Git SOP

HWTE EEIT

Version: 1.1
Update Date: 2021/9/28

Contents

1. Introduction

SMT station SW development involves code contribution from Apple internal teams, engineering service individuals and external vendor/CM teams. Git operation is intensive daily work for SW developers. It easily leads to misused branches, commits, pull requests etc which adds tremendous effort and difficulty in code review process and jeopardize SW quality. The goal of this SOP is to regulate all SMT SW developers' git operation practice and align with Synergy.

1.1 Scope

This SOP is applicable to any source code to be used on SMT stations submitted by Apple and external teams (include vendor, CM and ES)

1.2 Audience

HWTE, Engineering Services, Disclosed Vendors, Disclosed Apple Contract Manufacturer teams.

1.3 Knowledge Preparation

Code contributor shall already obtain git operation knowledge before reading this document. Refer to https://git-scm.com/doc to get familiar with git before moving on.

1.4 Release

rdar://83000060 (Git SOP doc external release)

2. Branch

2.1 Understand branch protection rules

Master branch contains the most recent stable code for production release only. No direct commits are allowed on master except for Apple DRI to perform mink tag.

Any code changes must be finally landed on master through pull request and review. No dangling branches are allowed except for below cases.

- This is a hot fix branch from an old tag.
- It's a temp DOE branch which did not make it to master. It's still recommended to delete this branch if it's not planned to be used later.

2.2 Prepare a branch

Before making any code changes, developer should prepare a dedicated branch for the feature/fix. Do NOT perform more than one change purpose on one branch, e.g. fix a bug as well as do some code reformatting.

If an open branch exists already and exactly for the purpose, just reuse the existing one. Do NOT continue to use a closed branch which has already been merged once to master but not deleted, aka. dangling branch.

If no existing branch fits your purpose, you need to create a new branch following the rules listed below.

- All branches need to be created based off head of master except for some hot fix
- Hot fix can be created on an old tag on master or head of master. If no tag exists, developer need to add a tag first. Detached head based branch is NOT allowed.
- More than one open branches on the same tag or head of master is allowed, but will be more difficult to resolve conflicts or rebase when merge, so it’s recommended to keep a limited number of concurrent branches.
- Refrain from creating branch off another branch wherever possible. Do NOT merge a branch off another branch back to master before the parent branch is merged.

2.3 Naming branch

When you create a new branch, use a prefix plus short feature name as branch name. The following prefixes are recommended:

- Your user name: jsmith/HDR_Capture
- feature for new feature branches: feature/HDR_Capture
- bugfix for fixes: bugfix/crash_on_load

For features/bugs tracked through the radar, the corresponding radar ID can be used after the prefix instead of text description: feature/radar-123456 or feature/PR-123456

Use tag name and a meaningful suffix to name the hot fix branch created from an existing tag. E.g. the branch created from tag 1.3.12 to fix a memory leak can be called bugfix/1.3.12_memleak_fix

3. Commit

3.1 General rules

Changes made to the source code MUST be pushed to the remote server on a regular basis. Avoid keeping changes on your local machine for too long. Do NOT commit code change which is syntax incomplete.

Keep each commit for ONLY one purpose.
Keep each commit as less lines of change as possible.

Ensure .gitignore file exists on your branch with proper file type filter to filter out undesired files to be committed.

3.2 Sanity check

Before commit a code change, developer needs to finish two sanity checks as below.

1. Format check

Run format check tool provided by Apple to sanitize the code formats of both Lua and C code. The same check is performed by Apple CI server as a pass criteria. Failed to pass the format check will block the pull request to be merged.

2. Feature/Fix functional validation

Developer needs to validate the change before commit to ensure proper functionality and SW behave as design expectation.

3.3 Commit Message

Git commit messages should be concise, and clearly explain the change being made in a commit. And most importantly, commit message should explain why the change is made.

In general, it's strongly recommended to always have a radar for any change made to the code. The radar can provide better context than even a set of detailed commit messages, since it contains discussions, questions and other information that might explain why a specific change was made, and what alternatives were considered.

When radar is available for a code change, the radar URL with title should be added at the beginning of the commit message.

3.4 The 7 rules of a good Git commit message

1. Separate subject line from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

Not every commit requires both a subject and a body. Sometimes a single line is fine, especially when the change is so simple that no further context is necessary. However body is suggested. Example:

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

4. Pull Request

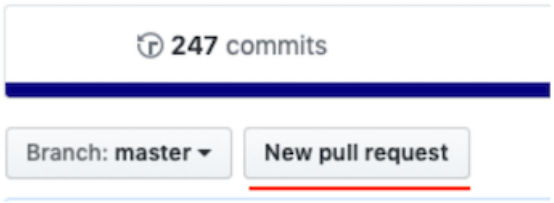
4.1 General rules

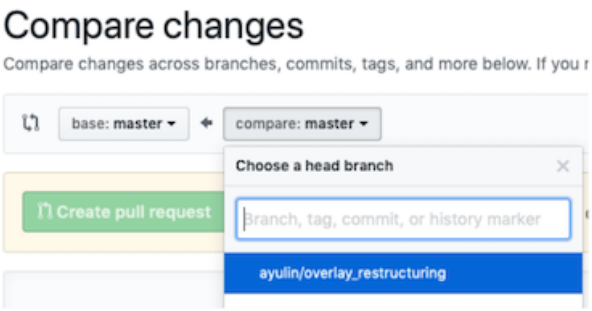
Before any code change can land on master, it has to pass code review through pull request since master is protected from direct commits.

Pull request should be created by SW developer who's been working on the branch.
Before creating a new pull request, rebase the branch to master head.

4.2 Create pull request

On GitHub web page, select New pull request and select your branch.
Example:





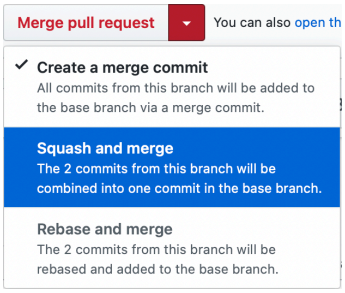
Fill in pull request title line and body comment and click Create pull request. Refer to #3.3 as the 7 rules for commit message also applies to pull request title and body.

When pull request is created, Github should auto assign Code Owner as default reviewer if CODEOWNERS are setup correctly per Source Control SOP by Apple DRI. More code reviewers can be added by project demand.

4.3 Review handling and merge

All comments and change requests must be fully addressed to receive approval.
Developer should respond to comments on GitHub for traceability. Files/images uploading are supported in comments.

When required approval is received, Apple DRI with admin privilege should merge the pull request. It's encouraged to use Squash Merge over merge commit as optional.



Branch should be deleted immediately after pull request is merged.

5. Version History

Date	Version	Comment	Author
2021/9/17	1.0	Initial Release.	Martin Zhang
2021/9/28	1.1	Add merge type suggestion n 4.3	Martin Zhang