TFSVersioning Custom Activity

Production Ready Assembly Versioning with TFS Build Workflow

Mark Nichols

January 22, 2013

Contents

Summary:

Versioning is part of Visual Studio projects within the AssemblyInfo file. It's easy to update the version once but how many people actually and effectively manage an application's version through this mechanism? I would bet that most only do it sporadically and eventually give up. Manual editing of the AssemblyVersion and AssemblyFileVersion is harder that it needs to be.

The goal of this project is to create a way to modify the automated build process of TFS 2010 so that versioning is automatic while giving the user the flexibility that they need given the project's process requirements. I've done this in the past using the angle-bracket programming paradigm of MSBuild – powerful but hard. This time I took advantage of the WF capabilities of TFS Build.

First: My Versioning Process Functional Requirements:

1. Must be able to define the AssemblyVersion and AssemblyFileVersion numbers separately
2. Must update the version for all of the assemblies within a solution with the same version numbers
3. Must provide an option to define the version patterns within the build definition **OR** within a version "seed file" that itself could be versioned in source control
4. Must provide the option to define the location and name of the seed file
5. If using the seed file, then the version numbers for a specific solution (.sln) must be definable by solution name
6. The seed file must be accessible as part of the source base of a project OR even outside of a project. This capability should allow multiple team projects to utilize a single seed file so version management can be managed from a central location
7. The seed file must be able to maintain multiple solution-specific version patterns in a single file
8. The seed file must be able to identify a separate version pattern for each of the multiple solutions simultaneously
9. If a solution name is not provided or is misspelled then a default version pattern should be able to be provided as a failsafe
10. Must be able to handle modifying versions within C#, VB and C++ projects
11. Must provide the option to automatically check the changes made to the AssemblyInfo file back in to source control during the build process (keeping all the code files in synch)
12. Must be able to define a versioning scheme appropriate to the needs of the application (i.e., use a pattern-based approach and/or explicit numbers to define what the version should be)
13. Version pattern options must provide ability for the build to automatically increment the version if necessary – such as with the AssemblyFileVersion so it can be directly tied back to a build, date, etc.

14. Must provide a way (if necessary to the project) to automatically differentiate version numbers when multiple builds use the same versioning pattern or seed file. In other words, The version number in an assembly must be traceable back to a build, its build definition AND the associated source code (this is an extension of requirement #12)
15. Must be a simple addition to a TFS 2010 workflow-based build process
16. Must use inherent TFS build capabilities – do not require any installation/deployment to the build server (GAC or local file system)

The Solution

After several iterations of design and implementation, I came up with a combination of custom code workflow activities (C#) and non-code workflow activities (XAML) that are all combined into a single composite activity that can be added to an existing TFS 2010 build process. I tried to build in as many tests as I could which were highly valuable for letting me know when I broke things but I also used them during development to trigger the execution of the custom activities without having to run them through a full build process. It made development and debugging tons easier.

Ultimately, I created seven custom activities but only one of them "VersionAssemblyInfoFiles" is necessary to be inserted into the overall build process. That is because "VersionAssemblyInfoFiles" is a composite of the other six XAML and code activities that do all the work.

Version Definition Options

As stated in the requirements list, you have the option of defining the version in the build definition or in a "seed file". You may be thinking why would I need anything more than being able to define the version in the build definition? After all, the build definition performs the build, right? This is true BUT…the build definition only performs one rendition of the build. What if you need to create multiple build definitions to build the same set of code? I almost always create multiple builds. So, the seed file allows you to define a single versioning pattern and use it across any number of build definitions.

Now some background…

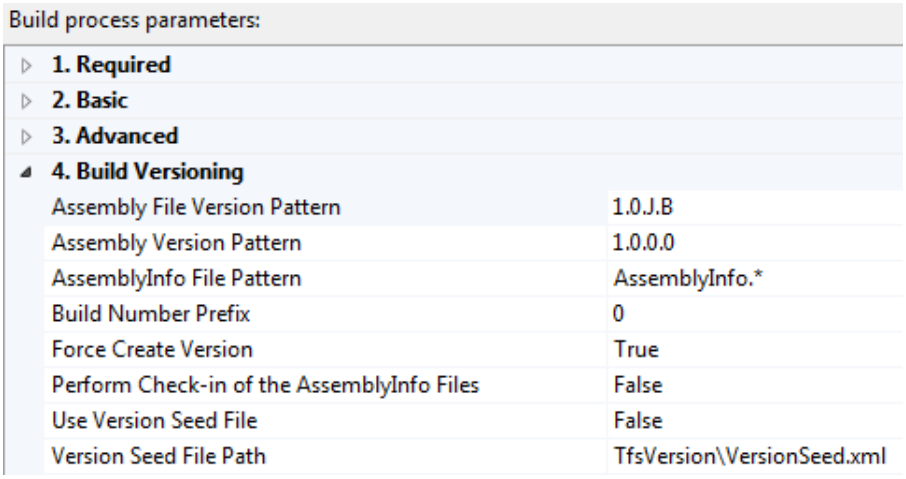
Assembly Version Numbers:

- AssemblyVersion: This is also the product version. During development, it is the version number that you are working towards. For example, version 2.3.4.5
- AssemblyFileVersion: This number can be similar to the AssemblyVersion but should indicate a specific build. For example, 2.3.11070.5 indicates that the major/minor is the same as the

AssemblyVersion but the build/revision uses a julian date and build number so that it is unique no matter what day or how many builds occur. Well, mostly unique – to make it completely unique review "Build Number Prefix" below.

Build Versioning Parameters

The Build Versioning parameters will appear when you select the build process template that contains the updated build workflow (more on that below). They will appear with defaults similar to below assuming that the build workflow is modified as described below.



Assembly Version and Assembly File Version Patterns

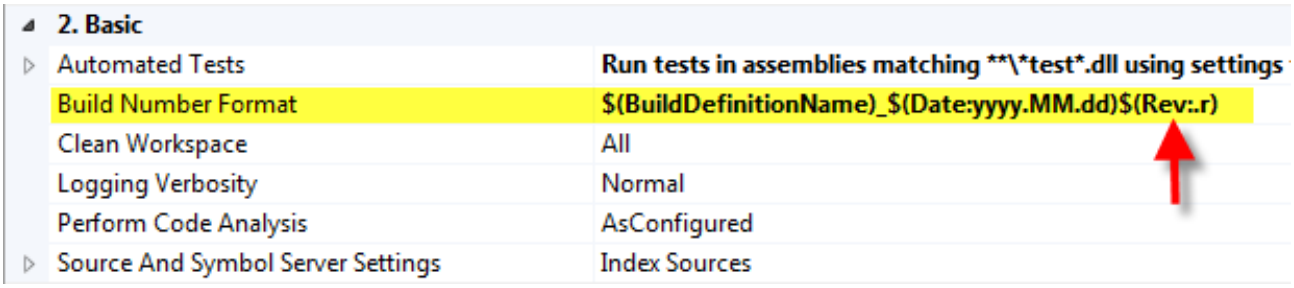
You have two choices for how you want to specify where the build will get the "pattern" for replacing the assembly version numbers.

