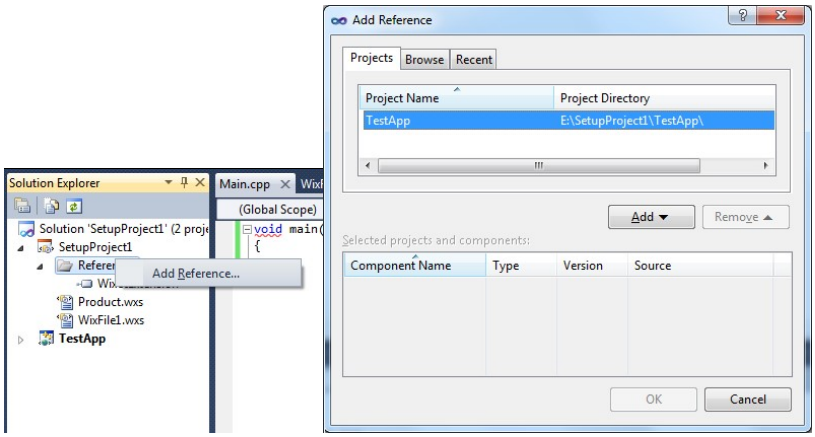
Wix Tips & Tricks

Wix is powerful, and I wouldn't recommend any other installation build system for serious projects. Infinitely customizable, scriptable, uses standard MSI files, doesn't cost anything and maybe best of all, integrates directly with Visual Studio. Both Wix itself and the MSI system it's built on introduce a number of quirks though, and the learning curve is pretty, pretty steep. And then there's learning the right mindset: one file per component, preferably added by hand. Here's some miscellaneous tips; nothing too advanced, but worth knowing all the same.

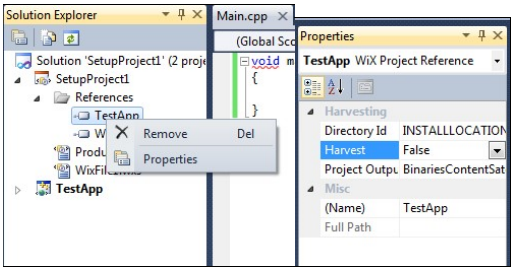
Project references

References to other projects in the solution are perhaps the single most important thing about using Wix, and what makes it so great to use. Adding a reference is simple:



Projects you reference will be built with the installer, and in the same configuration (debug/release/whatever) too. Takes care of build order issues instantly.

And references have properties too! You can set Wix to automatically harvest files that belong to a project using HEAT. Could be useful if your project includes COM information HEAT is able to process, though I've mostly used HEAT from pre-build steps to harvest things such as a directory full of documentation files. Haven't had much luck using HEAT on self-registering native .exe servers, either.



Anyway, each reference defines a number of variables you can use in your wxs files. In my case, I can now do things such as

```
<Component Id="Component1" Directory="INSTALLLOCATION" >
   <File Source="$(var.TestApp.TargetPath)" />
</Component>
<Component Id="Component2" Directory="INSTALLLOCATION" >
   <File Source="$(var.TestApp.ProjectDir)readme.txt" />
</Component>
```

Really cuts down on hard-coded information and messing around with paths relative to your installer.

