# Rebasing and merging

Maintaining a subsystem, as a general rule, requires a familiarity with the Git source-code management system. Git is a powerful tool with a lot of features; as is often the case with such tools, there are right and wrong ways to use those features. This document looks in particular at the use of rebasing and merging. Maintainers often get in trouble when they use those tools incorrectly, but avoiding problems is not actually all that hard.

One thing to be aware of in general is that, unlike many other projects, the kernel community is not scared by seeing merge commits in its development history. Indeed, given the scale of the project, avoiding merges would be nearly impossible. Some problems encountered by maintainers result from a desire to avoid merges, while others come from merging a little too often.

## Rebasing

"Rebasing" is the process of changing the history of a series of commits within a repository. There are two different types of operations that are referred to as rebasing since both are done with the `git rebase` command, but there are significant differences between them:

- Changing the parent (starting) commit upon which a series of patches is built. For example, a rebase operation could take a patch set built on the previous kernel release and base it, instead, on the current release. We'll call this operation "reparenting" in the discussion below.
- Changing the history of a set of patches by fixing (or deleting) broken commits, adding patches, adding tags to commit changelogs, or changing the order in which commits are applied. In the following text, this type of operation will be referred to as "history modification"

The term "rebasing" will be used to refer to both of the above operations. Used properly, rebasing can yield a cleaner and clearer development history; used improperly, it can obscure that history and introduce bugs.

There are a few rules of thumb that can help developers to avoid the worst perils of rebasing:

- History that has been exposed to the world beyond your private system should usually not be changed. Others may have pulled a copy of your tree and built on it; modifying your tree will create pain for them. If work is in need of rebasing, that is usually a sign that it is not yet ready to be committed to a public repository.

  That said, there are always exceptions. Some trees (linux-next being a significant example) are frequently rebased by their nature, and developers know not to base work on them. Developers will sometimes expose an unstable branch for others to test with or for automated testing services. If you do expose a branch that may be unstable in this way, be sure that prospective users know not to base work on it.
- Do not rebase a branch that contains history created by others. If you have pulled changes from another developer's repository, you are now a custodian of their history. You should not change it. With few exceptions, for example, a broken commit in a tree like this should be explicitly reverted rather than disappeared via history modification.
- Do not reparent a tree without a good reason to do so. Just being on a newer base or avoiding a merge with an upstream repository is not generally a good reason.
- If you must reparent a repository, do not pick some random kernel commit as the new base. The kernel is often in a relatively unstable state between release points; basing development on one of those points increases the chances of running into surprising bugs. When a patch series must move to a new base, pick a stable point (such as one of the -rc releases) to move to.
- Realize that reparenting a patch series (or making significant history modifications) changes the environment in which it was developed and, likely, invalidates much of the testing that was done. A reparented patch series should, as a general rule, be treated like new code and retested from the beginning.

A frequent cause of merge-window trouble is when Linus is presented with a patch series that has clearly been reparented, often to a random commit, shortly before the pull request was sent. The chances of such a series having been adequately tested are relatively low - as are the chances of the pull request being acted upon.

If, instead, rebasing is limited to private trees, commits are based on a well-known starting point, and they are well tested, the potential for trouble is low.

## Merging

Merging is a common operation in the kernel development process; the 5.1 development cycle included 1,126 merge commits - nearly 9% of the total. Kernel work is accumulated in over 100 different subsystem trees, each of which may contain multiple topic branches; each branch is usually developed independently of the others. So naturally, at least one merge will be required before any given branch finds its way into an upstream repository.

Many projects require that branches in pull requests be based on the current trunk so that no merge commits appear in the history.

The kernel is not such a project; any rebasing of branches to avoid merges will, most likely, lead to trouble.

Subsystem maintainers find themselves having to do two types of merges: from lower-level subsystem trees and from others, either sibling trees or the mainline. The best practices to follow differ in those two situations.

### Merging from lower-level trees

Larger subsystems tend to have multiple levels of maintainers, with the lower-level maintainers sending pull requests to the higher levels. Acting on such a pull request will almost certainly generate a merge commit; that is as it should be. In fact, subsystem maintainers may want to use the --no-ff flag to force the addition of a merge commit in the rare cases where one would not normally be created so that the reasons for the merge can be recorded. The changelog for the merge should, for any kind of merge, say *why* the merge is being done. For a lower-level tree, "why" is usually a summary of the changes that will come with that pull.

