Meaningful Version Numbering

Versioning Rules for
RLOTTO

– prozumr –

March 30, 2018

Abstract

Objective of this document is to describe conventions used as basic concept for handling of version numbers in a particular open source software project called RLOTTO as published on github[1]. Moreover the concept described herein has the potential to be generalized and used for other software projects even though this is not the primary intention of this document.
Please note: The concept or approach described in this document is up to a significant extent derived from another well known concept called *Semantic Versioning*[2].

Introduction

The need for version numbers in the world of software development is obvious and does not need any further explanation. However a strict set of rules and regulations how to approach versioning and how to determine version numbers is neither trivial nor self-evident and requires a bunch of considerations and decisions.
The underlying fundamental pillars of the resulting outcome as described in this document were a) the concept of Semantic Versioning by Tom Preston-Werner and b) my own set of requirements as described in section *Basic Requirements*.

One may ask: why not adopting Semantic versioning completely ”as-is”? The honest answer is that I was struggeling with the term of ”incompatible API changes” assuming that this does not make much sense in case of such a small console Program as RLOTTO that does not interact with any other kind of other systems or protocols. Some changes to the definition for the major version number were required to fit to specific needs as described in section *Meaningful Version Numbering*.

Basic Requirements

The following set of basic requirements have been formulated for issue # 11 on github:

1. A unique version number
2. A concept (with strictly formalized rules) using components in the version number that reflect major/minor releases and eventually a build or patch number
3. Mapping of version number to github commit/release must be possible
4. Easy maintenance of the version number

Most comprehensive in this enumeration is the second item. Luckily *Semantic Versioning* is fully covering this requirement. However and as already mentioned some adjustments were needed to fulfill the specific demand of software projects with no defined or documented API. This is described in the section after the next.

[1] https://github.com/prozumr/RLOTTO2
[2] https://semver.org/

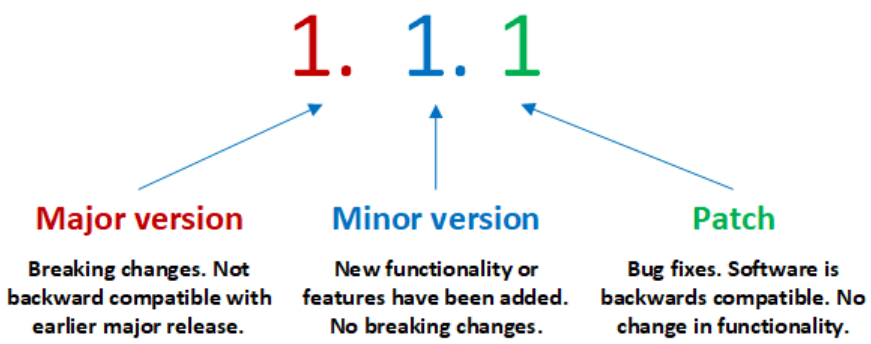
Unique Version Number

Naturally a unique version number is needed but what does it exactly mean? Here are some specifics:

- Any of two different software versions must not have the same version number.
- The version number always applies to all files of the source code. I.e. in a set of source code files all individual files have the same version number.
- The version number is not random but following defined rules for generating the new version number by an increment of the previous version number.
- Usage of a natural number just by incrementing by one (or any other constant increment) for a new version could be misleading even though fulfilling the requirement of uniqueness: tiny minor changes and huge fundamental changes would not be capable of being differntiated just by the version number.

Meaningful Version Numbering

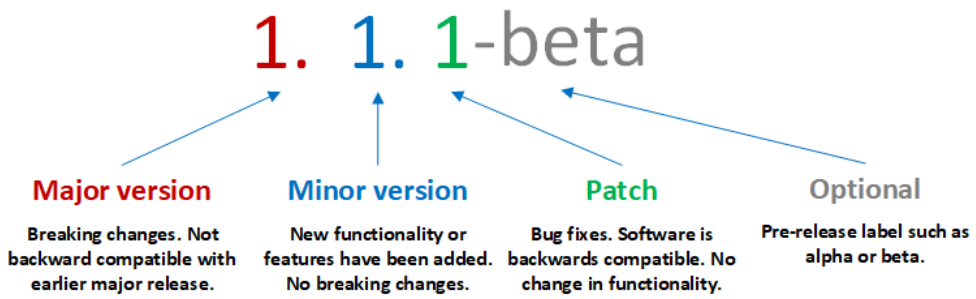
Semantic versioning is based on two elements: Meaningful version numbers and a set of rules and that dictate how to increment these version numbers. Embedding more meaning in more numbers is an obvious concept and used by Semantic Versioning as follows:



Thus three components are used to build the complete version string:

1. MAJOR version for breaking changes not backward compatible to earlier releases
2. MINOR version for adding functionality in a backwards-compatible manner, and
3. PATCH version for backwards-compatible bug fixes.

Additional labels for pre-release are available as extensions to the MAJOR.MINOR.PATCH format:



A more precise definition which of the version number components should be changed and how to increment the number is given in the following section.