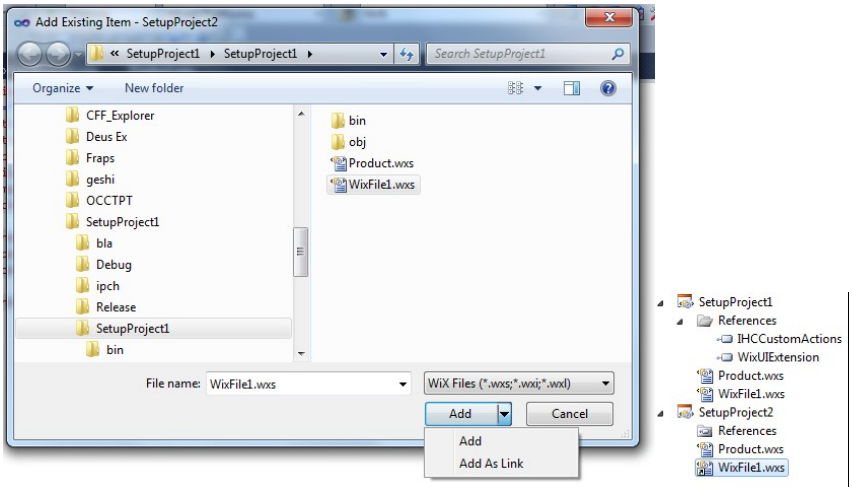
Installer version = file version, version in product name

Using the special `!(bind.FileVersion.)` variable, you can set your installer to have the same version as a certain file it references. I also like appending the version to the product name, so it's clear what's being installed, and having the product name as a macro that can be reused throughout the installer. For clarification I've defined `PRODUCT` in code here, but it's better to do it in the project options: that way, the macro is global to all files in the project.

```
<?define PRODUCT="Foo"?>
<?define MAINEXE=$(var.TestApp.TargetFileName)?>
<?define VERSION=!(bind.FileVersion.$(var.MAINEXE))?>
<Product Id="*" Name="$(var.PRODUCT) $(var.VERSION)" Language="1033" Version="$(var.VERSION)" Manufacturer="Marijn Kentie" UpgradeCode="PUT-GUID-HERE">
```

Adding files as links

Unlike for Visual C++, if you add an existing file to a Wix project, it copies it. Adding the file as a link is extremely simple; if you don't have the habit of overlooking additional Ok button options in Common Item Dialogs, that is.



ComponentGroups and Directories

One annoying thing about the Wix schema is that you can't have a *ComponentGroup* inside a *DirectoryRef* element, or vice versa. Both *ComponentGroup* and *DirectoryRef* elements are critical for larger projects, where a *Feature* might share component groups, which in turn are defined in different files. This means that you tend to see code such as

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Fragment>
    <DirectoryRef Id="INSTALLLOCATION">
      <Component Id="Component1" >
        <File Source="e:\bla.txt" />
      </Component>
      <Component Id="Component2" >
        <File Source="e:\bla2.txt" />
      </Component>
    </DirectoryRef>

    <ComponentGroup Id="ComponentGroup1">
      <ComponentRef Id="Component1"/>
      <ComponentRef Id="Component2"/>
    </ComponentGroup>

  </Fragment>
</Wix>
```

Note how the *ComponentGroup* must contain a *ComponentRef* for each component; in other words, for each component added, you must edit the file in two places, which is annoying. Fortunately, since version 3, Wix supports a *Directory* attribute for components:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Fragment>
    <ComponentGroup Id="ComponentGroup1">
      <Component Id="Component1" Directory="INSTALLLOCATION" >
        <File Source="e:\bla.txt" />
      </Component>
      <Component Id="Component2"  Directory="INSTALLLOCATION" >
        <File Source="e:\bla2.txt" />
      </Component>
    </ComponentGroup>

  </Fragment>
</Wix>
```

Much neater. The only disadvantage is that the directory must be repeated for each *Component*, but in practice I don't find this much of an issue. There's always the find/replace function, after all.

The MajorUpgrade element

Upgrades, updates and patches are one of the most complex parts of MSI authoring. I tend to only use Major Upgrades (which uninstall and then re-install the product), for the following reasons:

- They require the Product Id GUID to be changed for each version, which is easy in Wix by setting it to `'*'`. Also releasing smaller updates means the Product Id must be updated in some cases, but not others, meaning a more complex workflow.
- They allow auto-generated GUIDs to be used with components, which not only makes it less of a hassle to build wxs files, it also means that HEAT harvesting can be done as a pre-build step that re-runs for each build of the installer.
- Major Upgrades allow all the features/components for the installer to be reshuffled, making it easier to improve the installer as development goes along.

