Version: v3

How To Guides

This section includes How To documentation for performing common WiX tasks.

Files, Shortcuts and Registry

- Add a file to your installer
- Check the version number of a file during installation
- Write a registry entry during installation
- Read a registry entry during installation
- Create a shortcut on the Start Menu
- Create a shortcut to a web page
- Create an uninstall shortcut
- NGen managed assemblies during installation
- Reference another DirectorySearch element
- Get the parent directory of a file search

Redistributables and Install Checks

- Check for .NET Framework versions
- Install the .NET Framework using a bootstrapper
- Install DirectX 9.0 with your installer
- Install the Visual C++ Redistributable with your installer
- Block installation based on OS version

User Interface and Localization

- Build a localized version of your installer
- Make your installer localizable
- Run the installed application after setup
- Set your installer's icon in Add/Remove Programs

Product Updates

- Implement a major upgrade in your installer

Others

- Get a log of your installation for debugging
- Look inside your MSI with Orca

- Generate a GUID
- Use WiX Extensions
- Optimize building cabinet files
- Specify source file locations
- Install a windows service

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Add a File To Your Installer

Installing files is the most fundamental aspect of any installer, and is usually what leads people to build an installer in the first place. Learning how to place a file on disk using Windows Installer best practices not only ensures maintainability going forward, but also enables you to build patches later if necessary.

Step 1: Define the directory structure

Installers frequently have many files to install into a few locations on disk. To improve the readability of the WiX file, it is a good practice to define your installation directories first before listing the files you'll install. Directories are defined using the Directory element and describe the hierarchy of folders you would like to see on the target machine. The following sample defines a directory for the installation of the main application executable.

```
<Directory Id="TARGETDIR" Name="SourceDir">
    <Directory Id="ProgramFilesFolder">
        <Directory Id="APPLICATIONROOTDIRECTORY" Name="My Application Name"/>
    </Directory>
</Directory>
```

The element with the id TARGETDIR is required by the Windows Installer and is the root of all directory structures for your installation. Every WiX project will have this directory element. The second element, with the id ProgramFilesFolder, uses a pre-defined Windows Installer property to reference the Program Files folder on the user's machine. In most cases this will resolve to **c:\Program Files**. The third directory element creates your application's folder under Program Files, and it is given the id APPLICATIONROOTDIRECTORY for later use in the WiX project. The id is in all capital letters to make it a public property that can be set from UI or via the command line.

The result of these tags is a **c:\Program Files\My Application Name** folder on the target machine.

Step 2: Add files to your installer package

A file is added to the installer using two elements: a Component element to specify an atomic unit of installation and a File element to specify the file that should be installed.

The component element describes a set of resources (usually files, registry entries, and shortcuts) that need to be installed as a single unit. This is separate from whether the set of items consist of a logical feature the user can select to install which is discussed in Step 3. While it may not seem like a big deal when you are first authoring your installer, components play a critical role when you decide to build patches at a later date.

In general, you should restrict yourself to a single file per component. The Windows Installer is designed to support thousands of components in a single installer, so unless you have a very good reason, keep to one file per component. **Every component must have its own unique GUID**. Failure to follow these two basic rules can lead to many problems down the road when it comes to servicing.

The following sample uses the directory structure defined in Step 1 to install two files: an application executable and a documentation file.

```
<DirectoryRef Id="APPLICATIONROOTDIRECTORY">
    <Component Id="myapplication.exe" Guid="PUT-GUID-HERE">
        <File Id="myapplication.exe" Source="MySourceFiles\MyApplication.exe" KeyPath="yes"
Checksum="yes"/>
    </Component>
    <Component Id="documentation.html" Guid="PUT-GUID-HERE">
        <File Id="documentation.html" Source="MySourceFiles\documentation.html" KeyPath="yes"/>
    </Component>
</DirectoryRef>
```

find it and build it into the installer.

The KeyPath attribute is set to yes to tell the Windows Installer that this particular file should be used to determine whether the component is installed. If you do not set the KeyPath attribute, WiX will look at the child elements under the component in sequential order and try to automatically select one of them as a key path. Allowing WiX to automatically select a key path can be dangerous because adding or removing child elements under the component can inadvertantly cause the key path to change, which can lead to installation problems. In general, you should always set the KeyPath attribute to yes to ensure that the key path will not inadvertantly change if you update your setup authoring in the future.

The Checksum attribute should be set to yes for executable files that have a checksum value in the file header (this is generally true for all executables), and is used by the Windows Installer to verify the validity of the file on re-install.

Step 3: Tell Windows Installer to install the files

After defining the directory structure and listing the files to package into the installer, the last step is to tell Windows Installer to actually install the files. The Feature element is used to do this, and is where you break up your installer into logical pieces that the user can install independently. The following example creates a single feature that installs the application executable and

documentation from Step 2.

```
<Feature Id="MainApplication" Title="Main Application" Level="1">
    <ComponentRef Id="myapplication.exe" />
    <ComponentRef Id="documentation.html" />
</Feature>
```

The Feature is given a Id. If you are using an installer UI sequence that includes feature selection, the Title attribute contains the text displayed in the UI for the feature. The Level attribute should be set to 1 to enable the installation of the feature by default.

The ComponentRef element is used to reference the components created in Step 2 via the Id attribute.

The Complete Sample

The following is a complete sample that uses the above concepts. This example can be inserted into a WiX project and compiled, or compiled and linked from the command line, to generate an installer.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <Product Id="*" UpgradeCode="PUT-GUID-HERE" Version="1.0.0.0" Language="1033" Name="My Application
Name" Manufacturer="My Manufacturer Name">
        <Package InstallerVersion="300" Compressed="yes"/>
        <Media Id="1" Cabinet="myapplication.cab" EmbedCab="yes" />

        <!-- Step 1: Define the directory structure -->
        <Directory Id="TARGETDIR" Name="SourceDir">
            <Directory Id="ProgramFilesFolder">
                <Directory Id="APPLICATIONROOTDIRECTORY" Name="My Application Name"/>
```

```
            </Directory>
        </Directory>

        <!-- Step 2: Add files to your installer package -->
        <DirectoryRef Id="APPLICATIONROOTDIRECTORY">
            <Component Id="myapplication.exe" Guid="PUT-GUID-HERE">
                <File Id="myapplication.exe" Source="MySourceFiles\MyApplication.exe" KeyPath="yes"
Checksum="yes"/>
            </Component>
            <Component Id="documentation.html" Guid="PUT-GUID-HERE">
                <File Id="documentation.html" Source="MySourceFiles\documentation.html" KeyPath="yes"/>
            </Component>
        </DirectoryRef>

        <!-- Step 3: Tell WiX to install the files -->
        <Feature Id="MainApplication" Title="Main Application" Level="1">
            <ComponentRef Id="myapplication.exe" />
            <ComponentRef Id="documentation" />
        </Feature>
    </Product>
</Wix>
```

Edit this page

Version: v3

How To: Check the Version Number of a File During Installation

Installers often need to look up the version number of a file on disk during the installation process. The check is often used in advance of a conditional statement later in install, such as to block the user from installing if a file is missing, or to display custom installation UI depending on whether the file version is high enough. This how to demonstrates verifying the version of a file on disk, then using the resulting property to block the application's installation if the file version is lower than expected.

Step 1: Determine the version of the file

File versions are determined using the Property, DirectorySearch and FileSearch elements. The following snippet looks for the user32.dll file in the machine's System32 directory and checks to see if it is at least version 6.0.6001.1751.

Searching for a file is accomplished by describing the directories to search, and then specifying the file to look up in that directory.

The Property element defines the Id for the results of the file search. This Id is used later in the WiX project, for example in conditions. The DirectorySearch element is used to build the directory hierarchy to search for the file. In this case it is given a

```
<Property Id="USER32VERSION">
    <DirectorySearch Id="SystemFolderDriverVersion" Path="[SystemFolder]">
        <FileSearch Name="user32.dll" MinVersion="6.0.6001.1750"/>
    </DirectorySearch>
</Property>
```

Important: When doing a locale-neutral search for a file, **you must set the MinVersion property to one revision number lower than the actual version you want to search for**. In this example, while we want to find file version 6.0.6001.1751, the MinVersion is set to 6.0.6001.1750. This is because of a quirk in how the Windows Installer matches file versions. More information is available in the Windows Installer documentation.

Step 2: Use the property in a condition

Once you have determined whether the file exists with the requested version you can use the property in a condition. The following is a simple example that prevents installation of the application if the user32.dll file version is too low.

```
<Condition Message="The installed version of user32.dll is not high enough to support this installer.">
    <![CDATA[Installed OR USER32VERSION]]>
</Condition>
```

Installed is a Windows Installer property that ensures the check is only done when the user is installing the application, rather than on a repair or remove. The USER32VERSION part will pass if the property is set to anything, and will fail if it is not set. The file check in Step 1 will set the property to the full path of the user32.dll file if it is found with an appropriate file version, and will not set it otherwise.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Write a Registry Entry During Installation

Writing registry entries during installation is similar to writing files during installation. You describe the registry hierarchy you want to write into, specify the registry values to create, then add the component to your feature list.

Step 1: Describe the registry layout and values

The following example illustrates how to write two registry entries, one to a specific value and the other to the default value.

