What do they all mean?

In this post I look at the various different version numbers you can set for a .NET Core project, such as Version, VersionSuffix, and PackageVersion. For each one I'll describe the format it can take, provide some examples, and what it's for.

*This post is****very****heavily inspired by*[*Nate McMaster*](https://twitter.com/natemcmaster)*'s*[*question (which he also answered) on Stack Overflow*](https://stackoverflow.com/questions/42183300/what-is-the-difference-between-various-msbuild-version-properties-such-as-versi)*. I'm mostly just reproducing it here so I can more easily find it again later!*

Version numbers in .NET

.NET loves version numbers - they're sprinkled around everywhere, so [figuring out what version of a tool](https://andrewlock.net/the-net-core-2-0-preview-1-version-numbers-and-global-json/) you have is sometimes easier said than done.

Leaving aside the tooling versioning, .NET also contains a plethora of version numbers for you to add to your assemblies and NuGet packages. There are at least *seven* different version numbers you can set when you build your assemblies. In this post I'll describe what they're for, how you can set them, and how you can read/use them.

The version numbers available to you break logically into two different groups. The first group, below, exist only as MSBuild properties. You can set them in your *csproj* file, or pass them as command line arguments when you build your app, but their values are only used to control *other* properties; as far as I can tell, they're not visible directly anywhere in the final build output:

* [VersionPrefix](https://andrewlock.net/version-vs-versionsuffix-vs-packageversion-what-do-they-all-mean/#versionprefix)
* [VersionSuffix](https://andrewlock.net/version-vs-versionsuffix-vs-packageversion-what-do-they-all-mean/#versionsuffix)
* [Version](https://andrewlock.net/version-vs-versionsuffix-vs-packageversion-what-do-they-all-mean/#version)

So what are they for then? They control the default values for the version numbers which *are* visible in the final build output.

* [FileVersion](https://andrewlock.net/version-vs-versionsuffix-vs-packageversion-what-do-they-all-mean/#fileversion)
* [AssemblyVersion](https://andrewlock.net/version-vs-versionsuffix-vs-packageversion-what-do-they-all-mean/#assemblyversion)
* [InformationalVersion](https://andrewlock.net/version-vs-versionsuffix-vs-packageversion-what-do-they-all-mean/#informationalversion)
* [PackageVersion](https://andrewlock.net/version-vs-versionsuffix-vs-packageversion-what-do-they-all-mean/#packageversion)

I'll explain each number in turn, then I'll explain [how you can set the version numbers when you build your app](https://andrewlock.net/version-vs-versionsuffix-vs-packageversion-what-do-they-all-mean/#how-to-set-the-version-number-when-you-build-your-app-library).

VersionPrefix

* **Format**: major.minor.patch[.build]
* **Examples**: 0.1.0, 1.2.3, 100.4.222, 1.0.0.3
* **Default**: 1.0.0
* Typically used to set the overall SemVer version number for your app/library

You can use VersionPrefix to set the "base" version number for your library/app. It indirectly controls all of the other version numbers generated by your app (though you can override it for other specific versions). Typically, you would use a [SemVer 1.0 version number](http://semver.org/spec/v1.0.0.html) with three numbers, but technically you can use between 1 and 4 numbers. If you don't explicitly set it, VersionPrefix defaults to 1.0.0.

VersionSuffix

* **Format**: Alphanumberic (+ hyphen) string: [0-9A-Za-z-]\*
* **Examples**: alpha, beta, rc-preview-2-final
* **Default**: (blank)
* Sets the pre-release label of the version number

VersionSuffix is used to set the pre-release label of the version number, if there is one, such as alpha or beta. If you don't set VersionSuffix, then you won't have any pre-release labels. VersionSuffix is used to control the Version property, and will appear in PackageVersion and InformationalVersion.

Version

* **Format**: major.minor.patch[.build][-prerelease]
* **Examples**: 0.1.0, 1.2.3.5, 99.0.3-rc-preview-2-final
* **Default**: VersionPrefix-VersionSuffix (or just VersionPrefix if VersionSuffix is empty)
* The most common property set in a project, used to generate versions embedded in assembly.

