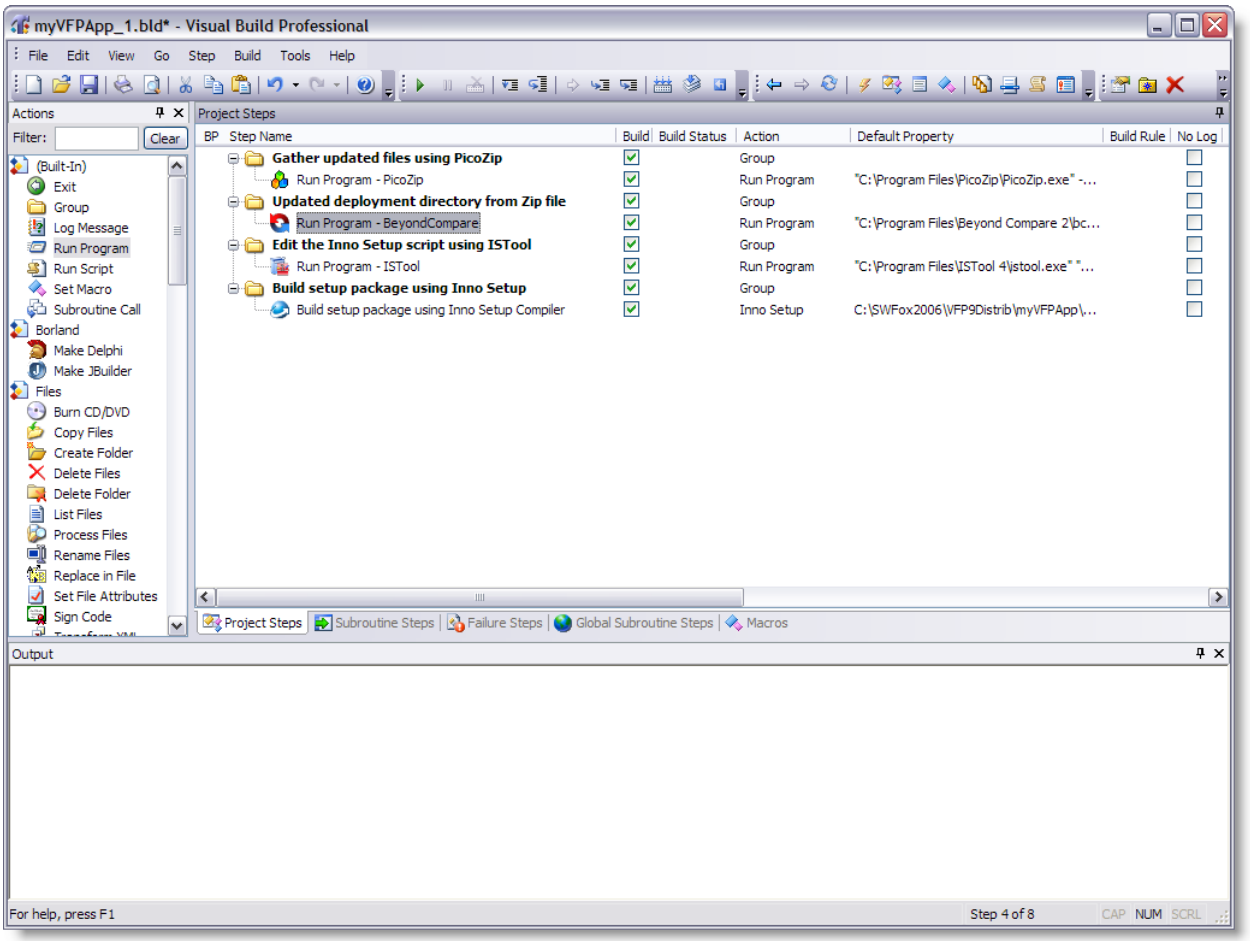

***Figure 21**: The Visual Build Pro main window lists available actions on the left and shows the contents of the current project on the right. The lower portion is where output is capture and displayed as the steps are run.*

The Run Program action can be used to launch a program from the command line. This makes it the ideal action to use when converting the steps from the .cmd file we built earlier. In Figure 17, you can see the Run Program – Beyond Compare action selected. This and other actions are configured by opening the action's properties sheet. **Figure 22** shows the properties sheet for the selected action.