```
<DirectoryRef Id="TARGETDIR">
    <Component Id="RegistryEntries" Guid="PUT-GUID-HERE">
        <RegistryKey Root="HKCU"
                     Key="Software\MyCompany\MyApplicationName"
                Action="createAndRemoveOnUninstall">
            <RegistryValue Type="integer" Name="SomeIntegerValue" Value="1" KeyPath="yes"/>
            <RegistryValue Type="string" Value="Default Value"/>
        </RegistryKey>
    </Component>
```

```
</DirectoryRef>
```

The snippet begins with a DirectoryRef that points to the TARGETDIR directory defined by Windows Installer. This effectively means the registry entries should be installed to the target user's machine. Under the DirectoryRef is a Component element that groups together the registry entries to be installed. The component is given an id for reference later in the WiX project and a unique guid.

The registry entries are created by first using the RegistryKey element to specify where in the registry the values should go. In this example the key is under **HKEY_CURRENT_USER\Software\MyCompany\MyApplicationName**. The optional Action attribute is used to tell Windows Installer that the key should be created (if necessary) on install, and that the key and all its sub-values should be removed on uninstall.

Under the RegistryKey element the RegistryValue element is used to create the actual registry values. The first is the SomeIntegerValue value, which is of type integer and has a value of 1. It is also marked as the KeyPath for the component, which is used by the Windows Installer to determine whether this component is installed on the machine. The second RegistryValue element sets the default value for the key to a string value of Default Value.

The id attribute is omitted on the RegistryKey and RegistryValue elements because there is no need to refer to these items elsewhere in the WiX project file. WiX will auto-generate ids for the elements based on the registry key, value, and parent component name.

Step 2: Tell Windows Installer to install the entries

After defining the directory structure and listing the registry entries to package into the installer, the last step is to tell Windows Installer to actually install the registry entry. The Feature element is used to do this. The following snippet adds a reference to the registry entries component, and should be inserted inside a parent Feature element.

```
<ComponentRef Id="RegistryEntries" />
```

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Read a Registry Entry During Installation

Installers often need to look up the value of a registry entry during the installation process. The resulting registry value is often used in a conditional statement later in install, such as to install a specific component if a registry entry is not found. This how to demonstrates reading an integer value from the registry and verifying that it exists in a launch condition.

Step 1: Read the registry entry into a property

Registry entries are read using the RegistrySearch element. The following snippet looks for the the presence of the key that identifies the installation of .NET Framework 2.0 on the target machine*.

```
<Property Id="NETFRAMEWORK20">
    <RegistrySearch Id="NetFramework20"
                    Root="HKLM"
                    Key="Software\Microsoft\NET Framework Setup\NDP\v2.0.50727"
                    Name="Install"
                    Type="raw" />
```

```
</Property>
```

The RegistrySearch element specifies a unique id, the root in the registry to search, and the key to look under. The name attribute specifies the specific value to query. The type attribute specifies how the value should be treated. Raw indicates that the value should be prefixed according to the data type of the value. In this case, since Install is a DWORD, the resulting value will be prepended with a #.

The above sample will set the NETFRAMEWORK20 property to "#1" if the registry key was found, and to nothing if it wasn't.

Step 2: Use the property in a condition

After the property is set you can use it in a condition anywhere in your WiX project. The following snippet demonstrates how to use it to block installation if .NET Framework 2.0 is not installed.

```
<Condition Message="This application requires .NET Framework 2.0. Please install the .NET Framework then
run this installer again.">
    <![CDATA[Installed OR NETFRAMEWORK20]]>
</Condition>
```

Installed is a Windows Installer property that ensures the check is only done when the user is installing the application, rather than on a repair or remove. The NETFRAMEWORK20 part of the condition will pass if the property was set. If it is not set the installer will display the error message then abort the installation process.
* This registry entry is used for sample purposes only. If you want to detect the installed version of .NET Framework you can use the built-in WiX support. For more information see How To: Check for .NET Framework Versions.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Create a Shortcut on the Start Menu

When installing applications it is a common requirement to place a shortcut on the user's Start Menu to provide a launching point for the program. This how to walks through how to create a shortcut on the start menu. It assumes you have a WiX source file based on the concepts described in How To: Add a file to your installer.

Step 1: Define the directory structure

Start Menu shortcuts are installed in a different directory than regular application files, so modifications to the installer's directory structure are required. The following WiX fragment should be placed inside a Directory element with the TARGETDIR ID and adds directory structure information for the Start Menu:

```
<Directory Id="ProgramMenuFolder">
    <Directory Id="ApplicationProgramsFolder" Name="My Application Name"/>
</Directory>
```

The ProgramMenuFolder Id is a standard Windows Installer property that points to the Start Menu folder on the target machine. The second Directory element creates a subfolder on the Start Menu called My Application Name, and gives it an id for use later in the WiX project.

Step 2: Add the shortcut to your installer package

A shortcut is added to the installer using three elements: a Component element to specify an atomic unit of installation, a Shortcut element to specify the shortcut that should be installed, and a RemoveFolder element to ensure proper cleanup when your application is uninstalled.

The following sample uses the directory structure defined in Step 1 to create the Start Menu shortcut.

```
<DirectoryRef Id="ApplicationProgramsFolder">
    <Component Id="ApplicationShortcut" Guid="PUT-GUID-HERE">
        <Shortcut Id="ApplicationStartMenuShortcut"
                  Name="My Application Name"
                  Description="My Application Description"
                  Target="[#myapplication.exe]"
                  WorkingDirectory="APPLICATIONROOTDIRECTORY"/>
        <RemoveFolder Id="CleanUpShortCut" Directory="ApplicationProgramsFolder" On="uninstall"/>
        <RegistryValue Root="HKCU" Key="Software\MyCompany\MyApplicationName" Name="installed"
Type="integer" Value="1" KeyPath="yes"/>
    </Component>
</DirectoryRef>
```

The DirectoryRef element is used to refer to the directory structure created in step 1. By referencing the ApplicationProgramsFolder directory the shortcut will be installed into the user's Start Menu inside the My Application Name folder.

Underneath the DirectoryRef is a single Component to group the elements used to install the Shortcut. The first element is Shortcut and it creates the actual shortcut in the Start Menu. The Id attribute is a unique id for the shortcut. The Name attribute is the text that will be displayed in the Start Menu. The description is an optional attribute for an additional application description. The Target attribute points to the executable to launch on disk. Notice how it references the full path using the `[#FileId]` syntax where `myapplication.exe` was previously defined. The WorkingDirectory attribute sets the working directory for the shortcut.

To set an optional icon for the shortcut you need to first include the icon in your installer using the Icon element, then reference it using the Icon attribute on the Shortcut element.

In addition to creating the shortcut the component contains two other important pieces. The first is a RemoveFolder element, which ensures the ApplicationProgramsFolder is correctly removed from the Start Menu when the user uninstalls the application. The second creates a registry entry on install that indicates the application is installed. This is required as a Shortcut cannot serve as the KeyPath for a component when installing non-advertised shortcuts for the current users. For more information on creating registry entries see How To: Write a registry entry during installation.

Step 3: Tell Windows Installer to install the shortcut

After defining the directory structure and listing the shortcuts to package into the installer, the last step is to tell Windows Installer to actually install the shortcut. The Feature element is used to do this. The following snippet adds a reference to the shortcut component, and should be inserted inside a parent Feature element.

```
<ComponentRef Id="ApplicationShortcut" />
```

The ComponentRef element is used to reference the component created in Step 2 via the Id attribute.

The Complete Sample

The following is a complete sample that uses the above concepts. This example can be inserted into a WiX project and compiled, or compiled and linked from the command line, to generate an installer.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <Product Id="*" UpgradeCode="PUT-GUID-HERE" Version="1.0.0.0" Language="1033" Name="My Application
Name" Manufacturer="My Manufacturer Name">
        <Package InstallerVersion="300" Compressed="yes"/>
        <Media Id="1" Cabinet="myapplication.cab" EmbedCab="yes" />

        <Directory Id="TARGETDIR" Name="SourceDir">
            <Directory Id="ProgramFilesFolder">
                <Directory Id="APPLICATIONROOTDIRECTORY" Name="My Application Name"/>
            </Directory>
            <!-- Step 1: Define the directory structure -->
            <Directory Id="ProgramMenuFolder">
                <Directory Id="ApplicationProgramsFolder" Name="My Application Name"/>
            </Directory>
        </Directory>

        <DirectoryRef Id="APPLICATIONROOTDIRECTORY">
            <Component Id="myapplication.exe" Guid="PUT-GUID-HERE">
                <File Id="myapplication.exe" Source="MySourceFiles\MyApplication.exe" KeyPath="yes"
Checksum="yes"/>
            </Component>
            <Component Id="documentation.html" Guid="PUT-GUID-HERE">
                <File Id="documentation.html" Source="MySourceFiles\documentation.html" KeyPath="yes"/>
            </Component>
```

```
        </DirectoryRef>

        <!-- Step 2: Add the shortcut to your installer package -->
        <DirectoryRef Id="ApplicationProgramsFolder">
            <Component Id="ApplicationShortcut" Guid="PUT-GUID-HERE">
                <Shortcut Id="ApplicationStartMenuShortcut"
                          Name="My Application Name"
                          Description="My Application Description"
                          Target="[#myapplication.exe]"
                          WorkingDirectory="APPLICATIONROOTDIRECTORY"/>
                <RemoveFolder Id="ApplicationProgramsFolder" On="uninstall"/>
                <RegistryValue Root="HKCU" Key="Software\MyCompany\MyApplicationName" Name="installed"
Type="integer" Value="1" KeyPath="yes"/>
            </Component>
        </DirectoryRef>

        <Feature Id="MainApplication" Title="Main Application" Level="1">
            <ComponentRef Id="myapplication.exe" />
            <ComponentRef Id="documentation.html" />
            <!-- Step 3: Tell WiX to install the shortcut -->
            <ComponentRef Id="ApplicationShortcut" />
        </Feature>
    </Product>
</Wix>
```

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Create a Shortcut to a Webpage

WiX provides support for creating shortcuts to Internet sites as part of the install process. This how to demonstrates referencing the necessary utility library and adding an Internet shortcut to your installer. It assumes you have already followed the steps in the How To: Create a shortcut on the Start Menu.