The Version property is the value most commonly set when building .NET Core applications. It controls the default values of all the version numbers embedded in the build output, such as PackageVersion and AssemblyVersion so it's often used as the single source of the app/library version.

By default, Version is formed from the combination of VersionPrefix and VersionSuffix, or if VersionSuffix is blank, VersionPrefix only. For example,

* If VersionPrefix = 0.1.0 and VersionSuffix = beta, then Version = 0.1.0-beta
* If VersionPrefix = 1.2.3 and VersionSuffix is empty, then Version = 1.2.3

Alternatively, you can explicitly overwrite the value of Version. If you do that, then the values of VersionPrefix and VersionSuffix are effectively unused.

The format of Version, as you might expect, is a combination of the VersionPrefix and VersionSuffix formats. The first part is typically a [SemVer](http://semver.org/spec/v1.0.0.html) three-digit string, but it can be up to four digits. The second part, the pre-release label, is an alphanumeric-plus-hyphen string, as for VersionSuffix.

AssemblyVersion

* **Format**: major.minor.patch.build
* **Examples**: 0.1.0.0, 1.2.3.4, 99.0.3.99
* **Default**: Version without pre-release label
* The main value embedded into the generated *.dll*. An important part of assembly identity.

Every assembly you produce as part of your build process has a version number embedded in it, which forms an important part of the assembly's identity. It's stored in the assembly manifest and is used by the runtime to ensure correct versions are loaded etc.

*The AssemblyVersion is used along with name, public key token and culture information only if the assemblies are strong-named signed. If assemblies are not strong-named signed, only file names are used for loading. You can read more about assembly versioning*[*in the docs*](https://docs.microsoft.com/en-us/dotnet/framework/app-domains/assembly-versioning#Anchor_0)*.*

The value of AssemblyVersion defaults to the value of Version, but without the pre-release label, and expanded to 4 digits. For example:

* If Version = 0.1.2, AssemblyVersion = 0.1.2.0
* If Version = 4.3.2.1-beta, AssemblyVersion = 4.3.2.1
* If Version = 0.2-alpha, AssemblyVersion = 0.2.0.0

The AssemblyVersion is embedded in the output assembly as an attribute, System.Reflection.AssemblyVersionAttribute. You can read this value by inspecting the executing Assembly object:

using System;

using System.Reflection;

class Program

{

static void Main(string[] args)

{

var assembly = Assembly.GetExecutingAssembly();

var assemblyVersion = assembly.GetName().Version;

Console.WriteLine($"AssemblyVersion {assemblyVersion}");