- The first is the easiest; you just enter the patterns in the build definition ("Assembly File Version Pattern" and "Assembly Version Pattern" parameters). Those values will then be passed through to the build workflow.
- The second is a slightly harder but much more flexible. An XML "seed file" is used to hold and specify the patterns. The benefits of this approach include:
    - The same version file can be used across any number of build definitions
    - A single file can contain patterns for multiple solutions and will even work if multiple solutions are specified in a single build – each solution receives their own version patterns
    - Since it is a file, it is versioned along with the source code and therefore history is maintained

Numeric and Symbol Patterns:

A version (AssemblyVersion or AssemblyFileVersion) can be 1 to 4 numbers (Major[.Minor][.Build][.Revision]) where the maximum numeric value for each is 65535 (a U16 number)

- If a number is used in any position in the version pattern then that number is passed through unchanged
- Use a symbol pattern and that value will be replaced in the AssemblyInfo file. The symbols are:
    - YYYY: Replaced with the current 4-digit year
    - YY: Replaced with the current 2-digit year
    - M or MM: Replaced with the number for the current month (MM does not give you a leading 0)
    - D or DD: Replaced with the number for the current day (DD does not give you a leading 0)
    - J: Replace with the current date in "Julian" 5-digit format (YYDDD where YY is the year and DDD is the number of the day within the year e.g., 11075 is March 16, 2011 – there are leading 0's for the day)
    - B: Replace with the current build number for the day. Note, using this pattern requires that the "Build Number Format" ends in "$(Rev:.r)". TFS does create the build number format with this "macro" at the end as the default so unless you change it there won't be a problem.



Examples:

"yyyy.mm.dd.b" - If you queued up the 2nd build of the day on April 26, 2011 the version would be: "**2011.4.26.2**"

"1.0.J.B" – Again, if you queued up the 2nd build of the day on April 26, 2011 the version would be: "**1.0.11116.2**" (This is the default for the assembly file version)

AssemblyInfo File Pattern:

This parameter is included if you want to specify a different name or different pattern for finding the file that should contain the AssemblyVersion and AssemblyFileVersion entries. Normally, you will not need to change this parameter but it is included for completeness and flexibility.

Build Number Prefix:

The ability to use a seed file to version assemblies across multiple build definitions definitely can ease the management of version numbers and lessen the maintenance within the definitions themselves. However, it creates another problem: different build definitions can generate the exact same version numbers. (See the example below)

The Build Number Prefix provides you with the ability to use the same version pattern across multiple build definitions and still be able to look at the version and trace it back to the build definition that built it and even the source code that was used. So, in other words, you can have your CI build, a daily build and even multiple manual builds and always be able to determine what built the code and what source was used. You can also look at a version number and determine if someone decided to insert an assembly into production that was from the wrong build.

The approach is simple. By entering a value in the "Build Number Prefix" parameter, that value will be added to the Build Number in the version (this assumes that the "B" flag is used in the version pattern – typically in the AssemblyFileVersion). So, if the build number is 1 and the Build Number Prefix is 100 then the version will indicate 101.

Example:

Let's say you have a situation like we already described: 2 build definitions (a CI and a daily build) and both definitions use the following pattern for the AssemblyFileVersion: "1.2.J.B". Now, queue up an instance of both definitions (assuming they are the 1st builds of the day). When the build process is done, you will have assemblies in 2 different drop folders with the exact same AssemblyFileVersion (e.g.: "1.2.11155.1"). So, to solve that problem, enter the value 100 in the Build Number Prefix parameter for the CI build and a value of 200 in the Build Number Prefix for the daily build. Now, queue them both up again. The resulting version numbers will be different (CI will be "1.2.11155.102" and the daily build will be "1.2.11155.202"). Now they are different and will continue to be different. I can even trace all the way back to the source code that built it via the build definition and the label attached to the source. You can make this tracing even easier if you add the number you use for the prefix to the Build Definition Name. Then, the version, the build definition and the label will all be aligned to the same naming/versioning pattern and easy to search for and find.

Note: The prefix number must always be larger than the build number or you will get number clashes. The example above assumes that you would never build more than 99 times in a day. If that is a possibility then just change the prefix to 1000. If you do more than 999 builds in a day then please contact me because you're probably melting the build servers and I would love to hear about it. In any event, you could put 10000 in the prefix to handle it. No matter what, prefix + build number can't go over the UInt16 value of 65535.

Force Create Version:

A “True” value here indicates to the process that even if an AssemblyVersion or AssemblyFileVersion entry does not already exist within the AssemblyInfo files, they should be created. A “False” value will key off their existence in the AssemblyInfo file and will only update a version if it is indicated. Maybe you want the AssemblyVersion but you don’t want to update the AssemblyFileVersion – just set this value to “False” and leave the AssemblyFileVersion entry out of the AssemblyInfo file and no AssemblyFileVersion will be created.

Perform Check-In of the AssemblyInfo File(s):

This parameter is a boolean that lets you say that you want the AssemblyInfo files checked back into source control after they are modified with the correct version numbers and before the source is labeled. This will keep the history current within source control. It defaults to “False” so therefore will not perform a check-in but change to true if you want them checked in.

Use Version Seed File:

This property is an indicator that defaults to “False” which tells the build to use the version patterns in the build definition. Change this to a value of “True” to tell the build to use the version “seed file”. See the descriptions below to find out more about the benefits of using the seed file to manage the version patterns across multiple builds.

Version Seed File Path:

The version seed file is an alternative to using the version patterns included in the build definition. By using the seed file, you can instruct any number of build definitions to use the same version number patterns across all of the builds. So, if you have a CI build and daily build, they both can use the same data file to define the version parameters. Another benefit of the seed file is that you can manage versioning multiple solutions with the same seed file. The solutions can be built separately or (if you wish) you can build multiple solutions in the same build definition and each solution is versioned with its own version patterns.

The example below shows the XML structure for the seed file. There is an overall “VersionSeed” group that contains the “Solutions”. Each solution element corresponds to a Visual Studio solution by name. So, the versioning activity will read the seed file and look for an element name that matches the name

of the solution. If it finds one, it will then extract the patterns to be used for the versioning replacement.

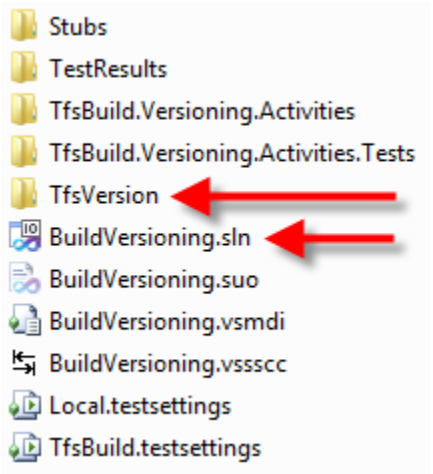
```
<VersionSeed>
 <Solution name="BuildVersioning">
  <AssemblyVersionPattern>1.0.3.0</AssemblyVersionPattern>
  <AssemblyFileVersionPattern>1.0.j.b</AssemblyFileVersionPattern>
 </Solution>
 <Solution name="Default">
  <AssemblyVersionPattern>1.0.2.0</AssemblyVersionPattern>
  <AssemblyFileVersionPattern>1.0.j.b</AssemblyFileVersionPattern>
 </Solution>
</VersionSeed>
```

To continue with the example, I have a TFS Project that contains a solution file called "BuildVersioning.sln". With this seed file, the "**BuildVersioning**" solution assemblies will all receive an AssemblyVersion of 1.0.3.0. All assemblies will also receive an AssemblyFileVersion of "1.0.11106.5" (for example - see the pattern translation below for Julian date and buildnumber). The "Default" value is used if there the solution being built does not have a corresponding Solution Name in the seed file. It is a safety catch but could also be used to version multiple solutions with the same version numbers (I don't have a good reason why you would want to do that but then everyone has different needs).

