

WiX Techniques

This whitepaper presents techniques that I've learnt while integrating WiX into my build scripts.

At the moment, this paper contains just two sections:

NAnt and Versioning discusses one way that versioning can be injected into a WiX installation from a NAnt built script.

Installations that Upgrade reviews the necessary elements for installations that cleanly upgrade.

If you find the following information useful, or if you have suggestions for improvement, please send an email to feedback@nichesoftware.co.nz.

NAnt and Versioning

In most build scenarios, the build script is in charge of versioning.

For example, the following NAnt target updates the master build number for a system, leaving the result in the property `build.version`.

```
sample.build
<target name="compile.version"
  description="Update build version">
  <version startdate="1 Jan 2006"
    path="version.txt"
    prefix="build"/>
</target>
```

Once this target has executed, the next step is to inject the version number into our assemblies and our installers. For .NET assemblies, NAnt has a very useful `<asminfo>` task which can be used generate source files for C# and Visual Basic that contain the version number.

To achieve the same results for our WiX files, the `<echo>` task comes to the fore.

```
sample.build
<target name="compile.installer">
  <!-- Generate a version.wxs file -->
  <echo file="{wix.src.dir}\version.wxs"
```



Bevan Arps (bevan@nichesoftware.co.nz) is a professional software developer and self confessed geek based in Wellington, New Zealand. With a career that spans analysis to testing, hardware installation to user training, and tech support to technical writing, he is currently he is a C#/.NET developer working for the Reserve Bank of New Zealand. Bevan's blog can be read online on his website, www.nichesoftware.co.nz.

Document
Version

1.2

Last
Update

October
2008

```

    message="&lt;Include&gt;"/>
    <echo file="{wix.src.dir}\version.wxs" append="true"
    message="&lt;?define version=&quot;${build.version}&quot; ?&gt;"/>
    <echo file="{wix.src.dir}\version.wxs" append="true"
    message="&lt;/Include&gt;"/>
    ...
</target>

```

The first `<echo>` statement overwrites any existing `version.wxs` file, and the following two `<echo>` statements append additional lines. If you decipher all the XML encoding, you'll see that the output file will look something like this:

```

<Include>
  <?define version="1.2.3.4" ?>
</Include>

```

version.wxs

With this version file in place, the version can be pulled into the main `WXS` file with an include directive.

```

<?xml version="1.0" encoding="utf-8"?>
<wix xmlns="http://schemas.microsoft.com/wix/2003/01/wix">
  ...
  <?include version.wxs ?>
  ...
</wix>

```

installer.wxs

The last step is to refer to the version number where required, replacing the hard coded values. As a minimum, replace the `Version` attribute on the `Product` element.

```

<Product Id="?????????-????-????-????-????????????????"
  Name="My Example Product"
  Language="1033"
  Version="$(var.version)"
  Manufacturer="Niche Software"
  UpgradeCode="$(var.UpgradeCode)">
  ...
</Product>

```

installer.wxs

The version number can also be injected into the filename of the final MSI, as can be seen in this build fragment.

```

<target name="compile.installer">
  ...
  <exec program="lib\wix\candle.exe" verbose="true">
    <arg value="-out"/>
    <arg file="{wix.tmp.dir}\rbquery.wixobj"/>
    <arg file="{wix.src.dir}\rbquery.wxs"/>
  </exec>

  <exec program="lib\wix\light.exe" verbose="true">

```

sample.build

```

<arg value="-out"/>
<arg file="{wix.build.dir}\rbquery-{$build.version}.msi"/>
<arg file="{wix.tmp.dir}\rbquery.wixobj"/>
</exec>

</target>

```

One catch to be aware of is that changing the name of the MSI file requires that each newer version of the installer be a major upgrade. If you want to perform minor updates or patches with your MSIs, the name of the MSI must remain unchanged.

Installations that Upgrade

When your build script is producing an installer, you want each installation to install cleanly, even if a previous version was already present. The most straightforward way to achieve this with an MSI installer is to build each as a **major upgrade** (in MSI terms) which removes any existing installation automatically. Achieving this requires a number of elements to interact in a precise manner.