Step 1: Add the WiX Utility extensions library to your project

The WiX support for Internet shortcuts is included in a WiX extension library that must be added to your project prior to use. If you are using WiX on the command-line you need to add the following to your candle and light command lines:

```
-ext WiXUtilExtension
```

If you are using WiX in Visual Studio you can add the extensions using the Add Reference dialog:

1. Open your WiX project in Visual Studio
2. Right click on your project in Solution Explorer and select Add Reference...

3. Select the **WixUtilExtension.dll** assembly from the list and click Add
4. Close the Add Reference dialog

Step 2: Add the WiX Utility extensions namespace to your project

Once the library is added to your project, you need to add the Utility extensions namespace to your project so you can access the appropriate WiX elements. To do this modify the top-level Wix element in your project by adding the following attribute:

```
xmlns:util="http://schemas.microsoft.com/wix/UtilExtension"
```

A complete Wix element with the standard namespace and the Utility extensions namespace added looks like this:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
     xmlns:util="http://schemas.microsoft.com/wix/UtilExtension">
```

Step 3: Add the Internet shortcut to your installer package

Internet shortcuts are created using the Util:InternetShortcut element. The following example adds an InternetShortcut element to the existing shortcut creation example from How To: Create a shortcut on the Start Menu.

```
<DirectoryRef Id="ApplicationProgramsFolder">
  <Component Id="ApplicationShortcut" Guid="PUT-GUID-HERE">
```

```
    <Shortcut Id="ApplicationStartMenuShortcut"
              Name="My Application Name"
              Description="My Application Description"
              Target="[#MyApplicationExeFileId]"
              WorkingDirectory="APPLICATIONROOTDIRECTORY"/>
    <util:InternetShortcut Id="OnlineDocumentationShortcut"
                           Name="My Online Documentation"
                           Target="http://wixtoolset.org/"/>
    <RemoveFolder Id="ApplicationProgramsFolder" On="uninstall"/>
    <RegistryValue Root="HKCU" Key="Software\MyCompany\MyApplicationName" Name="installed" Type="integer"
Value="1" KeyPath="yes"/>
  </Component>
</DirectoryRef>
```

The InternetShortcut is given a unique id with the Id attribute. in this case the application's Start Menu folder. The Name attribute specifies the name of the shortcut on the Start Menu. The Target attribute specifies the destination address for the shortcut. The DirectoryRef element is used to refer to the directory structure already defined by the project file. By referencing the ApplicationProgramsFolder directory the shortcut will be installed into the user's Start Menu inside the My Application Name folder.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Create an Uninstall Shortcut

When installing an application it is a common requirement to place a shortcut on the user's Start Menu to provide a method of uninstalling the application. This how to demonstrates the steps required to create an uninstall shortcut on the start menu that passes all ICE validation checks.

This how to assumes you are starting with the sample described the How To: Create a Shortcut on the Start Menu topic.

Step 1: Add the Uninstall Shortcut

The Shortcut element is used to add the uninstall shortcut to the start menu, and the shortcut points to msiexec.exe (the Windows Installer executable used to actually invoke the uninstall process). Anywhere within the existing ApplicationShortcut component add the following:

```
<Shortcut Id="UninstallProduct"
          Name="Uninstall My Application"
          Target="[SystemFolder]msiexec.exe"
          Arguments="/x [ProductCode]"
          Description="Uninstalls My Application" />
```

The Target attribute points to the location of msiexec.exe. The Windows Installer SystemFolder property will resolve to the *System32* directory where msiexec.exe resides. The Arguments attribute is used to let msiexec.exe know which product to uninstall by passing in the ProductCode for the install package.

To avoid ICE validation errors at build it is important to couple the Shortcut element with a registry entry and a RemoteFolder element. Both of these are described in more detail in the How To: Create a Shortcut on the Start Menu topic, and are shown in the complete sample below.

The Complete Sample

The following is a complete sample that uses the above concepts. This example can be inserted into a WiX project and compiled, or compiled and linked from the command line, to generate an installer.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <Product Id="*" UpgradeCode="PUT-GUID-HERE" Version="1.0.0.0" Language="1033" Name="My Application
Name" Manufacturer="My Manufacturer Name">
        <Package InstallerVersion="300" Compressed="yes"/>
        <Media Id="1" Cabinet="myapplication.cab" EmbedCab="yes" />

        <Directory Id="TARGETDIR" Name="SourceDir">
            <Directory Id="ProgramFilesFolder">
                <Directory Id="APPLICATIONROOTDIRECTORY" Name="My Application Name"/>
            </Directory>
            <Directory Id="ProgramMenuFolder">
                <Directory Id="ApplicationProgramsFolder" Name="My Application Name"/>
            </Directory>
        </Directory>
```

```
        <DirectoryRef Id="APPLICATIONROOTDIRECTORY">
            <Component Id="myapplication.exe" Guid="PUT-GUID-HERE">
                <File Id="myapplication.exe" Source="MySourceFiles\MyApplication.exe" KeyPath="yes"
Checksum="yes"/>
            </Component>
            <Component Id="documentation.html" Guid="PUT-GUID-HERE">
                <File Id="documentation.html" Source="MySourceFiles\documentation.html" KeyPath="yes"/>
            </Component>
        </DirectoryRef>

        <DirectoryRef Id="ApplicationProgramsFolder">
            <Component Id="ApplicationShortcut" Guid="PUT-GUID-HERE">
                <Shortcut Id="ApplicationStartMenuShortcut"
                          Name="My Application Name"
                          Description="My Application Description"
                          Target="[#myapplication.exe]"
                          WorkingDirectory="APPLICATIONROOTDIRECTORY"/>
                <!-- Step 1: Add the uninstall shortcut to your installer package -->
                <Shortcut Id="UninstallProduct"
                          Name="Uninstall My Application"
                          Description="Uninstalls My Application"
                          Target="[System64Folder]msiexec.exe"
                          Arguments="/x [ProductCode]"/>
                <RemoveFolder Id="ApplicationProgramsFolder" On="uninstall"/>
                <RegistryValue Root="HKCU" Key="Software\MyCompany\MyApplicationName" Name="installed"
Type="integer" Value="1" KeyPath="yes"/>
            </Component>
        </DirectoryRef>

        <Feature Id="MainApplication" Title="Main Application" Level="1">
            <ComponentRef Id="myapplication.exe" />
            <ComponentRef Id="documentation.html" />
```

```
            <ComponentRef Id="ApplicationShortcut" />
        </Feature>
    </Product>
</Wix>
```

Edit this page

Version: v3

How To: NGen Managed Assemblies During Installation

NGen during installation can improve your managed application's startup time by creating native images of the managed assemblies on the target machine. This how to describes using the WiX support to NGen managed assemblies at install time.

Step 1: Add the WiX .NET extensions library to your project

The WiX support for NGen is included in a WiX extension library that must be added to your project prior to use. If you are using WiX on the command-line you need to add the following to your candle and light command lines:

```
-ext WixNetFxExtension
```

If you are using WiX in Visual Studio you can add the extensions using the Add Reference dialog:

1. Open your WiX project in Visual Studio
2. Right click on your project in Solution Explorer and select Add Reference...