Relative Path or Source File Path

Relative Path:

The default value included in the build definition properties is "TfsVersion\VersionSeed.xml". This assumes that you will create a folder named "TfsVersion" within the solution and in that folder is a file named "VersionSeed.xml". This is using the relative path approach.

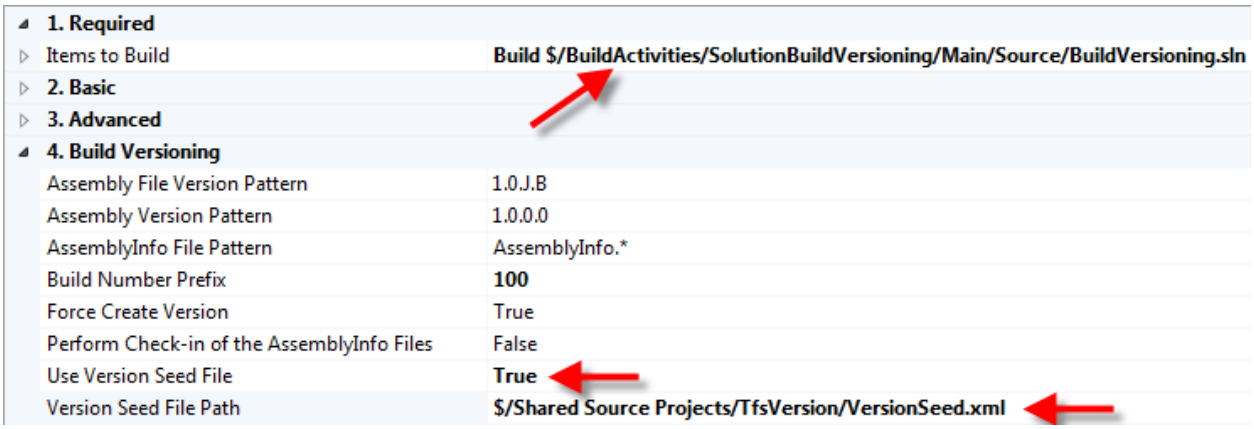


Building multiple solutions in the same build definition is easy within TFS 2010. If you want to do this, you can move the folder containing the seed file up a level or two (e.g.: "..\..\TfsVersion\VersionSeed.xml"). Then, the build will look at where the current solution being built is located and then traverse upward to the folder containing the seed file. Note: If you do this, the folder containing the seed file must be included in the Source Control Folder when you identify the Workspace in the build definition. Obviously, this is necessary so the file exists locally when the build server gets the source code.

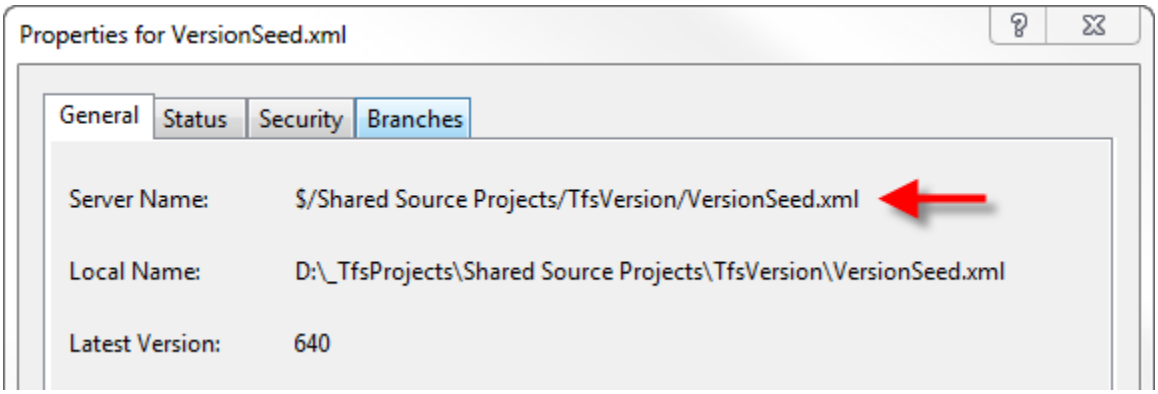
Source File Path:

You have a second option to identify the seed file. Instead of a relative path, you can use a complete source file path to specify where, in source control, the seed file exists. This approach solves a number of issues. Going this route, you can manage version numbers across multiple TFS Team Projects rather than just multiple solutions within a single TFS project. You also don't have to worry about making sure the seed file exists in the workspace identified in the build definition or if the seed file is the same distance away from each of the solution files.

The only difference is in how you identify the file. Rather than using relative path notation, you use source control notation (e.g.: "$/Project/folder/file.ext"). For example, below you can see that the solution is in the BuildActivities Team Project but the seed file is in the "Shared Source Projects" Team Project. I did this just to show the possibilities of the feature – the file does not need to be in a different Team Project. Also note that the "Use Version Seed File" property is set to "True". "False" means that the version patterns will be taken right from the build definition.



The easiest way to grab the path is to just right-click on the file in Team Explorer and go to properties. You can then just highlight and copy the file location (where ever you decide to put it).

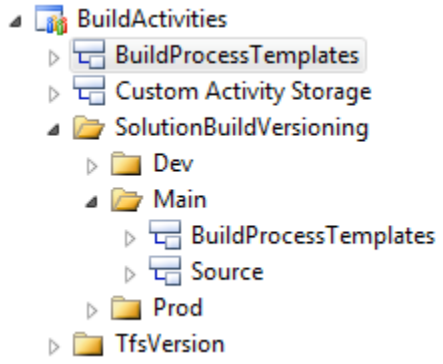


Installation and Verification:

Simplified Installation:

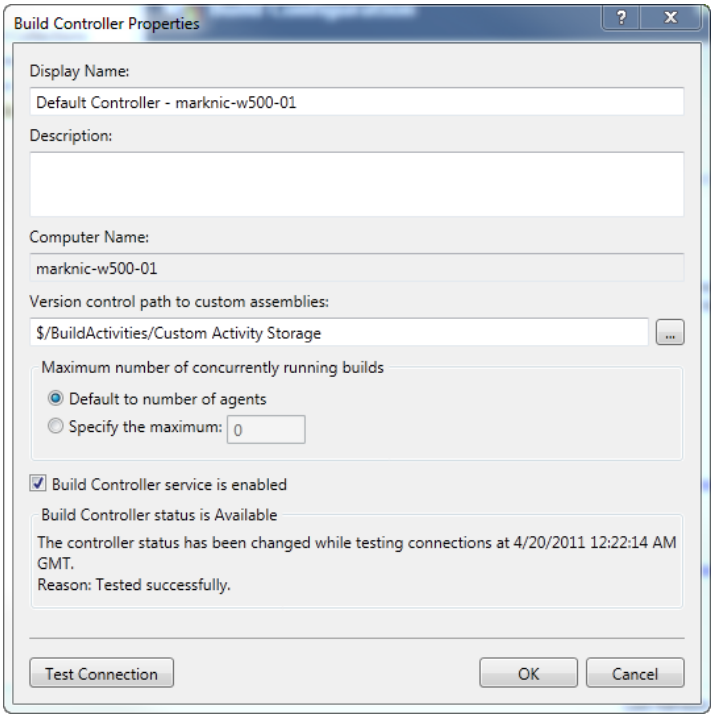
This method of installation assumes that you will use the modified workflow (XAML file) that is included with the source. The name of the workflow is "VersioningBuildTemplate.xaml".