Of course, the disadvantage is that you need to distribute a large MSI file even for simple updates.

Wix 3.5 introduced the *MajorUpgrade element*, which is easier to use than messing around with separate *Upgrade* and *Upgrade Version* elements. Here's how I use it (note that PRODUCT is a macro defined in the project options):

```
<Product Id="*" ... />
...
<MajorUpgrade DowngradeErrorMessage="A newer version of $(var.PRODUCT) is already installed; please uninstall it and re-run setup." AllowSameVersionUpgrades="yes" />
```

Watch out: the *Id* attribute of the *Product* element must still be set to `'*'`! Also note the *AllowSameVersionUprades* attribute. MSI installers ignore the fourth digit of the product version. By default, MSI installers of which only the fourth version digit differs, will be installed next to each other. This leads to confusing situations with duplicate entries in add/remove programs, etc. By setting *AllowSameVersionUprades* to true, these versions will be installed over one another like usual, with the side effect that it's possible to downgrade to versions with a lower fourth digit. For versioning systems where the fourth digit is, for instance, a branch number, this might actually be a positive thing.

Note that *AllowSameVersionUpgrades="yes"* does generate an ICE61 warning. To disable it, add it to the ignored ICE validations in the Wix project properties (under 'Tool Settings'). It's hard to make any nontrivial installer without ignoring at least a few ICE warnings though, especially when third-party merge modules are involved.

Disable Wix projects in Debug builds

Wix projects can take pretty long to build, and aren't always really needed. I suggest unloading them, or their solution folder (right click->'Unload Projects In Solution Folder') when they aren't needed. They can be completely disabled in debug mode by going to the solution configuration dialog and disabling them for Debug/All Platforms.

Wix and Team Foundation Server Team Build

Working with TFS Team Build can be a pain, and getting it to play nice with Wix took some experimentation, but it's not that hard:

1. Make sure Wix projects will be built for the configuration you want to use. By default and annoyingly, Wix projects are only built for the 'Mixed Platforms' platform, not 'Win32'. Make sure your build server's set to use 'Mixed Platforms', or enable the Wix projects for 'Win32'.

2. Make sure Wix is installed on your build system(s). Originally I wanted to avoid having to do that, and followed the steps described here: checking Wix into the source tree. This came with some major disadvantages though: having to edit each .wixproj file, and the Wix binaries/Team Build not playing nice when the time came to clean up:

3. Register Microsoft.Deployment.WindowsInstaller.dll with the GAC, otherwise Team Build won't be able to find it, somehow. Start a VS2010 command prompt as administrator:

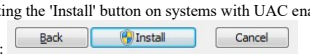
```
C:\Program Files (x86)\Windows Installer XML v3.5\bin>gacutil -i Microsoft.Deployment.WindowsInstaller.dll
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.1
Copyright (c) Microsoft Corporation. All rights reserved.

Assembly successfully added to the cache
```

One disadvantage is that the Wix project output file directory is ignored and that your installers will be dumped among all the other binaries. Can be solved using a post-build step if you're so inclined.

UAC shield on Install button

By default, Wix installers ask for administrator privileges after hitting the 'Install' button on systems with UAC enabled, but don't show the UAC shield on the button. By setting the *InstallScope* attribute of the *Package* element to `perMachine`, the shield can be made to appear:




```
<Package InstallerVersion="200" Compressed="yes" InstallScope="perMachine" />
```

Unfortunately, it seems that through no combination of *InstallScope* and/or *InstallPrivilege* settings can installers be made to only ask for administrator privileges when those are actually required, depending on the install directory. As it stands now, perUser installers will just plain fail when installed to, for example, the Program Files directory, while perMachine installers always ask for admin rights.