3. Select the **WixNetFxExtension.dll** assembly from the list and click Add
4. Close the Add Reference dialog

Step 2: Add the WiX .NET extensions namespace to your project

Once the library is added to your project you need to add the .NET extensions namespace to your project so you can access the appropriate WiX elements. To do this modify the top-level Wix element in your project by adding the following attribute:

```
xmlns:netfx="http://schemas.microsoft.com/wix/NetFxExtension"
```

A complete Wix element with the standard namespace and the .NET extensions namespace added looks like this:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
     xmlns:netfx="http://schemas.microsoft.com/wix/NetFxExtension">
```

Step 3: Mark the managed files for NGen

Once you have the .NET extension library and namespace added to your project you can use the NetFx:NativeImage element to enable NGen on your managed assemblies. The NativeImage element goes inside a parent File element:

```
<Component Id="myapplication.exe" Guid="PUT-GUID-HERE">
    <File Id="myapplication.exe" Source="MySourceFiles\MyApplication.exe" KeyPath="yes" Checksum="yes">
```

```
        <netfx:NativeImage Id="ngen_MyApplication.exe" Platform="32bit" Priority="0"
AppBaseDirectory="APPLICATIONROOTDIRECTORY"/>
    </File>
</Component>
```

The Id attribute is a unique identifier for the native image. The Platform attribute specifies the platforms for which the native image should be generated, in this case 32-bit. The Priority attribute specifies when the image generation should occur, in this case immediately during the setup process. The AppBaseDirectory attribute identifies the directory to use to search for dependent assemblies during the image generation. In this case it is set to the install directory for the application.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Reference another DirectorySearch element

There may be times when you need to locate different files or subdirectories under the same directory, and assign each to a separate property. Since you cannot define the same DirectorySearch element more than once, you must use a DirectorySearchRef element. To reference another DirectorySearch element, you must specify the same Id, Parent Id, and Path attribute values or you will get unresolved symbol errors when linking with light.exe.

Step 1: Define a DirectorySearch element

You first need to define the parent DirectorySearch element. This is expected to contain the different files or subdirectories you will assign to separate properties.

```
<Property Id="SHDOCVW">
    <DirectorySearch Id="WinDir" Path="[WindowsFolder]">
        <DirectorySearch Id="Media" Path="Media">
            <FileSearch Id="Chimes" Name="chimes.wav" />
        </DirectorySearch>
```

```
    </DirectorySearch>
</Property>
```

This will search for the file "chimes.wav" under the Media directory in Windows. If the file is found, the full path will be assigned to the public property "SHDOCVW".

Step 2: Define a DirectorySearchRef element

To search for another file in the Media directory, you need to reference all the same Id, Parent Id, and Path attributes. Because the Media DirectorySearch element is nested under the WinDir DirectorySearch element, its Parent attribute is automatically assigned the parent DirectorySearch elements Id attribute value; thus, that is what you must specify for the DirectorySearchRef element's Parent attribute value.

```
<Property Id="USER32">
    <DirectorySearchRef Id="Media" Parent="WinDir" Path="Media">
        <FileSearch Id="Chord" Name="chord.wav" />
    </DirectorySearchRef>
</Property>
```

If you wanted to refer to another DirectorySearch element that used the Id Media but was under a different parent path, you would have to define a new DirectorySearch element under a different parent than in step 1.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Get the parent directory of a file search

You can set a property to the parent directory of a file.

Step 1: Define the search root

In the following example, the path to [WindowsFolder]Microsoft.NET is defined as the root of the search. If you do not define a search root, Windows Installer will search all fixed drives up to the depth specified.

```
<Property Id="NGEN2DIR">
    <DirectorySearch Id="Windows" Path="[WindowsFolder]">
        <DirectorySearch Id="MS.NET" Path="Microsoft.NET">
        </DirectorySearch>
    </DirectorySearch>
</Property>
```

Step 2: Define the parent directory to find

Under the search root, define the directory you want returned and set the DirectorySearch/@AssignToProperty attribute to 'yes'. You must then define the file you want to find using a unique FileSearch/@Id attribute value.

```
<Property Id="NGEN2DIR">
    <DirectorySearch Id="Windows" Path="[WindowsFolder]">
        <DirectorySearch Id="MS.NET" Path="Microsoft.NET">
            <DirectorySearch Id="Ngen2Dir" Depth="2" AssignToProperty="yes">
                <FileSearch Id="Ngen_exe" Name="ngen.exe" MinVersion="2.0.0.0" />
            </DirectorySearch>
        </DirectorySearch>
    </DirectorySearch>
</Property>
```

In this example, if ngen.exe is newer than version 2.0.0.0 and is found no more than two directories under [WindowsFolder]Microsoft.NET its parent directory is returned in the NGEN2DIR property.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Check for .NET Framework Versions

When installing applications written using managed code, it is often useful to verify that the user's machine has the necessary version of the .NET Framework prior to installation. The WiX support for detecting .NET Framework versions is included in a WiX extension, WixNetFxExtension. This how to describes using the WixNetFxExtension to verify .NET Framework versions at install time. For information on how to install the .NET Framework during your installation see How To: Install the .NET Framework Using Burn.

Step 1: Add WixNetFxExtension to your project

You must add the WixNetFxExtension to your project prior to use. If you are using WiX on the command line, you need to add the following to your candle and light command lines:

-ext WixNetFxExtension

If you are using WiX in Visual Studio, you can add the extension using the Add Reference dialog:

1. Open your WiX project in Visual Studio.

2. Right click on your project in Solution Explorer and select **Add Reference...**.
3. Select the **WixNetFxExtension.dll** assembly from the list and click Add.
4. Close the Add Reference dialog.

Step 2: Add WixNetFxExtension's namespace to your project

Once the extension is added to your project, you need to add its namespace to your project so you can access the appropriate WiX elements. To do this, modify the top-level Wix element in your project by adding the following attribute:

```
xmlns:netfx="http://schemas.microsoft.com/wix/NetFxExtension"
```

A complete Wix element with the standard namespace and WixNetFxExtension's namespace added looks like this:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
     xmlns:netfx="http://schemas.microsoft.com/wix/NetFxExtension">
```

Step 3: Reference the required properties in your project

WixNetFxExtension defines properties for all current versions of the .NET Framework, including service pack levels. To make these properties available to your installer, you need to reference them using the PropertyRef element. For each property you want to use, add the corresponding PropertyRef to your project. For example, if you are interested in detecting .NET Framework 2.0 add the following:

```
<PropertyRef Id="NETFRAMEWORK20"/>
```

Step 4: Use the pre-defined properties in a condition

Once the property is referenced, you can use it in any WiX condition statement. For example, the following condition blocks installation if .NET Framework 2.0 is not installed.

```
<Condition Message="This application requires .NET Framework 2.0. Please install the .NET Framework then
run this installer again.">
  <![CDATA[Installed OR NETFRAMEWORK20]]>
</Condition>
```

Installed is a Windows Installer property that ensures the check is only done when the user is installing the application, rather than on a repair or remove. The NETFRAMEWORK20 part of the condition will pass if .NET Framework 2.0 is installed. If it is not set, the installer will display the error message then abort the installation process.
To check against the service pack level of the framework, use the *_SP_LEVEL properties. The following condition blocks installation if .NET Framework 3.0 SP1 is not present on the machine.

```
<Condition Message="This application requires .NET Framework 3.0 SP1. Please install the .NET Framework
then run this installer again.">
    <![CDATA[Installed OR (NETFRAMEWORK30_SP_LEVEL and NOT NETFRAMEWORK30_SP_LEVEL = "#0")]]>
</Condition>
```

As with the previous example, Installed prevents the check from running when the user is doing a repair or remove. The NETFRAMEWORK30_SP_LEVEL property is set to "#1" if Service Pack 1 is present. Since there is no way to do a numerical

comparison against a value with a # in front of it, the condition first checks to see if the NETFRAMEWORK30_SP_LEVEL is set and then confirms that it is set to a number. This will correctly indicate whether any service pack for .NET 3.0 is installed.

Edit this page

Version: v3

How To: Install the .NET Framework Using Burn

Applications written using the .NET Framework often need to bundle the .NET framework and install it with their application. Wix 3.6 and later makes this easy with Burn.

Step 1: Create a bundle for your application

Follow the instructions in Building Installation Package Bundles.

Step 2: Add a reference to one of the .NET PackageGroups

1. Add a reference to WixNetFxExtension to your bundle project.
2. Add a PackageGroupRef element to your bundle's chain that references the .NET package required by your application. For a full list, see WixNetfxExtension. Ensure that the PayloadGroupRef is placed before any other packages that require .NET.

```
<Chain>
    <PackageGroupRef Id="NetFx45Web"/>
    <MsiPackage Id="MyApplication" SourceFile="$(var.MyApplicationSetup.TargetPath)"/>
</Chain>
```

that does not require Internet connectivity, you can package the .NET redistributable with your bundle. Doing so requires you have a local copy of the redistributable, such as checked in to your source-control system.

```
<Bundle>
    <PayloadGroup Id="NetFx452RedistPayload">
        <Payload Name="redist\NDP452-KB2901907-x86-x64-AllOS-ENU.exe"
                 SourceFile="X:\path\to\redists\in\repo\NDP452-KB2901907-x86-x64-AllOS-ENU.exe"/>
    </PayloadGroup>
</Bundle>
```

Note that the PackageGroupRef in the bundle's chain is still required.

Customizing your bootstrapper application

Any native bootstrapper application, including the WiX Standard Bootstrapper Application, will work well with bundles that include .NET.

Managed bootstrapper applications must take care when including .NET to ensure that they do not unnecessarily depend on the .NET framework version being installed.