1. Create or identify a Team Project to hold the activity assembly and the updated versioning build workflow (XAML) file. Personally, I created a Team Project called BuildActivities to hold the source for this workflow addon. In that same project I hold the updated build process template as well as the custom activity assembly that does all of the versioning work.
2. Create or identify a folder to hold the custom versioning activity assembly. The TFS build controller keeps track of a folder in source control where all the custom activity assemblies can be stored and used in build workflows. Below you can see the one I created "Custom Activity Storage"



3. Add the VersioningBuildTemplate.xaml file to your BuildProcessTemplates folder and add the TfsBuild.Versioning.Activities.dll file to your custom activity storage folder
4. Check in the additions to source control
5. Start up the Team Foundation Server Administration Console. You need to tell the TFS Build Controller where it should look to find any custom build activity assemblies

6. Go to “Build Configuration” and select the build controller properties. The window should look something like the one below:



7. Click the ellipsis for the “Version control path to custom assemblies” and navigate to the folder where you added the TfsBuild.Versioning.Activities.dll file. After you identify the folder in source control, click OK

That’s it for installation. You can close down the administration console for TFS.

Testing the installation

1. Either create or go to a Team Project where you can test the build versioning process.
2. Create a new build definition (you could modify an existing one if you want)
3. Fill in the Name, Trigger, Workspace and Build Defaults as you normally would
4. When you get to “Process”, click on “Show details” in the “Build process template” section
5. Open up the “Build process file” drop-down and look for the “VersioningBuildTemplate.xaml” file and select it. If it is not there, click on the “New” button, click on the “Select an existing XAML file” radio button and browse to the folder location where you checked in the XAML file and then click “OK”
6. Within the “Build process parameters”, a new “Build Versioning” section should appear that contains all of the properties described previously

This is all you have to do to get the build updated with versioning. The defaults should work just fine to create assemblies with an assembly version number of “1.0.0.0” (which isn’t too amazing since that’s the default that comes in the assemblyInfo file) BUT the assembly file version should be recognizable

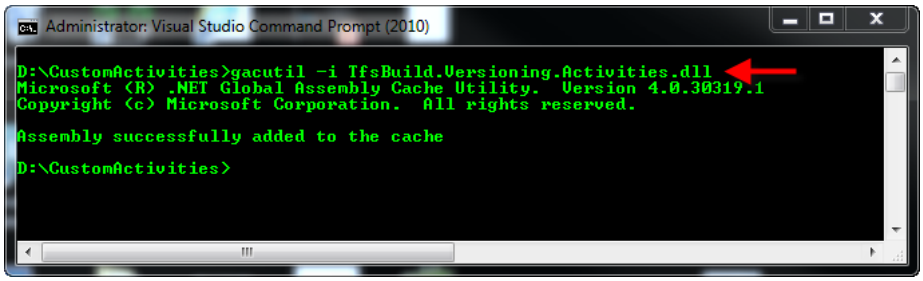
that it has been changed. It should be "1.0.#####.1". The "#####" number depends on the day you run the build but if it were April 22, 2011 then the number would be 11112. In any event it will be a 5 digit number with the 1st two digits being the year.

Cue up the build and see what happens.

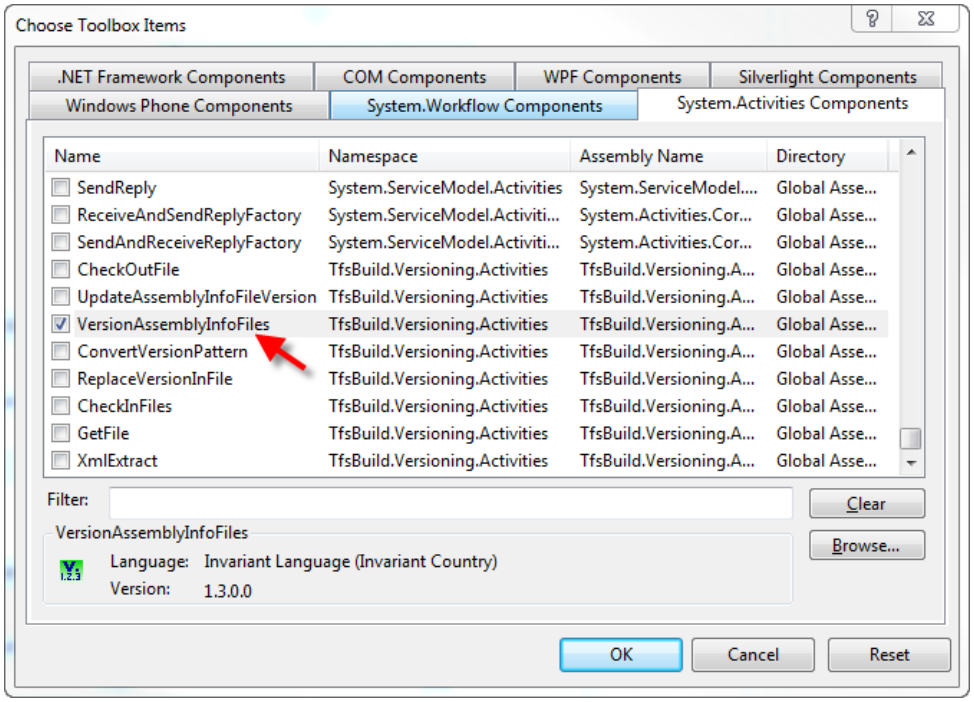
Harder Installation but More Instructive

This version of the installation assumes that you want to be able to modify your own workflow and insert the custom versioning activity. In this option, the versioning assembly must be added to the GAC and therefore signed. The reason it needs to be added to the GAC is so the workflow editor in Visual Studio will see the assembly and allow it to be added to the workflow. The assembly included in the download is signed (albeit not very securely since the key file doesn't use a password). You can use this assembly or compile your own with your own key.