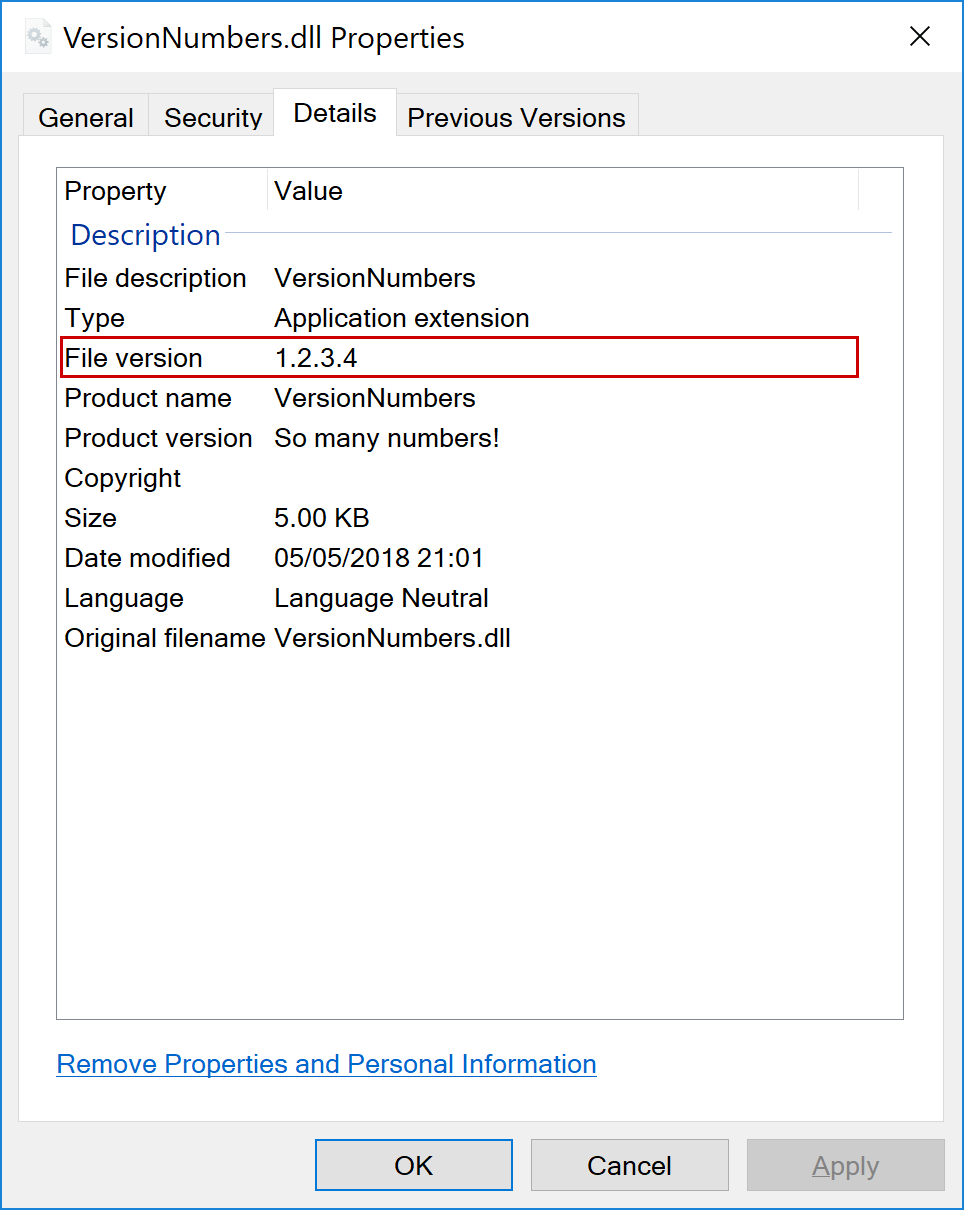
}

}

FileVersion

* **Format**: major.minor.patch.build
* **Examples**: 0.1.0.0, 1.2.3.100
* **Default**: AssemblyVersion
* The file-system version number of the *.dll* file, that doesn't *have* to match the AssemblyVersion, but usually does.

The file version is literally the version number exposed by the DLL to the file system. It's the number displayed in Windows explorer, which often matches the AssemblyVersion, but it doesn't have to. The FileVersion number isn't part of the assembly identity as far as the .NET Framework or runtime are concerned.



*When strong naming was more heavily used, it was common to keep the same AssemblyVersion between different builds and increment FileVersion instead, to avoid apps having to update references to the library so often.*

The FileVersion is embedded in the System.Reflection.AssemblyFileVersionAttribute in the assembly. You can read this attribute from the assembly at runtime, or you can use the FileVersionInfo class by passing the full path of the assembly (Assembly.Location) to the FileVersionInfo.GetVersionInfo() method:

using System;

using System.Diagnostics;

using System.Reflection;

class Program

{

static void Main(string[] args)

{

var assembly = Assembly.GetExecutingAssembly();

var fileVersionInfo = FileVersionInfo.GetVersionInfo(assembly.Location);

var fileVersion = fileVersionInfo.FileVersion;

Console.WriteLine($"FileVersion {fileVersion}");