1. Reference the managed bootstrapper application host from your bundle.

```
<BootstrapperApplicationRef Id="ManagedBootstrapperApplicationHost">
  <Payload Name="BootstrapperCore.config"
           SourceFile="$(var.MyMBA.TargetDir)\TestUX.BootstrapperCore.config"/>
  <Payload SourceFile="$(var.MyMBA.TargetPath)"/>
```

```
<configuration>
  <configSections>
    <sectionGroup name="wix.bootstrapper"
type="Microsoft.Tools.WindowsInstallerXml.Bootstrapper.BootstrapperSectionGroup, BootstrapperCore">
      <section name="host" type="Microsoft.Tools.WindowsInstallerXml.Bootstrapper.HostSection,
BootstrapperCore" />
    </sectionGroup>
  </configSections>
  <startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v2.0.50727" />
    <supportedRuntime version="v4.0" />
  </startup>
  <wix.bootstrapper>
    <host assemblyName="MyBootstrapperApplicationAssembly">
      <supportedFramework version="v3.5" />
      <supportedFramework version="v4\Client" />
        <!-- Example only. Replace the host/@assemblyName attribute with
        an assembly that implements BootstrapperApplication. -->
        <host assemblyName="$(var.MyMBA.TargetPath)" />
    </host>
  </wix.bootstrapper>
</configuration>
```

Edit this page

Version: v3

How To: Install DirectX 9.0 With Your Installer

Applications that require components from DirectX 9.0 can benefit from including the DirectX 9.0 Redistributable inside their installer. This simplifies the installation process for end users and ensures the required components for your application are always available on the target user's machine.

DirectX 9.0 can be re-distributed in several different ways, each of which is outlined in MSDN's Installing DirectX with DirectSetup article. This how to describes using the dxsetup.exe application to install DirectX 9.0 on a Vista machine assuming the application being installed only depends on a specific DirectX component.

Prior to redistributing the DirectX binaries you should read and understand the license agreement for the redistributable files. The license agreement can be found in the **Documentation\License Agreements\DirectX Redist.txt** file in your DirectX SDK installation.

Step 1: Add the installer files to your WiX project

Adding the files to the WiX project follows the same process as described in How To: Add a file to your installer. The following

example illustrates a typical fragment that includes the necessary files:

```
<DirectoryRef Id="APPLICATIONROOTDIRECTORY">
    <Directory Id="DirectXRedistDirectory" Name="DirectX9.0c">
        <Component Id="DirectXRedist" Guid="PUT-GUID-HERE">
            <File Id="DXSETUPEXE"
                  Source="MySourceFiles\DirectXMinInstall\dxsetup.exe"
                  KeyPath="yes"
               Checksum="yes"/>
            <File Id="dxupdate.cab"
                  Source="MySourceFiles\DirectXMinInstall\dxupdate.cab"/>
            <File Id="dxdllreg_x86.cab"
                  Source="MySourceFiles\DirectXMinInstall\dxdllreg_x86.cab"/>
            <File Id="dsetup32.dll"
                  Source="MySourceFiles\DirectXMinInstall\dsetup32.dll"/>
            <File Id="dsetup.dll"
                  Source="MySourceFiles\DirectXMinInstall\dsetup.dll"/>
            <File Id="DEC2006_d3dx9_32_x86.cab"
                  Source="MySourceFiles\DirectXMinInstall\DEC2006_d3dx9_32_x86.cab"/>
        </Component>
    </Directory>
</DirectoryRef>

<Feature Id="DirectXRedist"
         Title="!(loc.FeatureDirectX)"
         AllowAdvertise="no"
         Display="hidden" Level="1">
    <ComponentRef Id="DirectXRedist"/>
</Feature>
```

The files included are the minimal set of files required by the DirectX 9.0 install process, as described in the MSDN documentation.

The last file in the list, DEC2006_d3dx9_32_x86.cab contains the specific DirectX component required by the installed application. These files are all included in a single component as, even in a patching situation, all the files must go together. A Feature element is used to create a feature specific to DirectX installation, and its Display attribute is set to **hidden** to prevent the user from seeing the feature in any UI that may be part of your installer.

Step 2: Add a custom action to invoke the installer

To run the DirectX 9.0 installer a custom action is added that runs before the install is finalized. The CustomAction, InstallExecuteSequence and Custom elements are used to create the custom action, as illustrated in the following sample.

```
<CustomAction Id="InstallDirectX"
              FileKey="DXSETUPEXE"
              ExeCommand="/silent"
              Execute="deferred"
              Impersonate="no"
              Return="check"/>

<InstallExecuteSequence>
  <Custom Action="InstallDirectX" Before="InstallFinalize">
    <![CDATA[NOT REMOVE]]>
  </Custom>
</InstallExecuteSequence>
```

The CustomAction element creates the custom action that runs the setup. It is given a unique id, and the FileKey attribute is used to reference the installer application from Step 1. The ExeCommand attribute adds the **/silent** flag to the installer to ensure the user is not presented with any DirectX installer user interface. The Execute attribute is set to deferred and the Impersonate attribute is set to no to ensure the custom action will run elevated, if necessary. The Return attribute is set to check to ensure the custom action

runs synchronously.

The Custom element is used inside an InstallExecuteSequence to add the custom action to the actual installation process. The Action attribute references the CustomAction by its unique id. The Before attribute is set to InstallFinalize to run the custom action before the overall installation is complete. The condition prevents the DirectX installer from running when the user uninstalls your application, since DirectX components cannot be uninstalled.

Step 3: Include progress text for the custom action

If you are using standard WiX UI dialogs you can include custom progress text for display while the DirectX installation takes place. The UI and ProgressText elements are used, as illustrated in the following example.

```
<UI>
    <ProgressText Action="InstallDirectX">Installing DirectX 9.0c</ProgressText>
</UI>
```

The ProgressText element uses the Action attribute to reference the custom action by its unique id. The value of the ProgressText element is set to the display text for the install progress.

Edit this page

Version: v3

How To: Install the Visual C++ Redistributable with your installer

If your application depends on the Visual C++ runtimes you can include them as part of your installer to simplify the installation experience for your end users. This how to describes including the Visual C++ runtime merge modules into your installer and explains the expected ICE warnings you will see.

Step 1: Obtain the correct Visual C++ runtime merge modules

The Visual C++ runtime merge modules are installed with Visual Studio and are located in **\Program Files\Common Files\Merge Modules**. The Visual C++ 8.0 runtime file is **Microsoft_VC80_CRT_x86.msm**. This same MSM is used for the Visual C++ 8.0 SP1 runtime, however it is updated in place by the Visual Studio 2005 SP1 installer. The Visual Studio 9.0 runtime file is **Microsoft_VC90_CRT_x86.msm**. There is generally no need to include the policy MSMs as part of the installation.

Step 2: Include the merge module in your installer

To include the merge module in your installer use the Merge and MergeRef elements. The following example illustrates how these elements are used.

```
<DirectoryRef Id="TARGETDIR">
  <Merge Id="VCRedist" SourceFile="MySourceFiles\Microsoft_VC80_CRT_x86.msm" DiskId="1" Language="0"/>
</DirectoryRef>
```

```
<Feature Id="VCRedist" Title="Visual C++ 8.0 Runtime" AllowAdvertise="no" Display="hidden" Level="1">
  <MergeRef Id="VCRedist"/>
</Feature>
```

The Merge element ensures the merge module is included in the final Windows Installer package. A unique id is assigned using the Id attribute. The SourceFile attribute points to the location of the merge module on your machine. The DiskId attribute should match the DiskId specified in your project's Media element. The Language attribute should always be 0.

The MergeRef element is used within a Feature element to actually install the merge module. In the example above a feature specific to the runtime is created and marked as hidden to prevent it from displaying in any UI your installer may use. The MergeRef refers to the merge module by its unique id.

A note about ICE warnings

Including the Visual C++ Runtime merge module in your installer will result in the following ICE warnings:

```
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Component, Column: KeyPath, Key(s): downlevel_manifest.8.0.50727.762.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
```

```
Component, Column: KeyPath, Key(s): downlevel_manifest.8.0.50727.100.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Component, Column: KeyPath, Key(s): downlevel_manifest.8.0.50727.101.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Component, Column: KeyPath, Key(s): downlevel_manifest.8.0.50727.103.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Component, Column: KeyPath, Key(s): downlevel_manifest.8.0.50727.104.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Component, Column: KeyPath, Key(s): downlevel_manifest.8.0.50727.193.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Registry, Column: Registry, Key(s):
reg_downlevel_manifest.8.0.50727.100.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Registry, Column: Registry, Key(s):
reg_downlevel_manifest.8.0.50727.101.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Registry, Column: Registry, Key(s):
reg_downlevel_manifest.8.0.50727.103.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Registry, Column: Registry, Key(s):
reg_downlevel_manifest.8.0.50727.104.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Registry, Column: Registry, Key(s):
reg_downlevel_manifest.8.0.50727.193.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE03: String overflow (greater than length permitted in column); Table:
Registry, Column: Registry, Key(s):
reg_downlevel_manifest.8.0.50727.762.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
light.exe(0,0): warning LGHT1076: ICE25: Possible dependency failure as we do not find
CRT.Policy.63E949F6_03BC_5C40_FF1F_C8B3B9A1E18E@0 v in ModuleSignature table
light.exe(0,0): warning LGHT1076: ICE82: This action SystemFolder.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E has
duplicate sequence number 1 in the table InstallExecuteSequence
light.exe(0,0): warning LGHT1076: ICE82: This action SystemFolder.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E has
```

```
duplicate sequence number 1 in the table InstallUISequence
light.exe(0,0): warning LGHT1076: ICE82: This action SystemFolder.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E has
duplicate sequence number 1 in the table AdminExecuteSequence
light.exe(0,0): warning LGHT1076: ICE82: This action SystemFolder.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E has
duplicate sequence number 1 in the table AdminUISequence
light.exe(0,0): warning LGHT1076: ICE82: This action SystemFolder.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E has
duplicate sequence number 1 in the table AdvtExecuteSequence
```