1. Use the GACUtil utility to install the assembly to the Global Assembly Cache
    a. Go to "All Programs\ Microsoft Visual Studio 2010\ Visual Studio Tools\ Visual Studio Command Prompt" and start a command window with elevated privileges ("As Administrator")
    b. Install the assembly into the GAC with a command like this:

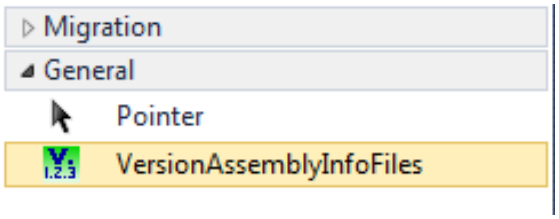


2. Open Visual Studio and the build workflow file (xaml) that you wish to modify
3. You need to tell the Visual Studio toolbox that you want to use the custom activity
    a. If you want, add a tab in the toolbox to contain the custom versioning activity. This is not at all necessary but it does organize the activities.
    b. Add the custom activity to the toolbox by right-clicking on the tab where you want the custom activity to reside. The tab should highlight to tell you where it will be placed.
    c. Select "Choose Items…" from the popup menu. This will bring up an aptly-named "Choose Toolbox Items" window
    d. The "System.Activities Components" tab will probably be highlighted. Click "Browse…" and go find the activity assembly. You just need to browse to the file not the GAC.



e. All of the activities in the assembly you just added will be checked by default. You can make things a little easier for later if you uncheck those activities and only check the "VersionAssemblyInfoFiles" activity. The others are used by "VersionAssemblyInfoFiles" to make up the composite activity.

f. Click "OK" and the toolbox will be updated

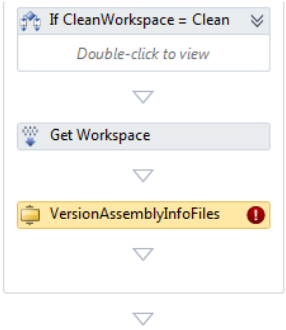


4. Create the "Arguments" in the workflow so that you can pass information to the activity during the build definition and the build itself. Note that all of the arguments use "In" as the direction and below I am giving you the Name, Argument type and Default value:

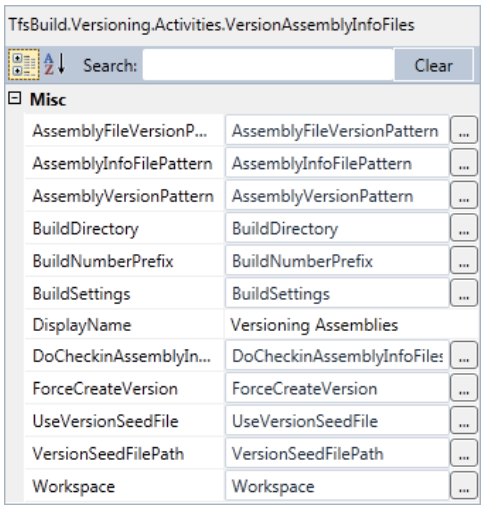
Argument Name	Type	Default Value
VersionSeedFilePath	String	"TfsVersion\VersionSeed.xml"
ForceCreateVersion	Boolean	True
DoCheckinAssemblyInfoFiles	Boolean	False
AssemblyVersionPattern	String	"1.0.0.0"
AssemblyFileVersionPattern	String	"1.0.J.B"

AssemblyInfoFilePattern	String	"AssemblyInfo.*"
UseVersionSeedFile	Boolean	False
BuildNumberPrefix	Int32	0

5. Insert the activity and tie the arguments to it:
    a. Search for the right position to place the activity by looking in the "Run On Agent" sequence. This can get confusing since the workflow will expand when you try to view areas that it hasn't shown before. So, collapse the activities as you scan downward. For example, right inside of "Run On Agent" is "Initialize Variables" and if you click on the up arrows on the right side of the activity it will collapse down. The next sequence below "Initialize Variables" should be "Initialize Workspace" and that's where the activity should go – right below "Get Workspace" (after the source is retrieved and before anything is labeled)
    b. Drag "VersionAssemblyInfoFiles" from the toolbox and place it right below "Get Workspace". It will tell you that there is an error since there are required values that have to be filled in.



    c. Add the arguments to the properties. All of the arguments that were created above need to be added to the activity properties but you will see that there are more properties than the number of arguments that we created. This is because the activity uses some of the arguments and variables that already exist in the workflow. ("BuildSettings" is an existing argument, "Workspace" and "BuildDirectory" are existing variables). The graphic below shows you what to change and when you're done the error icon should go away:



6. Almost there. Now you need to add the metadata to the workflow so that the build definition will know to ask you the right questions as you create the build. In the "Arguments" for the workflow, look for "Metadata" and click on the ellipsis.
    a. In the "Process Parameter Metadata Editor" you need to add an entry for each argument that will get passed from the build definition to the build workflow. Use the table below to copy/paste the entries in to the editor.
    b. The "Category" value for each of the parameters should be identical so the properties are all grouped together. I named the category "Build Versioning".
    c. Leave the "Required" box unchecked for all of the entries
    d. You can modify the "View this parameter when:" however you would like or you can leave the values at the default of "Only while editing a definition"

Parameter Name	Display Name	Description
AssemblyFileVersionPattern	Assembly File Version Pattern	This is the pattern used to replace the AssemblyFileVersion value.
AssemblyVersionPattern	Assembly Version Pattern	This is the pattern used to replace the AssemblyVersion value.
AssemblyInfoFilePattern	AssemblyInfo File Pattern	This is the pattern used to find the AssemblyInfo files. Generally, you shouldn't need to change this value.
DoCheckinAssemblyInfoFiles	Perform Check-in of the AssemblyInfo Files	Indicated whether the AssemblyInfo files should be checked back into source control after they are modified.
ForceCreateVersion	Force Create Version	If true, the versioning process will create AssemblyVersion or AssemblyFileVersion values even if they do not already exist.
UseVersionSeedFile	Use Version Seed File	Indicate which values to use as the versioning patterns. If set to True,

		the "seedfile.xml" file must exist in the location described by the "Version Seed File Path" setting. Otherwise, the "Assembly Version Pattern" and "Assembly File Version Pattern" values will be used.
VersionSeedFilePath	Version Seed File Path	Relative path location for the seed (xml) file containing the Assembly Version and Assembly File Version values.
BuildNumberPrefix	Build Number Prefix	Number added to the version component that uses the "B" symbol pattern (Build Number). This helps create a unique version for a build definition.

7. Last step. This step is a verification step. Sometimes, depending on how the workflow is edited an entry may be left out.
    a. Edit the build workflow in text mode so you can see the XML/XAML code
    b. Search for "TfsBuild.Versioning.Activities". The entry should be something like this:

```
xmlns:tva="clr-namespace:TfsBuild.Versioning.Activities;assembly=TfsBuild.Versioning.Activities"
```

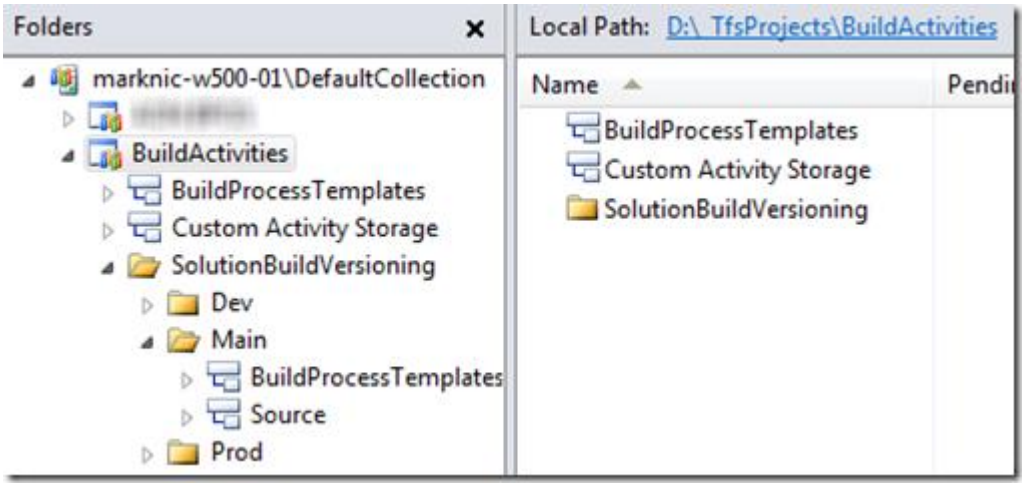
    c. If the "`;assembly=TfsBuild.Versioning.Activities`" statement is not there then definitely complete the entry and put it in manually otherwise the build will most likely fail when it is run.