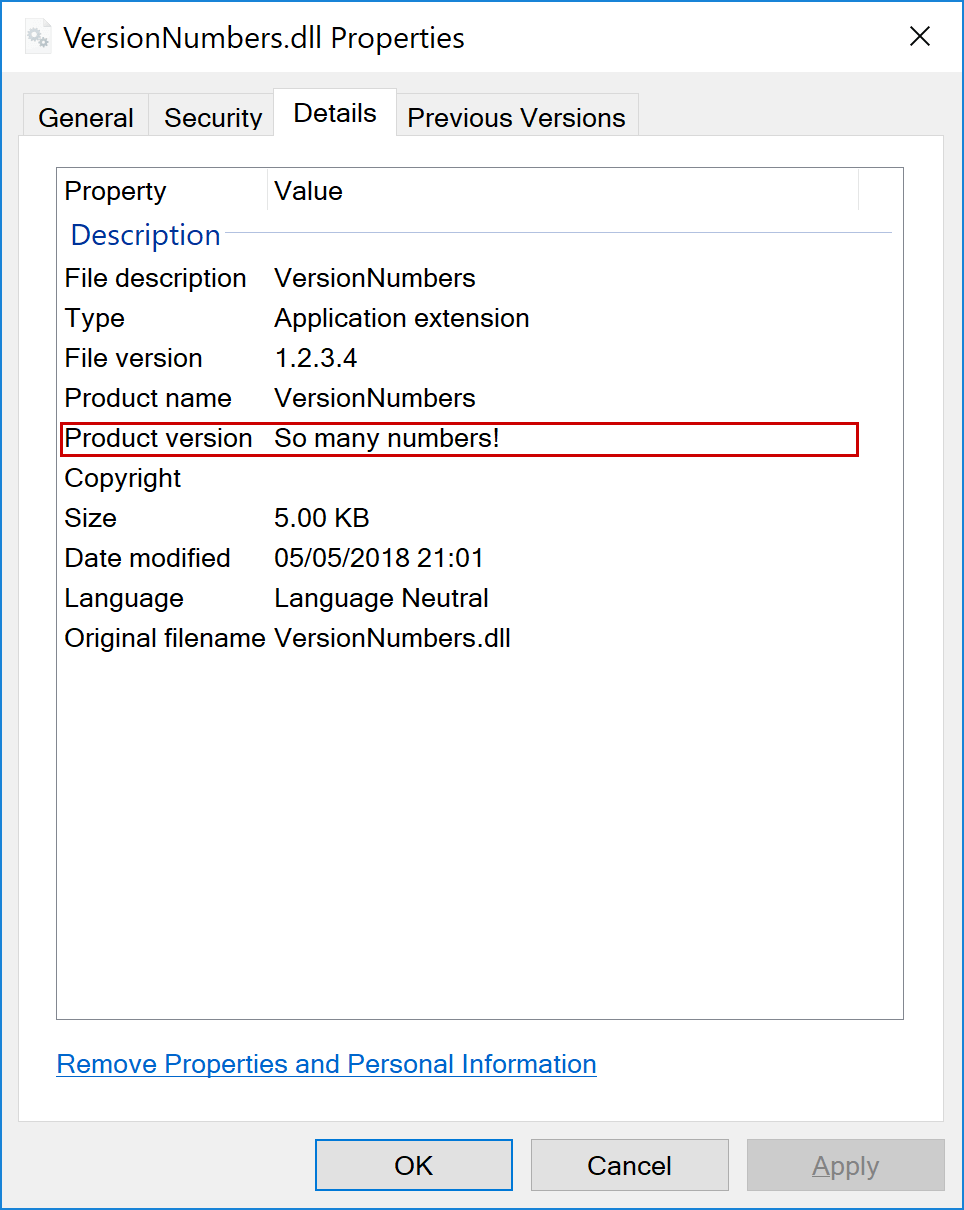
}

}

InformationalVersion

* **Format**: anything
* **Examples**: 0.1.0.0, 1.2.3.100-beta, So many numbers!
* **Default**: Version
* Another information number embedded into the DLL, can contain any text.