These warnings are expected and are due to how the Visual C++ merge modules were authored.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Block Installation Based on OS Version

Windows Installer provides the standard VersionNT property that can be used to detect the version of the user's operating system. Often it is desirable to use this property to block installation of an application on incompatible versions of an operating system. The following sample demonstrates how to use this property to block installation of an application on operating systems prior to Windows Vista/Windows Server 2008.

```
<Condition Message="This application is only supported on Windows Vista, Windows Server 2008, or higher.">
    <![CDATA[Installed OR (VersionNT >= 600)]]>
</Condition>
```

Installed is a Windows Installer property that ensures the check is only done when the user is installing the application, rather than on a repair or remove. The VersionNT part will pass if the property's value is greater than or equal to 600, the version that matches Windows Vista, the installation will proceed. The values for different versions of the Windows operating system are available on MSDN.
To check for versions of 64-bit Windows use the VersionNT64 property. To check for versions of Windows prior to Windows NT use the Windows9X property.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Build a Localized Version of Your Installer

Once you have described all the strings in your installer using language files, as described in How To: Make your installer localizable, you can then build versions of your installer for each supported language. This how to explains building the localized installers both from the command line and using Visual Studio.

Option 1: Building localized installers from the command line

The first step in building a localized installer is to compile your WiX sources using candle.exe:

```
candle.exe myinstaller.wxs -out myinstaller.wixobj
```

After the intermediate output file is generated you can then use light.exe to generate multiple localized MSIs:

```
light.exe myinstaller.wixobj -cultures:en-us -loc en-us.wxl -out myinstaller-en-us.msi
light.exe myinstaller.wixobj -cultures:fr-fr -loc fr-fr.wxl -out myinstaller-fr-fr.msi
```

Option 2: Building localized installers using Visual Studio

Visual Studio will automatically build localized versions of your installer. If your WiX project includes multiple .wxl files, one localized installer will be built for each culture, unless **Cultures to build** is specified.

For more information, see Specifying cultures to build

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Make your installer localizable

WiX supports building localized installers through the use of language files that include localized strings. It is a good practice to put all your strings in a language file as you create your setup, even if you do not currently plan on shipping localized versions of your installer. This how to describes how to create a language file and use its strings in your WiX project.

Step 1: Create the language file

Language files end in the .wxl extension and specify their culture using the WixLocalization element. To create a language file on the command line create a new file with the appropriate name and add the following:

```
<?xml version="1.0" encoding="utf-8"?>
<WixLocalization Culture="en-us" xmlns="http://schemas.microsoft.com/wix/2006/localization">
</WixLocalization>
```

If you are using Visual Studio you can add a new language file to your project by doing the following:

1. Right click on your project in Solution Explorer and select Add > New Item...
2. Select WiX Localization File, give the file an appropriate name, and select Add

By default Visual Studio creates language files in the en-us culture. To create a language file for a different culture change the Culture attribute to the appropriate culture string.

Step 2: Add the localized strings

Localized strings are defined using the String element. Each element consists of a unique id for later reference in your WiX project and the string value. For example:

```
<String Id="ApplicationName">My Application Name</String>
<String Id="ManufacturerName">My Manufacturer Name</String>
```

The String element goes inside the WixLocalization element, and you should add one String element for each piece of text you need to localize.

Step 3: Use the localized strings in your project

Once you have defined the strings you can use them in your project wherever you would normally use text. For example, to set your product's Name and Manufacturer to the localized strings do the following:

```
<Product Id="*"
         UpgradeCode="PUT-GUID-HERE"
         Version="1.0.0.0"
         Language="1033"
         Name="!(loc.ApplicationName)"
         Manufacturer="!(loc.ManufacturerName)">
```

Localization strings are referenced using the **!(loc.stringname)** syntax. These references will be replaced with the actual strings for the appropriate locale at build time.

For information on how to compile localized versions of your installer once you have the necessary language files see How To: Build a localized version of your installer.

Edit this page

Version: v3

How To: Run the Installed Application After Setup

Often when completing the installation of an application it is desirable to offer the user the option of immediately launching the installed program when setup is complete. This how to describes customizing the default WiX UI experience to include a checkbox and a WiX custom action to launch the application if the checkbox is checked.

This how to assumes you have already created a basic WiX project using the steps outlined in How To: Add a file to your installer.

Step 1: Add the extension libraries to your project

This walkthrough requires WiX extensions for UI components and custom actions. These extension libraries must must be added to your project prior to use. If you are using WiX on the command-line you need to add the following to your candle and light command lines:

```
-ext WixUIExtension -ext WixUtilExtension
```

If you are using Visual Studio you can add the extensions using the Add Reference dialog:

1. Right click on your project in Solution Explorer and select Add Reference...
2. Select the **WixUIExtension.dll** assembly from the list and click Add
3. Select the **WixUtilExtension.dll** assembly from the list and click Add
4. Close the Add Reference dialog

Step 2: Add UI to your installer

The WiX Minimal UI sequence includes a basic set of dialogs that includes a finished dialog with optional checkbox. To include the sequence in your project add the following snippet anywhere inside the `Product` element.

```
<UI>
  <UIRef Id="WixUI_Minimal" />
</UI>
```

To display the checkbox on the last screen of the installer include the following snippet anywhere inside the `Product` element:

```
<Property Id="WIXUI_EXITDIALOGOPTIONALCHECKBOXTEXT" Value="Launch My Application Name" />
```

The WIXUI_EXITDIALOGOPTIONALCHECKBOXTEXT property is provided by the standard UI sequence that, when set, displays the checkbox and uses the specified value as the checkbox label.

Step 3: Include the custom action

Custom actions are included in a WiX project using the CustomAction element. Running an application is accomplished with the WixShellExecTarget custom action. To tell Windows Installer about the custom action, and to set its properties, include the following in your project anywhere inside the `Product` element:

```
<Property Id="WixShellExecTarget" Value="[#myapplication.exe]" />
<CustomAction Id="LaunchApplication" BinaryKey="WixCA" DllEntry="WixShellExec" Impersonate="yes" />
```

The Property element sets the WixShellExecTarget to the location of the installed application. WixShellExecTarget is the property Id the WixShellExec custom action expects will be set to the location of the file to run. The Value property uses the special # character to tell WiX to look up the full installed path of the file with the id myapplication.exe.

The CustomAction element includes the action in the installer. It is given a unique Id, and the BinaryKey and DllEntry properties indicate the assembly and entry point for the custom action. The Impersonate property tells Windows Installer to run the custom action as the installing user.

Step 4: Trigger the custom action

Simply including the custom action, as in Step 3, isn't sufficient to cause it to run. Windows Installer must also be told when the custom action should be triggered. This is done by using the Publish element to add it to the actions run when the user clicks the Finished button on the final page of the UI dialogs. The Publish element should be included inside the `UI` element from Step 2, and looks like this:

The Dialog property specifies the dialog the Custom Action will be attached to, in this case the ExitDialog. The Control property specifies that the Finish button on the dialog triggers the custom action. The Event property indicates that a custom action should be run when the button is clicked, and the Value property specifies the custom action that was included in Step 3. The condition on

```
<Publish Dialog="ExitDialog"
         Control="Finish"
         Event="DoAction"
         Value="LaunchApplication">WIXUI_EXITDIALOGOPTIONALCHECKBOX = 1 and NOT Installed</Publish>
```

The Complete Sample

```
<?xml version="1.0" encoding="UTF-8"?>
<<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <Product Id="*"
             UpgradeCode="PUT-GUID-HERE"
             Version="1.0.0.0"
             Language="1033"
             Name="My Application Name"
             Manufacturer="My Manufacturer Name">
    <Package InstallerVersion="300" Compressed="yes"/>
    <Media Id="1" Cabinet="myapplication.cab" EmbedCab="yes" />

    <!-- The following three sections are from the How To: Add a File to Your Installer topic-->
    <Directory Id="TARGETDIR" Name="SourceDir">
        <Directory Id="ProgramFilesFolder">
            <Directory Id="APPLICATIONROOTDIRECTORY" Name="My Application Name"/>
        </Directory>
    </Directory>

    <DirectoryRef Id="APPLICATIONROOTDIRECTORY">
        <Component Id="myapplication.exe" Guid="PUT-GUID-HERE">
            <File Id="myapplication.exe" Source="MySourceFiles\MyApplication.exe" KeyPath="yes"
Checksum="yes"/>
        </Component>
        <Component Id="documentation.html" Guid="PUT-GUID-HERE">
            <File Id="documentation.html" Source="MySourceFiles\documentation.html" KeyPath="yes"/>
```

```
        </Component>
    </DirectoryRef>

    <Feature Id="MainApplication" Title="Main Application" Level="1">
        <ComponentRef Id="myapplication.exe" />
        <ComponentRef Id="documentation.html" />
    </Feature>

    <!-- Step 2: Add UI to your installer / Step 4: Trigger the custom action -->
    <UI>
        <UIRef Id="WixUI_Minimal" />
        <Publish Dialog="ExitDialog"
            Control="Finish"
            Event="DoAction"
            Value="LaunchApplication">WIXUI_EXITDIALOGOPTIONALCHECKBOX = 1 and NOT Installed</Publish>
    </UI>
    <Property Id="WIXUI_EXITDIALOGOPTIONALCHECKBOXTEXT" Value="Launch My Application Name" />

    <!-- Step 3: Include the custom action -->
    <Property Id="WixShellExecTarget" Value="[#myapplication.exe]" />
    <CustomAction Id="LaunchApplication"
        BinaryKey="WixCA"
        DllEntry="WixShellExec"
        Impersonate="yes" />
    </Product>
</Wix>
```