That's it for editing you can now test the build versioning. If you want more instruction on how to do this, just go back up to "Testing the Installation" but you should be ready to create a new build definition (or modify an old one) using the workflow you just created and version your assemblies.

How to Implement or Create Your Own

Here are the steps required to install/implement this solution. This may look like a lot of things to do just to get versioning going BUT what I am showing is how to do everything from source code to creating the build. If I could provide you with the updated build template and the custom activity assembly then you would only have to create a folder and tell the controller where to look for the assembly. After that, it would just be a matter of creating build definitions for your projects. That's it. So, what we have below is a learning experience. You'll be able to use the descriptions and lessons learned to help you create your own custom activities and build workflows.

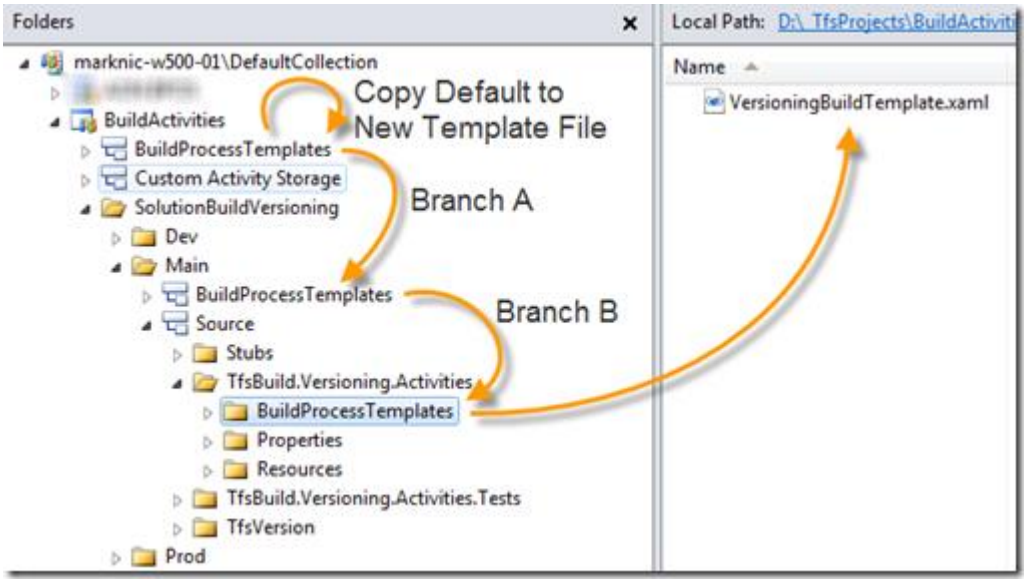
1. **Create a Team Project in TFS**. This is to hold the source code of the custom activities, the updated build process template and a folder to hold the custom assembly(s) for all workflows. Note: the build controller maintains a single path (within version control) to store custom assemblies used within any build process. Since they are in version control, the build process will always know where to look – there is no need to install anything on any of the build machines. In a momentary lack of creativity, I named the Team Project "BuildActivities". I now use the BuildProcessTemplates folder in this Team Project as the base storage location for all other projects that use the versioning build process.

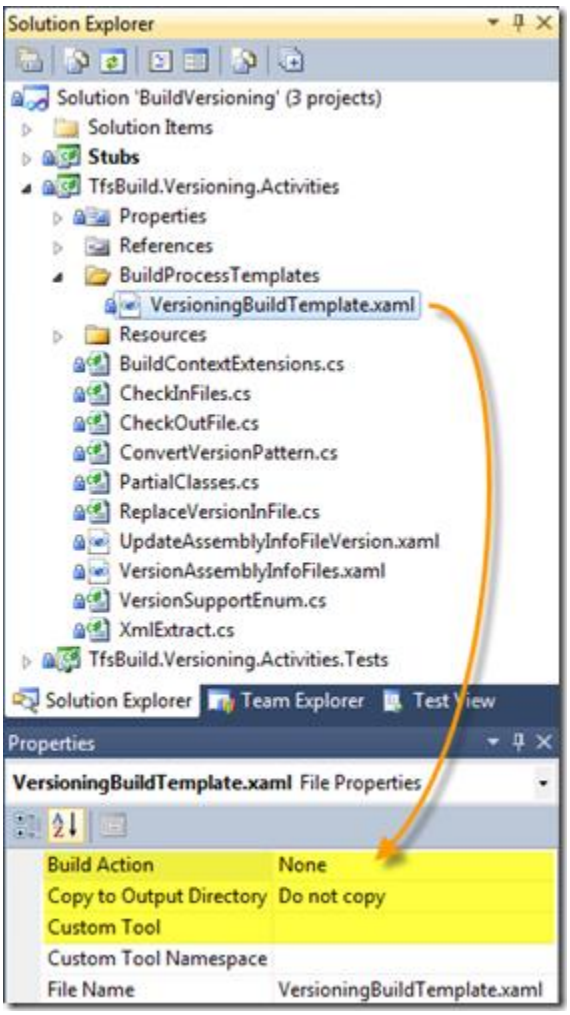


2. **Place the source code provided below in a folder within TFS** so you can edit and save your changes to version control. As you can see above, I called that folder "SolutionBuildVersioning"

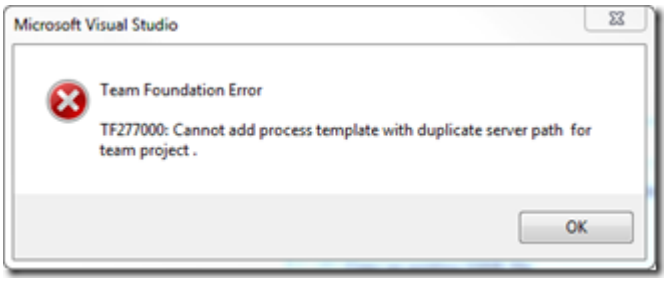
3. P**rep for the modification of the build process**. I created a copy of the "DefaultTemplate.xaml" within the BuildProcessTemplates folder and called it "VersioningBuildTemplate.xaml". Then I branched the BuildProcessTemplates folder into the Main branch of my source code ("Branch A"). I did this so I could make modifications and test the updates to the template without messing with the files in what I am now using for production. Finally, I branched again ("Branch B") within my custom activities solution folders AND added the XAML file to my solution. This allows me to edit the XAML workflow within Visual Studio and the workflow will recognize my

custom activities in the designer toolbox so I could just drag my versioning activity into the workflow. Note: although the XAML template was part of my solution, it was not compiled or included in the assembly. After I made the necessary changes to the XAML file, I merged the changes up a level.





