



Configuration Management Software Versioning Guidance

Summary

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Overview

This strategy is based on but not necessarily limited to pre-existing widespread common practices in use in both closed and open-source software. For this system to work, you first need to declare a public API. This may consist of documentation or be enforced by the code itself. Regardless, it is important that this API be clear and precise. Once you identify your public API, you communicate changes to it with specific increments to your version number. Consider a version format of X.Y.Z (Major.Minor.Patch). Bug fixes not affecting the API increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version.

Under this scheme, version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next.

Versioning Specification

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described at <https://tools.ietf.org/html/rfc2119>.

You must declare a public API. This API could be declared in the code itself or exist strictly in documentation. However it is done, it should be precise and comprehensive.

A normal version number MUST take the form X.Y.Z where X, Y, and Z are non-negative integers, and MUST NOT contain leading zeroes. X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically. For instance: 1.9.0 -> 1.10.0 -> 1.11.0.

Once a versioned package has been released, the contents of that version MUST NOT be modified. Any modifications MUST be released as a new version.

Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable.

Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.

Patch version Z ($x.y.Z \mid x > 0$) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.

Minor version Y ($x.Y.z \mid x > 0$) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.

Major version X ($X.y.z \mid X > 0$) MUST be incremented if any backwards incompatible changes are introduced to the public API. It MAY include minor and patch level changes. Patch and minor version MUST be reset to 0 when major version is incremented.

A pre-release version MAY be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch version. Identifiers MUST comprise only ASCII alphanumerics and hyphen [0-9A-Za-z-]. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT include leading zeroes. Pre-release versions have a lower precedence than the associated normal version. A pre-release version indicates that the version is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version. Examples: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.

Build metadata MAY be denoted by appending a plus sign and a series of dot separated identifiers immediately following the patch or pre-release version. Identifiers MUST comprise only ASCII alphanumerics and hyphen [0-9A-Za-z-]. Identifiers MUST NOT be empty. Build metadata SHOULD be ignored when determining version precedence. Thus two versions that differ only in the build metadata, have the same precedence. Examples: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85.

Precedence refers to how versions are compared to each other when ordered. Precedence MUST be calculated by separating the version into major, minor, patch and pre-release identifiers in that order (Build metadata does not figure into precedence). Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: Major, minor, and patch versions are always compared numerically. Example: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1. When major, minor, and patch are equal, a pre-release version has lower precedence than a normal version. Example: 1.0.0-alpha < 1.0.0. Precedence for two pre-release versions with the same major, minor, and patch version MUST be determined by comparing each dot separated identifier from left to right until a difference is found as follows: identifiers consisting of only digits are compared numerically and identifiers with letters or hyphens are compared lexically in ASCII sort order. Numeric identifiers always have lower precedence than non-numeric identifiers. A larger set of pre-release fields has a higher precedence than a smaller set, if all of the preceding identifiers are equal. Example: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0.

FAQ

How should I deal with revisions in the 0.y.z initial development phase?

The simplest thing to do is start your initial development release at 0.1.0 and then increment the minor version for each subsequent release.

How do I know when to release 1.0.0?

If your software is being used in production, it should probably already be 1.0.0. If you have a stable API on which users have come to depend, you should be 1.0.0. If you're worrying a lot about backwards compatibility, you should probably already be 1.0.0.

Doesn't this discourage rapid development and fast iteration?

Major version zero is all about rapid development. If you're changing the API every day you should either still be in version 0.y.z or on a separate development branch working on the next major version.

If even the tiniest backwards incompatible changes to the public API require a major version bump, won't I end up at version 42.0.0 very rapidly?

This is a question of responsible development and foresight. Incompatible changes should not be introduced lightly to software that has a lot of dependent code. The cost that must be incurred to upgrade can be significant. Having to bump major versions to release incompatible changes means you'll think through the impact of your changes, and evaluate the cost/benefit ratio involved.

Documenting the entire public API is too much work!

It is your responsibility as a professional developer to properly document software that is intended for use by others. Managing software complexity is a hugely important part of keeping a project efficient, and that's hard to do if nobody knows how to use your software, or what methods are safe to call. In the long run, Semantic Versioning, and the insistence on a well defined public API can keep everyone and everything running smoothly.

What do I do if I accidentally release a backwards incompatible change as a minor version?

As soon as you realize that you've broken the Semantic Versioning spec, fix the problem and release a new minor version that corrects the problem and restores backwards compatibility. Even under this circumstance, it is unacceptable to modify versioned releases. If it's appropriate, document the offending version and inform your users of the problem so that they are aware of the offending version.

What should I do if I update my own dependencies without changing the public API?

That would be considered compatible since it does not affect the public API. Software that explicitly depends on the same dependencies as your package should have their own dependency specifications and the author will notice any conflicts. Determining whether the change is a patch level or minor level modification depends on whether you updated your dependencies in order to fix a bug or introduce new functionality. I would usually expect additional code for the latter instance, in which case it's obviously a minor level increment.

What if I inadvertently alter the public API in a way that is not compliant with the version number change (i.e. the code incorrectly introduces a major breaking change in a patch release)?

Use your best judgment. If you have a huge audience that will be drastically impacted by changing the behavior back to what the public API intended, then it may be best to perform a major version release, even though the fix could strictly be considered a patch release. Remember, Semantic Versioning is all about conveying meaning by how the version number changes. If these changes are important to your users, use the version number to inform them.

How should I handle deprecating functionality?

Deprecating existing functionality is a normal part of software development and is often required to make forward progress. When you deprecate part of your public API, you should do two things: (1) update your documentation to let users know about the change, (2) issue a new minor release with the deprecation in place. Before you completely remove the functionality in a new major release there should be at least one minor release that contains the deprecation so that users can smoothly transition to the new API.

Does a version have a size limit on the version string?

No, but use good judgment. A 255 character version string is probably overkill, for example. Also, specific systems may impose their own limits on the size of the string.