Writing Good FreeBSD Commit Messages

BY ED MASTE

Why are commit messages important?

When you commit a change in Git, Subversion, or another version control system (VCS), you're prompted to write some text describing the commit—a commit message. How important is this commit message? Should you spend some significant effort writing it? Does it really matter if you write simply `fixed a bug`?

Most projects have more than one developer and last for some length of time. Commit messages are a very important method of communicating with other developers in the present and for the future.

FreeBSD has hundreds of active developers and hundreds of thousands of commits spanning decades of history. Over that time the developer community has learned how valuable good commit messages are; sometimes these are hard-learned lessons.

Commit messages serve at least three purposes:

- **Communicating with other developers**
  FreeBSD commits generate email to various mailing lists. These include the commit message along with a copy of the patch itself. Commit messages are also viewed through commands like `git log`. These serve to make other developers aware of changes that are ongoing; that other developer may want to test the change, may have an interest in the topic and will want to review in more detail, or may have their own projects underway that would benefit from interaction.

Commit messages are a very important method of communicating with other developers in the present and for the future

- **Making changes discoverable**
  In a large project with a long history it may be difficult to find changes of interest when investigating an issue or change in behavior. Verbose, detailed commit messages allow searches for changes that might be relevant. For example, `git log --since 1 year --grep 'USB timeout'`.

- **Providing historical documentation**
  Commit messages serve to document changes for future developers, perhaps years or decades later. This future developer may even be you, the original author. A change that seems obvious today may be decidedly not so much later on.
  The `git blame` command annotates each line of a source file with the change (hash and subject line) that brought it in.

Having established the importance, here are elements of a good FreeBSD commit message:

- **Start with a subject line**
  Commit messages should start with a single-line subject that briefly summarizes the change. The subject should, by itself, allow the reader to quickly determine if the change is of interest or not.
- **Keep subject lines short**
  The subject line should be as short as possible while still retaining the required information. This is to make browsing `git log` more efficient, and so that `git log --oneline` can display the short hash and subject on a single 80-column line. A good rule of thumb is to stay below 63 characters and aim for about 50 or fewer if possible.
- **Prefix the subject line with a component, if applicable**
  If the change relates to a specific component the subject line may be prefixed with that component name and a colon (:).
  ✓ `foo: Add -k option to keep temporary data`
  Include the prefix in the 63-character limit suggested above so that `git log --oneline` avoids wrapping.
- **Capitalize the first letter of the subject**
  Capitalize the first letter of the subject itself. The prefix, if any, is not capitalized unless necessary (e.g., `USB:`).
- **Do not end the subject line with punctuation**
  Do not end with a period or other punctuation. In this regard the subject line is like a newspaper headline, or the subject header in an email message.

Commit messages are a very important method of communicating with other developers in the present and for the future

- **Separate the subject and body with a blank line**
  Separate the body from the subject with a blank line.
  Some trivial commits do not require a body and will have only a subject.
  ✓ `ls: Fix typo in usage text`
- **Limit messages to 72 columns**
  `git log` and `git format-patch` indent the commit message by four spaces. Wrapping at 72 columns provides a matching margin on the right edge. Limiting messages to 72 characters also keeps the commit message in formatted patches below RFC 2822's sug-

gested email line length limit of 78 characters. This limit works well with a variety of tools that may render commit messages; line wrapping might be inconsistent with longer line length.

- **Use the present tense, imperative mood**
  This facilitates short subject lines and provides consistency, including with automatically generated commit messages (e.g., as generated by `git revert`). This is important when reading a list of commit subjects. Think of the subject as finishing the sentence "when applied, this change will …".

  ✓ `foo: Implement the -k (keep) option`

  ✗ `foo: Implemented the -k option`

  ✗ `This change implements the -k option in foo`

  ✗ `-k option added`

- **Focus on what and why, not how**
  Explain what the change accomplishes and why it is being done, rather than how.

  Do not assume that the reader is familiar with the issue. Explain the background and motivation for the change. Include benchmark data if you have it.

  If there are limitations or incomplete aspects of the change, describe them in the commit message.

Do not assume that the reader is familiar with the issue. Explain the background and motivation for the change. Include benchmark data if you have it.

- **Consider whether parts of the commit message could be code comments instead**
  Sometimes while writing a commit message you may find yourself writing a sentence or two explaining some tricky or confusing aspect of the change. When this happens consider whether it would be valuable to have that explanation as a comment in the code itself.

- **Write commit messages for your future self**
  While writing the commit message for a change you have all of the context in mind—what prompted the change, alternate approaches that were considered and rejected, limitations of the change, and so on. Imagine yourself revisiting the change a year or two in the future and write the commit message in a way that would provide that necessary context.

- **Commit messages should stand alone**
  You may include references to mailing list postings, benchmark result web sites, or code review links. However, the commit message should contain all of the relevant information in case these references are no longer available in the future.

  Similarly, a commit may refer to a previous commit, for example in the case of a bug fix or revert. In addition to the commit identifier (revision or hash), include the subject line from the referenced commit (or another suitable brief reference). With each VCS migration (from CVS to Subversion to Git) revision identifiers from previous systems may become difficult to follow.

- **Include appropriate metadata in a footer**
  Commit messages may have one or more of a number of standard metadata tags in the final paragraph. Standard tags used in FreeBSD are:

Tag	Description
PR	FreeBSD problem report (Bugzilla) number
Submitted by	ID the original author, if not the committer
Reported by	ID of a third party who reported the issue
Reviewed by	Reviewer ID
Tested by	ID of those who have tested the change
Approved by	Mentor or code owner who approved the change
Obtained from	Source of a change in another project
MFC after	Time period before merging the change from Current to Stable
MFC with	Associated commit that this change should be merged along with
MFH	Ports quarterly branch for merge request
Relnotes	Yes/No whether this change should be included in release notes
Security	External reference for a security issue, such as a CVE number
Sponsored by	Organization or event that sponsored work on the change
Differential Revision	Full URL of code review in FreeBSD's Phabricator instance

"ID" indicates either a FreeBSD userid or a name and email address. Multiple IDs may be presented as a comma-separated list or by repeating metadata tags on subsequent lines.

ED MASTE manages project development for the FreeBSD Foundation. He is also a member of the elected FreeBSD Core Team. Aside from FreeBSD, he has contributed to a number of other open-source projects, including LLVM, ELF Tool Chain, QEMU, and Open vSwitch. He lives in Kitchener-Waterloo, Canada, with his wife, Anna, and sons, Pieter and Daniel.