Edit this page

Version: v3

How To: Set Your Installer's Icon in Add/Remove Programs

Windows Installer supports a standard property, ARPPRODUCTICON, that controls the icon displayed in Add/Remove Programs for your application. To set this property you first need to include the icon in your installer using the Icon element, then set the property using the Property element.

```
<Icon Id="icon.ico" SourceFile="MySourceFiles\icon.ico"/>
<Property Id="ARPPRODUCTICON" Value="icon.ico" />
```

These two elements can be placed anywhere in your WiX project under the Product element. The Icon element specifies the location of the icon on your source machine, and gives it a unique id for use later in the WiX project. The Property element sets the ARPPRODUCTION property to the id of the icon to use.

Edit this page

Version: v3

How To: Implement a Major Upgrade In Your Installer

When creating an .msi-based installer, you are strongly encouraged to include logic that supports Windows Installer major upgrades. Major upgrades are the most common form of updates for .msi's, and including support in your initial .msi release gives you flexibility in the future. Without including support for major upgrades you risk greatly complicating your distribution story if you ever need to release updates later on.

You can use the following steps to enable major upgrades in your .msi, build multiple versions of your .msi and test major upgrade scenarios.

Step 1: Add upgrade information needed to cause new versions to upgrade older versions

In order to allow major upgrades, you must include the following information in your .msi:

Add a unique ID to identify that the product can be upgraded

To accomplish this, you must include an UpgradeCode attribute in your Product element. This looks like the following:

```
<Product Id="*"
         UpgradeCode="PUT-GUID-HERE"
         Name="My Application Name"
         Language="1033"
         Version="1.0.1"
         Manufacturer="My Manufacturer Name"/>
```

Schedule the removal of old versions and handle out-of-order installations

The MajorUpgrade element upgrades all older versions of the .msi. By default, it prevents out-of-order installations: installing an older version after installing a newer version.

```
<MajorUpgrade
  DowngradeErrorMessage="A later version of [ProductName] is already installed. Setup will now exit.">
```

There are several options for where you can schedule the RemoveExistingProducts action to remove old versions of the .msi. You need to review the options and choose the one that makes the most sense for your scenarios. You can find a summary of the options in the RemoveExistingProducts documentation.

By default, MajorUpgrade schedules RemoveExistingProducts after InstallValidate. You can change the scheduling using the Schedule attribute. For example, If you choose to schedule it after InstallInitialize, it will look like the following:

Windows Installer looks for other installed .msi files with the same UpgradeCode value during the FindRelatedProducts action. If you do not specifically schedule the FindRelatedProducts action in your setup authoring, WiX will automatically schedule it for you

```
<MajorUpgrade
  Schedule="afterInstallInitialize"
  DowngradeErrorMessage="A later version of [ProductName] is already installed. Setup will now exit.">
```

Step 2: Build version 1 and version 2 of your .msi

Creating version 1 of your .msi is as simple as running your standard build process - this means you compile and link it with the WiX toolset. In order to create version 2 of your .msi, you must make the following changes to your setup authoring, then re-run your build process to create a new .msi:

- Increment the Version value in your Product element to be higher than any previous versions that you have shipped. Windows Installer only uses the first 3 parts of the version in upgrade scenarios, so make sure to increment your version such that one of the first 3 parts is higher than any previously shipped version. For example, if your version 1 uses Version value 1.0.1.0, then version 2 should have a Version value of 1.0.2.0 or higher (1.0.1.1 will not work here).
- Generate a new Id value in the Product element of the new version of the .msi.

Step 3: Test upgrade scenarios before you ship version 1

This step is very important and is too often ignored. In order to make sure that upgrade scenarios will behave the way you expect, you should test upgrades before you ship the first version of your .msi. There are some upgrade-related bugs that can be fixed purely by making fixes in version 2 or higher of your .msi, but there are some bugs that affect the uninstall of version 1 that must be fixed before you ship version 1. Once version 1 ships, you are essentially locked into the uninstall behavior that you ship with version 1, and that impacts major upgrade scenarios because Windows Installer performs an uninstall of version 1 behind the scenes during version 2 installation.

Here are some interesting scenarios to test:

- Install version 1, then install version 2. Make sure that version 1 is correctly removed and version 2 functions correctly. Make

sure version 2 cleanly uninstalls afterwards.

- Install version 2, then try to install version 1. Make sure that version 1 correctly detects that version 2 is already installed and either blocks or silently exits, depending on what behavior you choose to implement for your out-of-order installation scenarios.

When testing major upgrade scenarios, make sure to pay particular attention to the conditions on custom actions in your .msi because you may run into issues caused by custom actions running during a major upgrade uninstall and leaving your product in a partially installed state. The UPGRADINGPRODUCTCODE property can be useful to prevent actions from running during an uninstall that is invoked by the RemoveExistingProducts action.

In addition, pay attention to assemblies that need to be installed to the GAC or the Win32 WinSxS store. There is some information about a sequence of events that can remove assemblies from the GAC and the WinSxS store during some major upgrades in this knowledge base article.

Edit this page

Version: v3

How To: Get a Log of Your Installation for Debugging

When authoring installers it is often necessary to get a log of the installation for debugging purposes. This is particularly helpful when trying to debug file searches and launch conditions. To obtain a log of an installation use the command line msiexec tool:

```
msiexec /i MyApplication.msi /l*v MyLogFile.txt
```

This will install your application and write a verbose log to MyLogFile.txt in the current directory.

If you need to get a log of your installer when it is launched from the Add/Remove Programs dialog you can enable Windows Installer logging via the registry.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Look Inside Your MSI With Orca

When building installers it can often be useful to look inside your installer to see the actual tables and values that were created by the WiX build process. Microsoft provides a tool with the Windows SDK, called Orca, that can be used for this purpose. To install Orca, download and install the Windows SDK. After the SDK installation is complete navigate to the install directory (typically **C:\Program Files\Microsoft SDKs\Windows\v7.0**) and open the **Tools** folder. Inside the Tools folder run Orca.msi to complete the installation. (If the Windows 8.1 SDK is installed, then Orca-x86.msi can typically be found in **c:\Program Files\Windows Kits\8.1\bin\x86**)

Once Orca is installed you can right click on any MSI file from Windows Explorer and select **Edit with Orca** to view the contents of the MSI.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Generate a GUID

GUIDs are used extensively with the Windows Installer to uniquely identify products, components, upgrades, and other key elements of the installation process. To generate GUIDs use the guidgen tool that ships with Visual Studio, generally located under **Tools > Create GUID** menu, or the GuidGen.com site. GUIDs generated this way will work fine in WiX, however since they are in mixed case they may cause issues if you share them with users of other, non-WiX tools. For complete compatibility be sure to change the letters in the GUID to uppercase prior to use.

All examples in the How To documentation use the text **PUT-GUID-HERE** for GUIDs. Every **PUT-GUID-HERE** must be replaced with a newly-generated GUID.

The Component, Package, Patch, Product elements support auto-generation of GUIDs every time you build your project by specifying a * in place of the GUID. For example:

```
<Product Id="*"
         Version="1.0.0.0"
         Language="1033"
         Name="My Application Name"
         Manufacturer="My Manufacturer Name">
```

For the Component element the generated GUID is based on the install directory and filename of the KeyPath for the component. This GUID will stay consistent from build-to-build provided the directory and filename of the KeyPath do not change.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Use WiX Extensions

The WiX extensions can be used both on the command line and within the Visual Studio IDE. When you use WiX extensions in the Visual Studio IDE, you can also enable IntelliSense for each WiX extension.

Using WiX extensions on the command line

To use a WiX extension when calling the WiX tools from the command line, use the -ext command line parameter and supply the extension assembly (DLL) needed for your project. Each extension DLL must be passed in via separate -ext parameters. For example:

```
light.exe MySetup.wixobj
 -ext WixUIExtension
 -ext WixUtilExtension
 -ext "C:\My WiX Extensions\FooExtension.dll"
 -out MySetup.msi
```

Extension assemblies in the same directory as the WiX tools can be referred to without path or .dll extension. Extension assemblies in other directories must use a complete path name, including .dll extension.