To begin, use your favourite tool to define an UpgradeCode – a unique GUID that must remain constant through all versions of the product.

For ease of reference (since the upgrade code needs to be repeated in a couple of locations), define the upgrade code as a variable at the top of your WXS file.

```
<?define UpgradeCode="{YOURGUID-GOES-HERE-9586-B92717D03E90}"?>
```

installer.wxs

This ensures that every reference is identical, avoiding some subtle potential bugs.

The UpgradeCode should remain unchanged throughout the lifetime of your product – it is the one thing in the installer that remains constant. Also, the Upgrade code should be unique to the product you're installing – never reuse the same upgrade code for multiple different product lines.

Note: You should always include an upgrade code in every installer, even if you do nothing else. Without an upgrade code, it is impossible for any future installer to be an upgrade.

The upgrade code variable should then be referenced from the Product element.

```

<Product Id="YOURGUID-GOES-HERE-0123-012345678901"
  Name="My Example Product"
  Language="1033"
  Version="$(var.version)"
  Manufacturer="Niche Software"
  UpgradeCode="$(var.UpgradeCode)">
  ...
</Product>

```

installer.wxs

With the upgrade code in place, the next step is to ensure that new product and package GUIDs are used for each installer built.

Fortunately, the WiX toolset provides a simple way to achieve this – instead of supplying a hard coded Id for each, specify a wildcard to request auto-generation of an appropriate GUID.

For WiX 2.x, use a sequence of “?” characters like this:

```

installer.wxs
<Product Id="?????????-????-????-????-?????????????"
  Name="My Example Product"
  Language="1033"
  Version="$(var.version)"
  Manufacturer="Niche Software"
  UpgradeCode="$(var.UpgradeCode)">
...
</Product>

```

For WiX 3.x, a simpler wildcard “*” can be used instead.

```

installer.wxs
<Product Id="*"
  Name="My Example Product"
  Language="1033"
  Version="$(var.version)"
  Manufacturer="Niche Software"
  UpgradeCode="$(var.UpgradeCode)">
...
</Product>

```

The same autogeneration needs to be applied to the Package element.

```

installer.wxs
<Package Id="?????????-????-????-????-?????????????"
  Description="Installer for Example Product"
  InstallerVersion="200"
  Compressed="yes" />
...
</Package>

```

Again, if you are working with WiX 3.x, use “*” instead.

As a part of the installer, include an Upgrade section to define upgrade policy.

```

installer.wxs
<Upgrade Id="$(var.UpgradeCode)">
  <!-- Detect any newer version of this product -->
  <UpgradeVersion Minimum="$(var.version)"
    IncludeMinimum="no"
    OnlyDetect="yes"
    Language="1033"
    Property="NEWPRODUCTFOUND"/>
  <!-- Detect and remove any older version of this product -->
  <UpgradeVersion Maximum="$(var.version)"
    IncludeMaximum="yes"
    OnlyDetect="no"

```

```

        Language="1033"
        Property="OLDPRODUCTFOUND"/>
</Upgrade>

```

Using the UpgradeCode and Version variables, this policy may define one of two properties. If an installation of this product with a greater version number is found, the property NEWPRODUCTFOUND will be defined. If an installation of this product with a lower or equal version number is found, the property OLDPRODUCTFOUND will be defined.

The policy shown here is sufficient to automatically uninstall any old version when installing a newer one.

To prevent downgrading, add the following.

```

                                                                    installer.wxs
<!-- Define a custom action -->
<CustomAction Id="PreventDowngrading"
              Error="Newer version already installed."/>

<InstallExecuteSequence>

  <!-- Prevent downgrading -->
  <Custom Action="PreventDowngrading"
          After="FindRelatedProducts">NEWPRODUCTFOUND</Custom>
  <RemoveExistingProducts After="InstallFinalize" />

</InstallExecuteSequence>

<InstallUISequence>

  <Custom Action="PreventDowngrading"
          After="FindRelatedProducts">NEWPRODUCTFOUND</Custom>

</InstallUISequence>

```

With all of these elements in place, each new build of your installer should cleanly replace the previous.

In Closing

If you found this whitepaper useful, or if you have suggestions for improvement, please send an email to feedback@nichesoftware.co.nz.