Lesson learned: I tried to test the updated XAML build workflow template by running a build on the BuildVersioning solution and you can't do that. This is because the build template needs to be outside of the workspace that you specify for the solution that you are building. In other words, you can't cross the streams. However, all I needed to do was merge the changes in the build template XAML file up a single level (to the folder pointed to by "Branch A" above) and all was good. You will know this is the problem if you try to create a build definition and get this error:

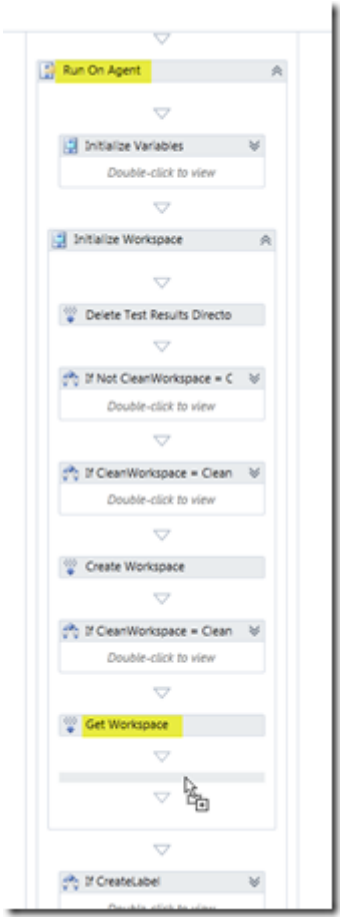


Another General Lesson: Branching and merging saves time! All I needed to do to get the build template where I could test it was to right-click, merge and check-in. Getting it to the production

location was just one more merge. Learn it, live it, love it.

4. **Compile the solution** and then edit the "VersioningBuildTemplate.xaml". The custom activities will (hopefully) automatically display in the activity toolbox. Sometimes they don't. If not, right-click on the "General" tab and select "Choose Items…". Scroll to the bottom of the "System.Activities Components" tab and look for "VersionAssemblyInfoFiles". Check the box next to it and click OK and the activity will appear in the toolbox.

5. **In the build workflow, look for the "Get Workspace" activity.** It is in the "Run On Agent" sequence activity. You want to insert the "VersionAssemblyInfoFiles" activity right after "Get Workspace". This location is right after the source is retrieved and right before the source is labeled. It is a perfect location for making modifications to files so the changes get compiled and even checking those changes back in to source control as the changes will then be labeled along with the rest of the source code. Drag the "VersionAssemblyInfoFiles" activity onto the workflow as below:



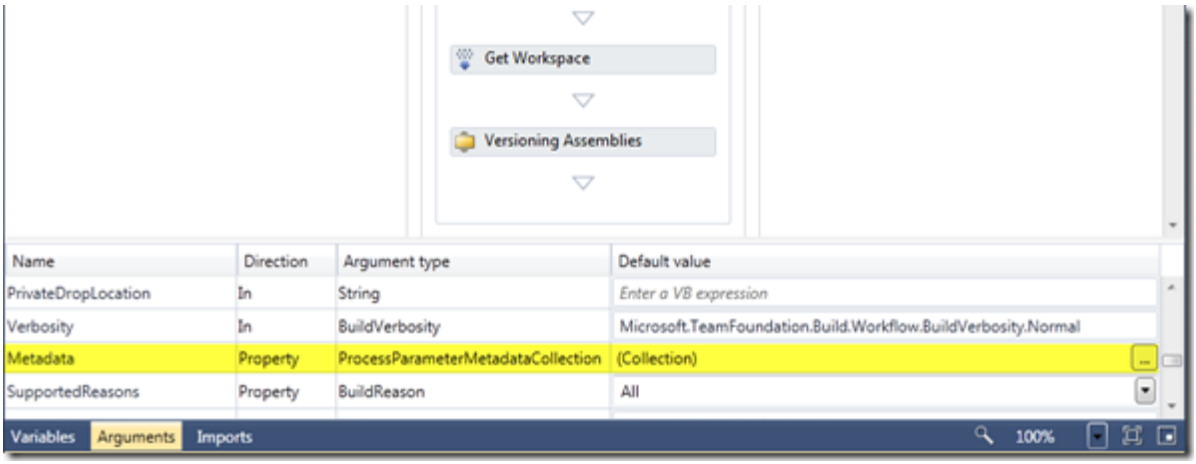
6. Change the activity's a "Display Name" property to something that you will want to see in the build window and log. I named mine "Versioning Assemblies". I know, very creative.

7. **Create the 7 arguments in the workflow so that you can pass values to the activity during the build.** A little background here: Arguments in a workflow are name/value pairs global to that workflow. They are used to pass data into and/or out of a workflow and it doesn't matter where you are in the workflow, they are always accessible. Variables, on the other hand, are local to the area in the workflow where they are defined. For example, let's say you have two sequence activities in a workflow (a sequence is a container that holds one or more child activities that are executed in the order that they appear in their parent sequence container). Now, let's say sequence A contains sequence B. If we click on sequence B, then click on "Variables" at the bottom of the workflow editor and then click on "Create Variable" and give it a name, type and value (e.g., SourceFilePath, String, "$\ProjName\folder\filename.ext", that variable will be visible to any activity inside of sequence B. However, "SourceFilePath" is invisible to all of the other activities in sequence A.
   Ok, Create the following Arguments (They are all "In" arguments)

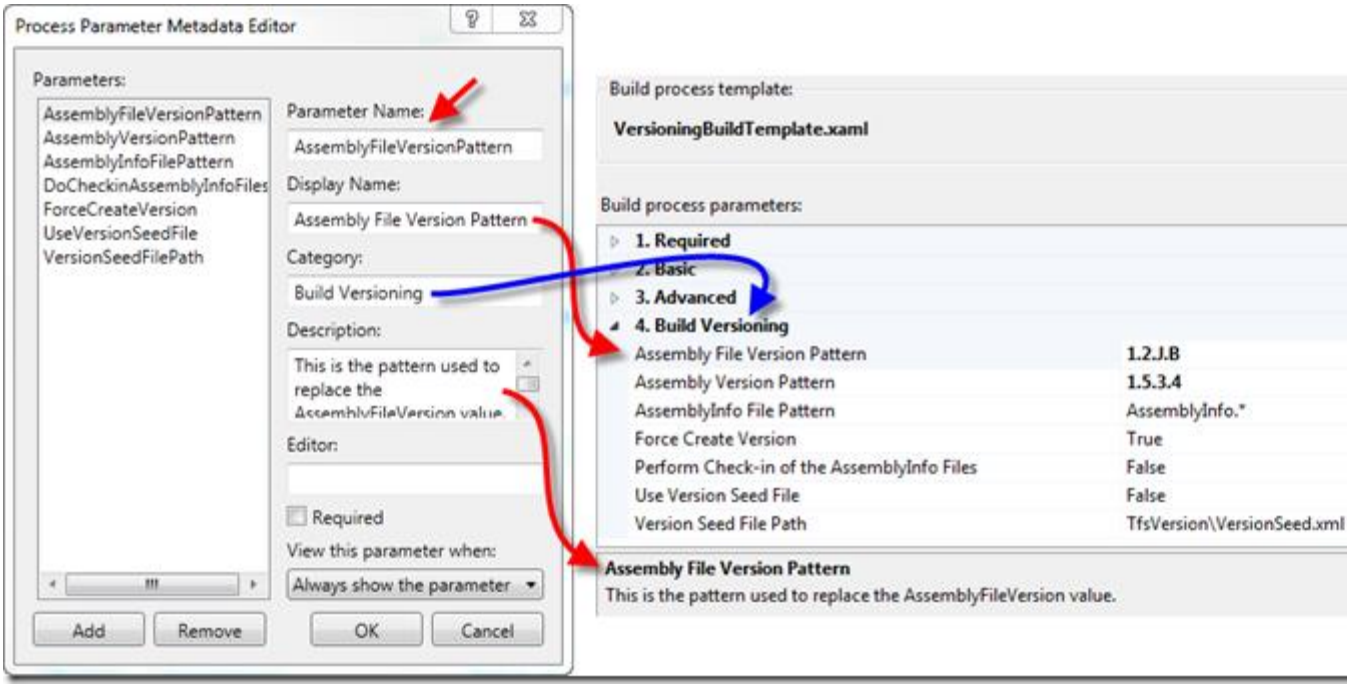
Argument Name	Type	Default Value
AssemblyFileVersionPattern	String	"1.0.J.B"
AssemblyVersionPattern	String	"1.0.0.0"
AssemblyInfoFilePattern	String	"AssemblyInfo.*"
VersionSeedFilePath	String	"TfsVersion\VersionSeed.xml"
DoCheckinAssemblyInfoFiles	Boolean	False
ForceCreateVersion	Boolean	True
UseVersionSeedFile	Boolean	False