Note: Code Access Security manages the trust levels of assemblies loaded by managed code, including WiX extensions. By default, CAS prevents a WiX tool running on a local machine from loading a WiX extension on a network share.

Using WiX extensions in Visual Studio

To use a WiX extension when building in Visual Studio with the WiX Visual Studio package:

1. Right-click on the WiX project in the Visual Studio solution explorer and select Add Reference...
2. In the Add WiX Library Reference dialog, click on the Browse tab and browse to the WiX extension DLL that you want to include.
3. Click the Add button to add a reference to the chosen extension DLL.
4. Browse and add other extension DLLs as needed.

To enable IntelliSense for a WiX extension in the Visual Studio IDE, you need to add an XMLNS declaration to the <Wix> element in your .wxs file. For example, if you want to use the NativeImage functionality in the WixNetFxExtension, the <Wix> element would look like the following:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
     xmlns:netfx="http://schemas.microsoft.com/wix/NetFxExtension">
```

After adding this, you can add an element named <netfx:NativeImage/> and view IntelliSense for the attributes supported by the NativeImage element.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Optimize build speed

WiX provides two ways of speeding up the creation of cabinets for compressing files:

- Multithreaded cabinet creation.
- Cabinet reuse.

Multithreaded cabinet creation

Light uses multiple threads to build multiple cabinets in a single package. Unfortunately, because the CAB API itself isn't multithreaded, a single cabinet is built with one thread. Light uses multiple threads when there are multiple cabinets, so each cabinet is built on one thread.

By default, Light uses the number of processors/cores in the system as the number of threads to use when creating cabinets. You can override the default using Light's -ct switch or the CabinetCreationThreadCount property in a .wixproj project.

You can use multiple cabinets both externally and embedded in the .msi package (using the Media/@EmbedCab attribute).

Cabinet reuse

If you build setups with files that don't change often, you can generate cabinets for those files once, then reuse them without spending the CPU time to re-build and re-compress them.

There are two Light.exe switches involved in cabinet reuse:

-cc (CabinetCachePath property in .wixproj projects)
The value is the path to use to both write new cabinets and, when -reusecab/ReuseCabinetCache is specified, look for cached cabinets.
-reusecab (ReuseCabinetCache property in .wixproj projects)
When -cc/CabinetCachePath is also specified, WiX reuses cabinets that don't need to be rebuilt.

WiX automatically validates that a cached cabinet is still valid by ensuring that:

- The number of files in the cached cabinet matches the number of files being built.
- The names of the files are all identical.
- The order of files is identical.
- The timestamps for all files all identical.

Edit this page

Version: v3

How To: Specify source files

WiX provides three ways of identifying a setup package's payload - the files that are included in the setup and installed on the user's machine.

- By file name and directory tree.
- By explicit source file.
- Via named binder paths.

Compiling, linking, and binding

The WiX toolset models a typical C/C++ compiler in how authoring is built, with a compiler that parses the WiX source authoring to object files and a linker that combines the object files into an output. For WiX, the output is an .msi package, .msm merge module, or .wixlib library, which have a third phase: binding payload files into the output. Light.exe includes both the linker and binder.

Though WiX source authoring refers to payload files, the compiler never looks at them; instead, only the binder does, when it creates cabinets containing them or copies them to an uncompressed layout.

You can provide the binder with one or more *binder input paths* it uses to look for files. It also looks for files relative to the current

working directory. Light.exe's -b switch and the BindInputPaths .wixproj property let you specify one or more binder input paths.

Binder input paths can also be prefixed with a *name* which will append that path to the identified binder input path bucket (unprefixed paths will be added to the unnamed binder paths bucket). The bucket name must be more than two characters long and be followed by an equal sign ("="). See an example in the *Identifying payload via named binder paths* section

Identifying files by name and directory tree

When you use the File/@Name attribute and don't use the File/@Source attribute, the compiler constructs an implicit path to the file based on the file's parent component directory plus the name you supply. So, for example, given the partial authoring

```
<Directory Id="TARGETDIR">
  <Directory Name="foo">
    <Directory Name="bar">
      <Component>
        <File Name="baz.txt" />
```

the binder looks for a file *foo\bar\baz.txt* in the unnamed binder input paths.

Overriding implicit payload directories

The FileSource attribute for the Directory and DirectoryRef elements sets a new directory for files in that directory or any child directories. For example, given the partial authoring

```
<Directory Id="TARGETDIR">
  <Directory Name="foo" FileSource="build\retail\x86">
    <Directory Name="bar">
```

```
        <Component>
          <File Name="baz.txt" />
```

the binder looks for a file *build\retail\x86\bar\baz.txt* in the unnamed binder input paths.

The FileSource attribute can use preprocessor variables or environment variables. If the value is an absolute path, the binder's unnamed input paths aren't used.

Preferred use

If the build tree serving as your payload source is almost identical to the tree of your installed image and you have a moderate-to-deep directory tree, using implicit paths will avoid repetition in your authoring.

Source directories

The Directory/@SourceName attribute controls both the name of the directory where Light.exe looks for files and the "source directory" in the .msi package. Unless you also want to control the source directory, just use FileSource.

Identifying payload by source files

The File/@Source attribute is a path to the payload file. It can be an absolute path or relative to any unnamed binder input path. If File/@Source is present, it takes precedence over the implicit path created by Directory/@Name, Directory/@FileSource, and File/@Name.

If you specify File/@Source, you can omit File/@Name because the compiler automatically sets it to the filename portion of the source path.

Preferred use

If the build tree serving as your payload source is different from the tree of your installed image, using File/@Source makes it easy to pick explicit paths than are different than the .msi package's directory tree. You can use multiple unnamed binder input paths to shorten the File/@Source paths.

For example, the WiX setup .wixproj project points to the output tree for the x86, x64, and ia64 platforms WiX supports and the WiX source tree. Unique filenames can be referred to with just their filenames; files with the same name across platforms use relative paths.

See the WiX authoring in src\Setup*.wxs for examples.

Identifying payload via named binder paths

This is similar in authoring style to "Identifying payload by source files" while searching multiple paths like "Identifying files by name and directory tree". As such, it is sort of a hybrid between the two.

Named bind paths uses the File/@Source path prefixed with a bindpath variable like !(bindpath.*bucketname*). As with the unnamed binder paths used when the File/@Source is not present each path tagged with the same bucket name will be tested until a matching file is found. If the resulting path is not an absolute filepath, the unnamed binder file paths will be searched for each string in the bucket.

```
<File Source="!(bindpath.foo)bar\baz.txt" />
<File Source="!(bindpath.bar)baz\foo.txt" />

light -b foo=C:\foo\ -b bar=C:\bar\ -b foo=D:\
```

will look for the baz.txt file first at *C:\foo\bar\baz.txt* and then at *D:\bar\baz.txt*, using the first one found, while looking for the foo.txt file at *C:\bar\baz\foo.txt*; while

```
<File Source="!(bindpath.foo)bar\baz.txt" />
<File Source="!(bindpath.bar)baz\foo.txt" />

light -b foo=foo\ -b bar=bar\ -b foo=baz\
```

will search for the baz.txt file as if looking for two files having File/@Source values of *foo\bar\baz.txt* and *baz\bar\baz.txt* and will search for the foo.txt file as if the File/@Source was *bar\baz\foo.txt*.

Preferred use

If the build tree serving as your payload source places the same category of files in several locations and you need to search those locations differently for different categories of payload source files, using File/@Source with the "!(bindpath.*bucketname*)" prefix makes it easy to pick explicit groups of search paths. You can use multiple unnamed binder input paths to shorten the File/@Source paths and/or the unnamed binder paths.

For example, a partial build system may separate binary and non-binary files to different paths stored on a network share while the local override build may not have them separated. By prefixing the File/@Source values with the appropriate bindpath variable unique filenames can be referred to with just their filenames while files with the same name across platforms use relative paths.

Edit this page

> This is documentation for WiX Toolset **v3**, which is no longer actively maintained.
>
> For up-to-date documentation, see the **latest version** (v4.0).

Version: v3

How To: Install a Windows service

To install a Windows service, use the ServiceInstall element. Other configuration can be made using the ServiceControl element and the ServiceConfig element from WixUtilExtension.

Step 1: Install the service

The `ServiceInstall` element contains the basic information about the service to install. This element should be the child of a Component element whose key path is a sibling File element that specifies the service executable file.

Tip: to specify a system account, such as LocalService or NetworkService, use the prefix `NT AUTHORITY`. For example, use `NT AUTHORITY\LocalService` as the `Account` attribute value to run the service under this account.

Step 2: Configure the service (optional)

Using the util:ServiceConfig element from WixUtilExtension, you can configure how the service behaves if it fails. To use it, add WixUtilExtension to your project, add the the util namespace to your WiX authoring, and prefix the element name with the `util` prefix:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi" xmlns:util="http://schemas.microsoft.com
/wix/UtilExtension">
  ...
  <ServiceInstall>
      <util:ServiceConfig FirstFailureActionType="restart"
                          SecondFailureActionType="restart"
                          ThirdFailureActionType="restart" />
  </ServiceInstall>
```

Edit this page