Maintainers at all levels should be using signed tags on their pull requests, and upstream maintainers should verify the tags when pulling branches. Failure to do so threatens the security of the development process as a whole.

As per the rules outlined above, once you have merged somebody else's history into your tree, you cannot rebase that branch, even if you otherwise would be able to.

### Merging from sibling or upstream trees

While merges from downstream are common and unremarkable, merges from other trees tend to be a red flag when it comes time to push a branch upstream. Such merges need to be carefully thought about and well justified, or there's a good chance that a subsequent pull request will be rejected.

It is natural to want to merge the master branch into a repository; this type of merge is often called a "back merge". Back merges can help to make sure that there are no conflicts with parallel development and generally gives a warm, fuzzy feeling of being up-to-date. But this temptation should be avoided almost all of the time.

Why is that? Back merges will muddy the development history of your own branch. They will significantly increase your chances of encountering bugs from elsewhere in the community and make it hard to ensure that the work you are managing is stable and ready for upstream. Frequent merges can also obscure problems with the development process in your tree; they can hide interactions with other trees that should not be happening (often) in a well-managed branch.

That said, back merges are occasionally required; when that happens, be sure to document *why* it was required in the commit message. As always, merge to a well-known stable point, rather than to some random commit. Even then, you should not back merge a tree above your immediate upstream tree; if a higher-level back merge is really required, the upstream tree should do it first.

One of the most frequent causes of merge-related trouble is when a maintainer merges with the upstream in order to resolve merge conflicts before sending a pull request. Again, this temptation is easy enough to understand, but it should absolutely be avoided. This is especially true for the final pull request: Linus is adamant that he would much rather see merge conflicts than unnecessary back merges. Seeing the conflicts lets him know where potential problem areas are. He does a lot of merges (382 in the 5.1 development cycle) and has gotten quite good at conflict resolution - often better than the developers involved.

So what should a maintainer do when there is a conflict between their subsystem branch and the mainline? The most important step is to warn Linus in the pull request that the conflict will happen; if nothing else, that demonstrates an awareness of how your branch fits into the whole. For especially difficult conflicts, create and push a *separate* branch to show how you would resolve things. Mention that branch in your pull request, but the pull request itself should be for the unmerged branch.

Even in the absence of known conflicts, doing a test merge before sending a pull request is a good idea. It may alert you to problems that you somehow didn't see from linux-next and helps to understand exactly what you are asking upstream to do.

Another reason for doing merges of upstream or another subsystem tree is to resolve dependencies. These dependency issues do happen at times, and sometimes a cross-merge with another tree is the best way to resolve them; as always, in such situations, the merge commit should explain why the merge has been done. Take a moment to do it right; people will read those changelogs.

Often, though, dependency issues indicate that a change of approach is needed. Merging another subsystem tree to resolve a dependency risks bringing in other bugs and should almost never be done. If that subsystem tree fails to be pulled upstream, whatever problems it had will block the merging of your tree as well. Preferable alternatives include agreeing with the maintainer to carry both sets of changes in one of the trees or creating a topic branch dedicated to the prerequisite commits that can be merged into both trees. If the dependency is related to major infrastructural changes, the right solution might be to hold the dependent commits for one development cycle so that those changes have time to stabilize in the mainline.

## Finally

It is relatively common to merge with the mainline toward the beginning of the development cycle in order to pick up changes and fixes done elsewhere in the tree. As always, such a merge should pick a well-known release point rather than some random spot. If your upstream-bound branch has emptied entirely into the mainline during the merge window, you can pull it forward with a command like:

```
git merge v5.2-rc1^0
```

The "^0" will cause Git to do a fast-forward merge (which should be possible in this situation), thus avoiding the addition of a spurious

merge commit.

The guidelines laid out above are just that: guidelines. There will always be situations that call out for a different solution, and these guidelines should not prevent developers from doing the right thing when the need arises. But one should always think about whether the need has truly arisen and be prepared to explain why something abnormal needs to be done.