TARGETDIR and the system partition

When trying to install to a subdirectory of the system drive root (e.g. '`C:\application`'), it might sense to assume that in something like

```
<Directory Id="TARGETDIR" Name="SourceDir">
  <Directory Id="INSTALLLOCATION" Name="SetupProject1">
  </Directory>
</Directory>
```

`TARGETDIR` refers to the system partition, as `ProgramFilesFolder` is always given as a child of `TARGETDIR`. This is not the case; `TARGETDIR` is the partition with the most free disk space. It can even be a partition on an external hard drive. To set it to the true system partition, use the below approach:

```
<Directory Id="TARGETDIR" Name="SourceDir">
  <Directory Id="WINDOWSVOLUME" >
    <Directory Id="INSTALLLOCATION" Name="SetupProject1">
    </Directory>
  </Directory>
</Directory>

<SetDirectory Id="WINDOWSVOLUME" Value="[WindowsVolume]"/>
```

The *SetDirectory* element is required as trying to use `WindowsVolume` directly results in

```
error LGHT0204: ICE99: The directory name: WindowsVolume is the same as one of the MSI Public Properties and can cause unforeseen side effects.
```

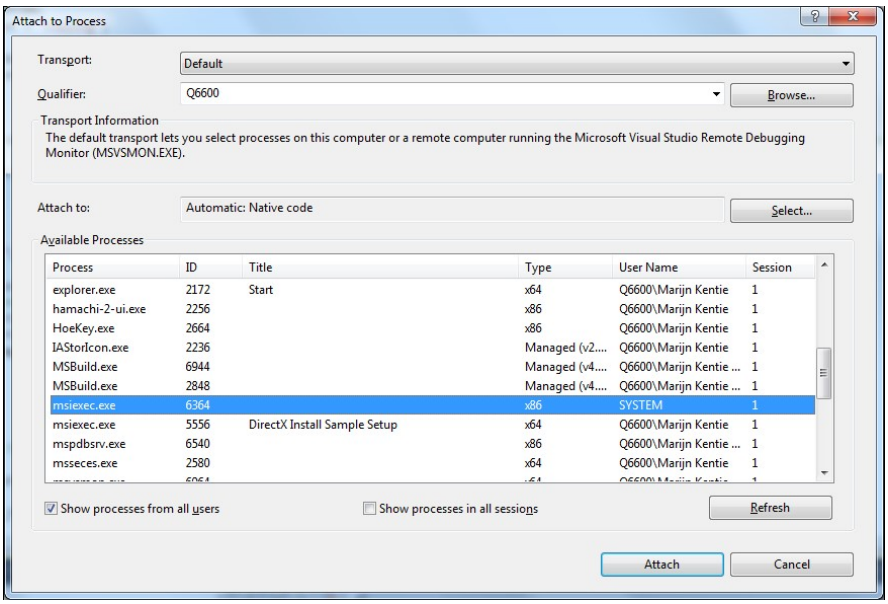
Signing MSIs

If you sign your MSI files, they'll get a nice professional UAC prompt that verifies the package as being yours. Once you've got a code signing key in .pfx format, signing an MSI easy to do as a Wix project post-build step. Make sure to pass a description for the package, as otherwise the UAC prompt will show some mangled temporary filename due to how msiexec works. For example:

```
signtool /d "MyProduct Setup" /f "$(SolutionDir)\sign\cert.pfx" /p password1234
```

Debugging Custom Actions

One easy way to debug dll custom actions is to have them spawn a message box using the `MessageBox` Windows call and then attach a debugger. Make sure the custom action in question has been built in debug mode; if you're using project references to include the custom action dll, a debug build of the Wix project will include a debug version of the dll, as well. Have it spawn a message box near where you want to debug, and attach the Visual Studio debugger. Note that when running a 32-bit custom action on a 64-bit version of Windows, a new `msiexec` instance will have spawned to run the CA; check the 'Show processes from all users' box to find it. Once the debugger has attached, place a breakpoint somewhere, hit F5 to resume, click 'Ok' on the message box and you're in business.



Created: Jan 18 2012
Modified: Feb 12 2012