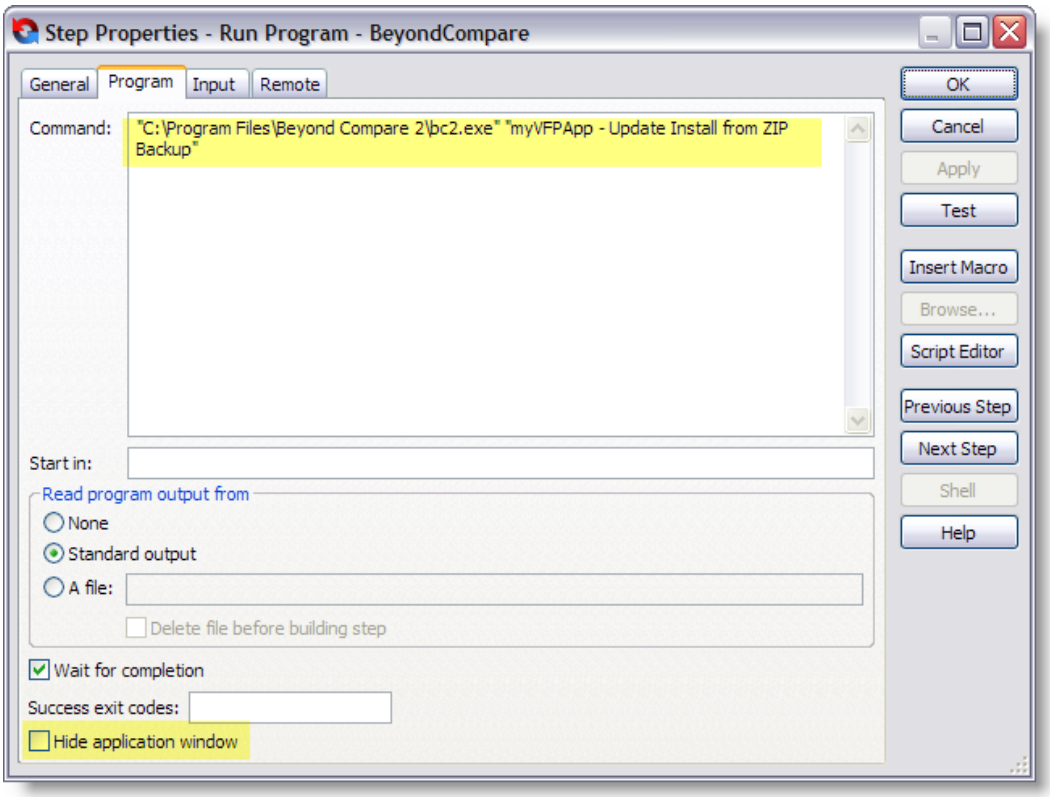

***Figure 22**: Actions are configured by setting the desired options in the action's Properties sheet.*

The Command area of the Properties sheet contains the same command we used in the .cmd file earlier. Near the bottom of the Properties sheet are three options you'll want to pay attention to for each step. When marked, the *Wait for completion* check box tells Visual Build Pro to wait for this step to finish before continuing with the next step. This is normally what you want to do. The default success exit code for most processes is zero, but if different for some process you need to run you can enter the appropriate value in the *Success exit code* field.

The *Hide application* window check box is marked by default, but if the step requires user interaction, as is the case with the Beyond Compare step shown in Figure 22, you should unmark this check box. However, Visual Build Pro detects when a Windows executable is being run and always opens it with its window visible, so even if you forget to unmark this check box you'll be able to interact with the application.

Visual Build Pro is aware of and has pre-configured steps for many popular programs and utilities commonly used in the build process. For example, Visual Build Pro is aware of several installers including Inno Setup and InstallShield. The last step shown in Figure 22 is to build the setup package using Inno Setup. This step is configured not as a generic Run Program action but as an Inno Setup action, as shown in **Figure 23**.