Meaningful Version Numbering Specification

As already discussed in the introduction of this document this versioning approach can be called a light-weight, slightly adjusted version of the original Semantic Versioning concept by Tom Preston-Werner. The main difference is that rule No 1 of Semantic Dersioning is neglected ("Software using Semantic Versioning MUST declare a public API."). Instead the conditions when a change of the major version is required is described in a different manner. Please see below for a complete set of rules required for "Meaningful Version Numbering".

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 in RFC 2119[3].

1. Software using Meaningful Versioning Numbering **MUST NOT** declare a public API. This is reflecting the author's belief that it is not possible to rely fully on a deterministic an mechanic approach to cover all scenarios by a set of rules to guarantee the best way of consistent versioning. See also article Why Semantic Versioning Isn't for details on this point of view. Instead pretty precise rules and guideline are given below.

2. 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.

3. 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.

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

[3] https://tools.ietf.org/html/rfc2119

5. Version 1.0.0 defines the public released version - sometimes also called 'Production Release'. The way in which the version number is incremented after this release is dependent on this version as baseline.

6. Patch version Z ($x.y.Z|x > 0$)

   - MUST be incremented if only backwards compatible bug fixes are introduced.
   - A bug fix is defined as an internal change that fixes incorrect behavior.

7. Minor version Y ($x.Y.z|x > 0$)

   - MUST be incremented if new, backwards compatible functionality is introduced that is considered as NOT 'big' enough to justify a change of the major version. Intentionally this leaves room for human interpretation.
   - It MUST be incremented if any public functionality is marked as deprecated.
   - It MAY include patch level changes.
   - Patch version MUST be reset to 0 when minor version is incremented.

8. Major version X ($X.y.z|X > 0$)

   - MUST be incremented if any backwards incompatible functional or non-functional changes are introduced.
   - MUST be incremented if any backwards compatible functional or non functional changes are introduced that are considered to be too 'big' to justify just a minor version change. Intentionally this leaves room for human interpretation.
   - It MAY include minor and patch level changes.
   - Patch and minor version MUST be reset to 0 when major version is incremented.

   A good indicator for a backwards incompatible change is a non-consistent behavior or even a malfunction after a software update. E.g. The updated version cannot read the input data stored in the database or on the file system anymore.

9. A pre-release version MUST be denoted by appending suffix like 'alpha' or 'beta'. 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. The absence of the pre-release suffix indicates a normal version - sometimes also called 'Production Release'. Examples: 1.0.1-alpha, 1.1.2-beta.

Mapping of version number to github commit/release

Git respectively github assigns a unique identifier - called commit id or commit hash value - to any set of source code files by the git commit process. This raises immediately at least two questions:

1. Can we use the commit id as release number?
2. How to map a certain commit/version to a release number?

Apparently the second question is only relevant in case the first one has to be negated. So why not use the commit id as release or version number?

Well, these git commit IDs are not very helpful for the human eye and mind and can be quite confusing. Humans can neither memorize such a forty-character number nor recognize any sequence of version nor getting any other meaning just by the number.

Also there is another even more principal reason why the git commit ID is not a good choice to be a release number: in case the release number should appear in the software itself - i.e. the release number should be visible for users - it needs to be include or 'hardcoded' in the source code before compiling the code. However this is not possible since the commit ID is not provided before the actual commit. In other words you would have to change the source code after the commit which makes the commit ID invalid.

So what is the conclusion? How to map commit ID and release number and how does the entire release number process look like?

This is the proposed release number process as it is used for RLOTTO:

1. Determine the new release number(s) in semantic format MAJOR.MINOR.PATCH-PRE-RELEASE by rules as defined in section *Meaningful Version Numbering Specification*
2. Hardcode your release number inside the source code as described in section *Source Code Implementation*
3. Commit this version of your source code to github
4. Tag the commit on github[4]as new release by using exactly the same release string as hardcoded in the source code

The following and last section addresses the concrete implementation of the release number in the source code.

Source Code Implementation

RLOTTO uses a dedicated header file to maintain version numbers within the source code. The file is called `version.h`. Within `version.h` dedicated variables are defined:

- `MAJOR` to maintain the major component of the version number.
- `MINOR` to maintain the minor component of the version number.
- `PATCH` to maintain the patch component of the version number.
- `STAGE` to maintain the (pre-) release status component of the version number.

The range of values for `STAGE` just includes ”alpha”, ”beta” and ”production”. These four variables are used to build the version number string that appears on the user interface (console window or shell) during start of RLOTTO. In case `STAGE` is not set (empty) or has any other value than ”alpha”, ”beta” or ”production” the program throws a warning message during start: ”Release state undefined”. In case `STAGE` is set to ”production” the release state is not indicated on the user interface bust just shows the version number MAJOR.MINOR.PATCH. In case STAGE is set to a pre-release it is shown on the user interface like MAJOR.MINOR.PATCH-alpha or MAJOR.MINOR.PATCH-beta.

The full version number is generated by a `printf` statement inside the function `welcome()` in `rlotto.c` considering the three different scenarios as described above (wrong or missing value for `STAGE`, pre-release, production release).

The full version number is used once again by a `fprintf` statement inside the function `evaluateTicket(void)` in `evaluate.c` for generating the header of the result file.

In any case the maintenance of the version number and release state is easy and requires just to update the variables `MAJOR`, `MINOR`, `PATCH` and `STAGE` in `version.h`.

[4]See also Git Basics - Tagging