The InformationalVersion is a bit of an odd-one out, in that it doesn't need to contain a "traditional" version number per-se, it can contain any text you like, though by default it's set to Version That makes it generally less useful for programmatic uses, though the value is still displayed in Windows explorer:



The InformationalVersion is embedded into the assembly as a System.Reflection.AssemblyInformationalVersionAttribute, so you can read it at runtime using the following:

using System;

using System.Reflection;

class Program

{

static void Main(string[] args)

{

var assembly = Assembly.GetExecutingAssembly();

var informationVersion = assembly.GetCustomAttribute<AssemblyInformationalVersionAttribute>().InformationalVersion;

Console.WriteLine($"InformationalVersion {informationVersion}");

}

}

PackageVersion

* **Format**: major.minor.patch[.build][-prerelease]
* **Examples**: 0.1.0, 1.2.3.5, 99.0.3-rc-preview-2-final
* **Default**: Version
* Used to generate the NuGet package version when building a package using dotnet pack

PackageVersion is the only version number that isn't embedded in the output dll directly. Instead, it's used to control the version number of the NuGet package that's generated when you call dotnet pack.

By default, PackageVersion takes the same value as Version, so it's typically a three value SemVer version number, with or without a pre-release label. As with all the other version numbers, it can be overridden at build time, so it can differ from all the other assembly version numbers.

How to set the version number when you build your app/library

That's a lot of numbers, and you can technically set every one to a different value! But if you're a bit overwhelmed, don't worry. It's likely that you'll only want to set one or two values: either VersionPrefix and VersionSuffix, or Version directly.

You can set the value of *any* of these numbers in several ways. I'll walk through them below.

Setting an MSBuild property in your csproj file

With .NET Core, and the simplification of the *.csproj* project file format, adding properties to your project file is no longer an arduous task. You can set any of the version numbers I've described in this post by setting a property in your *.csproj* file.

For example, the following *.csproj* file sets the Version number of a console app to 1.2.3-beta, and adds a custom InformationalVersion:

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<OutputType>Exe</OutputType>

<TargetFramework>netcoreapp2.0</TargetFramework>

<Version>1.2.3-beta</Version>

<InformationalVersion>This is a prerelease package</InformationalVersion>

</PropertyGroup>

</Project>

Overriding values when calling dotnet build

As well as hard-coding the version numbers into your project file, you can also pass them as arguments when you build your app using dotnet build.

If you just want to override the VersionSuffix, you can use the --version-suffix argument for dotnet build. For example:

dotnet build --configuration Release --version-suffix preview2-final

If you want to override any other values, you'll need to use the MSBuild property format instead. For example, to set the Version number:

dotnet build --configuration Release /p:Version=1.2.3-preview2-final

Similarly, if you're creating a NuGet package with dotnet pack, and you want to override the PackageVersion, you'll need to use MSBuild property overrides

dotnet pack --no-build /p:PackageVersion=9.9.9-beta

Using assembly attributes

Before .NET Core, the standard way to set the AssemblyVersion, FileVersion, and InformationalVersion were through attributes, for example:

[assembly: AssemblyVersion("1.2.3.4")]

[assembly: AssemblyFileVersion("6.6.6.6")]

[assembly: AssemblyInformationalVersion("So many numbers!")]

However, if you try to do that with a .NET Core project you'll be presented with errors!

> Error CS0579: Duplicate 'System.Reflection.AssemblyFileVersionAttribute' attribute

> Error CS0579: Duplicate 'System.Reflection.AssemblyInformationalVersionAttribute' attribute

> Error CS0579: Duplicate 'System.Reflection.AssemblyVersionAttribute' attribute

As the SDK sets these attributes automatically as part of the build, you'll get build time errors. Simply delete the assembly attributes, and use the MSBuild properties instead.

Alternatively, as [James Gregory](https://twitter.com/jamesjrg) [points out on Twitter](https://twitter.com/jamesjrg/status/993903720919379973), you can still use the Assembly attributes in your code if you turn off the auto-generated assembly attributes. You can do this by setting the following property in your *csproj* file:

<PropertyGroup>

<GenerateAssemblyInfo>false</GenerateAssemblyInfo>

</PropertyGroup>

This could be useful if you already have tooling or a CI process to update the values in the files, but otherwise I'd encourage you to embrace the new approach to setting your project's version numbers.

Summary

In this post I described the difference between the various version numbers you can set for your apps and libraries in .NET Core. There's an overwhelming number of versions to choose from, but generally it's best to just set the Version and use it for all of the version numbers.