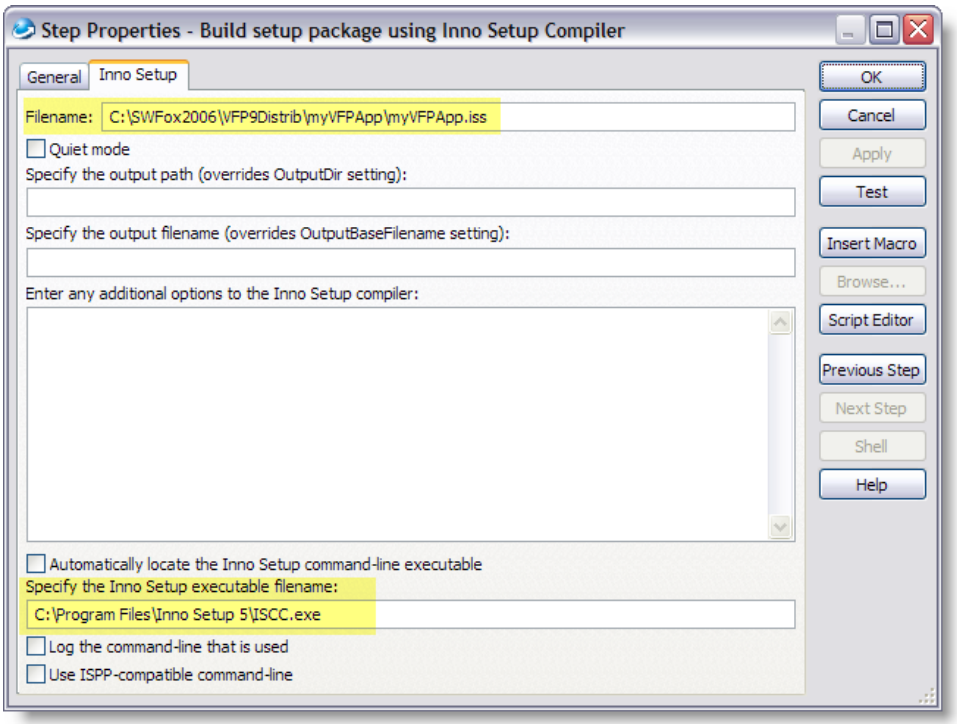

***Figure 23**: Visual Build Pro is aware of and has pre-configured actions for many popular software packages commonly used in the build process, such as Inno Setup.*

Figure 23 highlights where you specify the Inno Setup script you want this step to run, and also where you specify the location of the Inno Setup compiler. There are other options for overriding the default output path and filename, if desired.

In the Visual Build Pro main window shown in Figure 21, you can see there is a check box labeled Build to the right of each step. Marking and unmarking these checkboxes enables you to control which steps are performed when you run the build. Marking or unmarking the Build check box for a group automatically makes the same change to the steps within that group.

When your project is configured the way you want it, you can run it by pressing F7 or clicking the Build button on the toolbar. Progress is indicated in the main window as each step is run, and console output (if any) from each step is captured and displayed in the output portion of the main window. **Figure 24** shows the state of the Visual Build Pro window after successfully running the sample build.