8.

9. Add the arguments to the workflows's metadata so you can modify the values in the Build Definition. Just look for "Metadata" in the build workflow's arguments and click on the ellipsis.



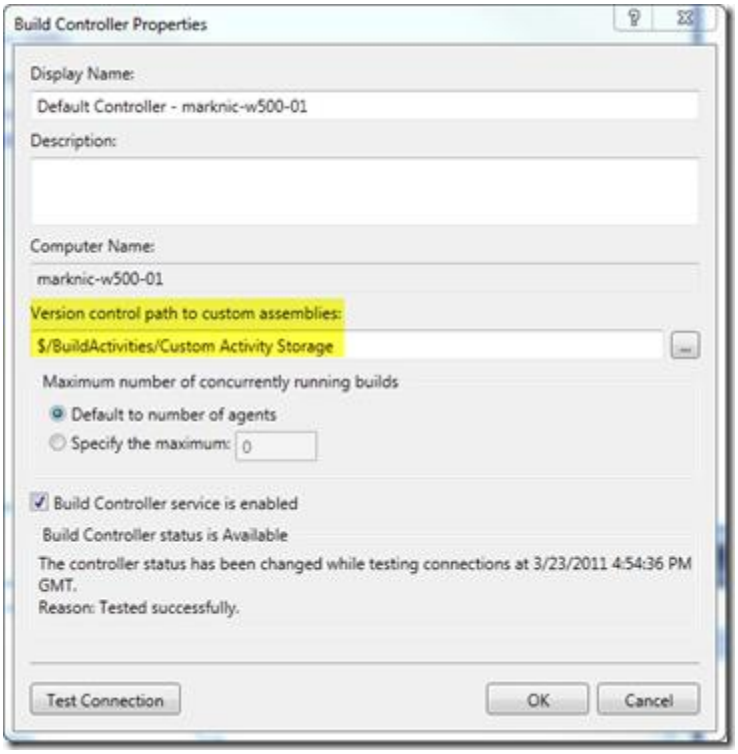
Enter the data for each of the 7 arguments. The “Parameter Name” is the argument that will end up with the value provided in the build definition. “Display Name” is what is displayed in the build definition. “Category” is where these arguments/parameters will be grouped within the build definition and “Description” appears at the bottom of the build definition if you click on the parameter (“help text”). The “View this parameter when” drop-down lets you say when you will see the parameter (Never, always, only in the build definition or only in the



Lesson Learned: The “Required” checkbox is only helpful if you need the user to change the value in the build definition. If you provide default values in the arguments (as we did) then do not check this box. If so, it will require the user to change the default value to something else. So, it’s not all that helpful: yes we require a value but we also want to be able to use the default.

What I would rather have is for “Required” to mean non-blank.

10. **Create a source folder to hold the custom build assembly(s) and tell your build controller where it is**. If you look at the picture in step 1, you can see that I created a folder called “Custom Activity Storage”. This is where I put the assembly that I created that contains all of the custom activities used in this project. If/when I create additional build activity assemblies, that is also where I will put them. The only other thing you have to do to make that folder official is tell the TFS build controller. Start up the Team Foundation Server Administration Console and go to the Build Controller Properties - as you can see below, you just browse to the folder that will contain the assembly(s):



11. Place the source code provided below in a folder within TFS so you can edit and save your changes to version control. Here is how I did it. I created a folder within BuildActivities that maintains my source.

To do your own editing: There are a few steps required to introduce any custom activity into the existing build process so here are some suggested steps including what I did:

1. You can either: use the solution provided below (and do your editing of the build template) or you can try creating your own solution or you can try to edit the template directly. No matter what, the visual editor in VS2010 will need to be able to find the assembly containing the activities that you want in the process. You could deploy the assembly to the GAC but that’s

wouldn't satisfy requirement #13 and you will have to sign the assembly, and honestly, isn't necessary.

2. I created a TFS Project to contain my custom activities and modified build templates. Of course you want to store your source in TFS but I also have two folders in that project that all my other projects have access to. One folder holds the build templates and the other holds the custom activity assemblies. I notified the build controller where that folder is within source control so the assembly is available for all builds that need it.
3. Make a copy of the default build process template and include it within your solution somewhere. It's always a good idea to work on a backup rather than the real thing but additionally, by including the file within your solution, you will see the custom activities in the Toolbox. Then you drag the activity out on the workflow design canvass. Set the Build Action property to None and Do Not Copy to the output directory. In my environment, I branched the "DefaultTemplate.xaml" to a new file called "VersioningBuildTemplate.xaml" (you could just make a copy) and then I branched the new template into a folder within my solution. This way I can make changes within the context of my solution (so the editor can find assembly references) and then I can merge the changes all the way back where I will actually use the template.

Tips and Tricks

- When you add a custom activity assembly to a build template all will look fine until you try to use it within Team Build. There's a good chance you will get a error saying that the build process cannot create an unknown type – and then it will display they type of the activity that you just created. To get around this, open the build template XAML file in text mode. Then, at the top, look for a reference to your custom activity. The one I have included here will look like this: xmlns:local="clr-namespace:TfsBuild.Versioning.Activities". Change that text to include the assembly: xmlns:local="clr-namespace:TfsBuild.Versioning.Activities;assembly=TfsBuild.Versioning.Activities" and all should be good.

Result

- A simple modification to an existing build template
- Provides the necessary versioning for assemblies within a solution and across builds
- Resulting assemblies can easily be tied back to a labeled set of source code
- The effort required to maintain the version numbers is minimized to some initial setup of the build and a single file (only if you use the seed file approach)