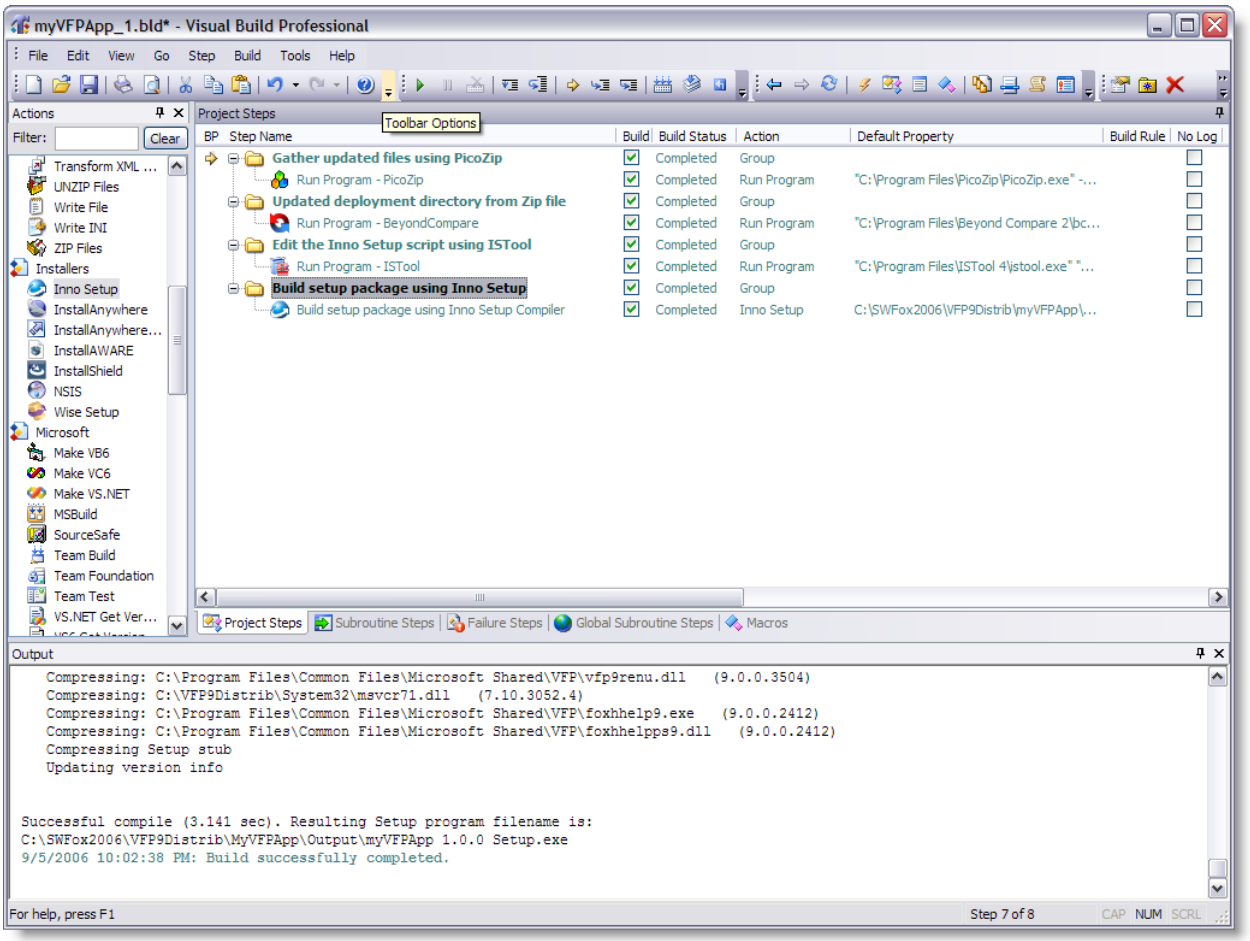

***Figure 24**: The Visual Build Pro main window shows all steps successfully completed. The output window captures and displays the console output from each step along with a log entry for each step.*

As with the section on Final Builder, I've barely scratched the surface of what Visual Build Pro can do for you here. If you're interested in learning more, download an evaluation copy of Visual Build Pro and start exploring!

A final note

Although the professional tools I've introduced here are entirely capable of creating a fully automated build, the examples I've given are deliberately simplified for the sake of illustration. The examples here do not constitute what I would consider a fully automated build, primarily because they still require a good deal of user interaction.

Moving from these examples to what might be considered a more fully automated build would involve at least reconfiguring the Beyond Compare step to copy the files without a visible window and without user interaction. And of course, there are numerous other ways to approach the entire build process that don't involve utilities like PicoZip and Beyond Compare at all.

Regardless of how fully you automate the build process, is may not be possible to get away from user interaction entirely. Unless you require a build process that can run completely unattended—

for example, if it needs to run overnight—a small amount of user interaction should not be considered a bad thing.

How much to automate?

In the end, the question you're likely to ask yourself is "How much should I automate my build process?" The answer, of course, is that it's entirely up to you. In this paper I've showed you several ways to automate the parts of the build process, and you've probably formed your own ideas by now about how far you want to go with this. You've probably also had new ideas about ways to automate things I haven't even mentioned here.

As you consider how much to automate your own build process, keep in mind the benefits you're trying to achieve: a standardized, reliable, and repeatable way of performing the build. How you get there is up to you.

Conclusion

As you move from manual builds to semi-automated builds to fully automated builds, you benefit by saving time, reducing your work load, reducing or eliminating opportunities for manual error, improving reliability, and creating intrinsic documentation for your build process. It takes some effort to put together an automated build process, but there are tools to help you accomplish this and once you've done it you'll appreciate its value each time you build your deployment package.

About the author

Rick Borup is an independent developer specializing in the design, development, and support of mission-critical business software solutions for small to medium-size businesses. Rick earned B.S. and M.B.A. degrees from the University of Illinois at Urbana-Champaign, and is owner and president of Information Technology Associates in Champaign, Illinois. He has been developing solutions with FoxPro/Visual FoxPro (VFP) full-time since 1993, and is a Microsoft Certified Solution Developer (MCSD) and a Microsoft Certified Professional (MCP) in VFP. Rick is co-author of the books *Deploying Visual FoxPro Solutions* and *Visual FoxPro Best Practices for the Next Ten Years,* both from Hentzenwerke Publishing. He is technical editor for the Advisor Discovery column in Advisor Guide to Microsoft Visual FoxPro (formerly FoxPro Advisor) and a frequent speaker at VFP conferences and user groups.