Contributing to Bitcoin Core

The Bitcoin Core project operates an open contributor model where anyone is welcome to contribute towards development in the form of peer review, testing and patches. This document explains the practical process and guidelines for contributing.

First, in terms of structure, there is no particular concept of "Bitcoin Core developers" in the sense of privileged people. Open source often naturally revolves around a meritocracy where contributors earn trust from the developer community over time. Nevertheless, some hierarchy is necessary for practical purposes. As such, there are repository "maintainers" who are responsible for merging pull requests, as well as a "lead maintainer" who is responsible for the release cycle as well as overall merging, moderation and appointment of maintainers.

Getting Started

New contributors are very welcome and needed.

Reviewing and testing is highly valued and the most effective way you can contribute as a new contributor. It also will teach you much more about the code and process than opening pull requests. Please refer to the peer review section below.

Before you start contributing, familiarize yourself with the Bitcoin Core build system and tests. Refer to the documentation in the repository on how to build Bitcoin Core and how to run the unit tests, functional tests, and fuzz tests.

There are many open issues of varying difficulty waiting to be fixed. If you're looking for somewhere to start contributing, check out the good first issue list or changes that are up for grabs. Some of them might no longer be applicable. So if you are interested, but unsure, you might want to leave a comment on the issue first.

You may also participate in the weekly Bitcoin Core PR Review Club meeting.

Good First Issue Label

The purpose of the `good first issue` label is to highlight which issues are suitable for a new contributor without a deep understanding of the codebase.

However, good first issues can be solved by anyone. If they remain unsolved for a longer time, a frequent contributor might address them.

You do not need to request permission to start working on an issue. However, you are encouraged to leave a comment if you are planning to work on it. This will help other contributors monitor which issues are actively being addressed and is also an effective way to request assistance if and when you need it.

Communication Channels

Most communication about Bitcoin Core development happens on IRC, in the `#bitcoin-core-dev` channel on Libera Chat. The easiest way to participate on IRC is with the web client, web.libera.chat. Chat history logs can be found on https://www.erisian.com.au/bitcoin-core-dev/ and https://gnusha.org/bitcoin-core-dev/.

Discussion about codebase improvements happens in GitHub issues and pull requests.

The developer mailing list should be used to discuss complicated or controversial consensus or P2P protocol changes before working on a patch set.

Contributor Workflow

The codebase is maintained using the "contributor workflow" where everyone without exception contributes patch proposals using "pull requests" (PRs). This facilitates social contribution, easy testing and peer review.

To contribute a patch, the workflow is as follows:

1. Fork repository (only for the first time)
2. Create topic branch
3. Commit patches

For GUI-related issues or pull requests, the https://github.com/bitcoin-core/gui repository should be used. For all other issues and pull requests, the https://github.com/bitcoin/bitcoin node repository should be used.

The master branch for all monotree repositories is identical.

As a rule of thumb, everything that only modifies `src/qt` is a GUI-only pull request. However:

- For global refactoring or other transversal changes the node repository should be used.
- For GUI-related build system changes, the node repository should be used because the change needs review by the build systems reviewers.
- Changes in `src/interfaces` need to go to the node repository because they might affect other components like the wallet.

For large GUI changes that include build system and interface changes, it is recommended to first open a pull request against the GUI repository. When there is agreement to proceed with the changes, a pull request with the build system and interfaces changes can be submitted to the node repository.

The project coding conventions in the developer notes must be followed.

Committing Patches

In general, commits should be atomic and diffs should be easy to read. For this reason, do not mix any formatting fixes or code moves with actual code changes.

Make sure each individual commit is hygienic: that it builds successfully on its own without warnings, errors, regressions, or test failures.

Commit messages should be verbose by default consisting of a short subject line (50 chars max), a blank line and detailed explanatory text as separate paragraph(s), unless the title alone is self-explanatory (like "Correct typo in init.cpp") in which case a single title line is sufficient. Commit messages should be helpful to people reading your code in the future, so explain the reasoning for your decisions. Further explanation here.

If a particular commit references another issue, please add the reference. For example: `refs #1234` or `fixes #4321`. Using the `fixes` or `closes` keywords will cause the corresponding issue to be closed when the pull request is merged.

Commit messages should never contain any `@` mentions (usernames prefixed with "@").

Please refer to the Git manual for more information about Git.

- Push changes to your fork
- Create pull request

Creating the Pull Request

The title of the pull request should be prefixed by the component or area that the pull request affects. Valid areas as:

- `consensus` for changes to consensus critical code
- `doc` for changes to the documentation
- `qt` or `gui` for changes to bitcoin-qt
- `log` for changes to log messages
- `mining` for changes to the mining code
- `net` or `p2p` for changes to the peer-to-peer network code
- `refactor` for structural changes that do not change behavior
- `rpc`, `rest` or `zmq` for changes to the RPC, REST or ZMQ APIs
- `script` for changes to the scripts and tools
- `test`, `qa` or `ci` for changes to the unit tests, QA tests or CI code
- `util` or `lib` for changes to the utils or libraries
- `wallet` for changes to the wallet code
- `build` for changes to the GNU Autotools or MSVC builds
- `guix` for changes to the GUIX reproducible builds

Examples:

```
consensus: Add new opcode for BIP-XXXX OP_CHECKAWESOMESIG
net: Automatically create onion service, listen on Tor
qt: Add feed bump button
log: Fix typo in log message
```

The body of the pull request should contain sufficient description of *what* the patch does, and even more importantly, *why*, with justification and reasoning. You should include references to any discussions (for example, other issues or mailing list discussions).

The description for a new pull request should not contain any `@` mentions. The PR description will be included in the commit message when the PR is merged and any users mentioned in the description will be annoyingly notified each time a fork of Bitcoin Core copies the merge. Instead, make any username mentions in a subsequent comment to the PR.

Translation changes

Note that translations should not be submitted as pull requests. Please see Translation Process for more information on helping with translations.

Work in Progress Changes and Requests for Comments

If a pull request is not to be considered for merging (yet), please prefix the title with [WIP] or use Tasks Lists in the body of the pull request to indicate tasks are pending.

Address Feedback

At this stage, one should expect comments and review from other contributors. You can add more commits to your pull request by committing them locally and pushing to your fork.

You are expected to reply to any review comments before your pull request is merged. You may update the code or reject the feedback if you do not agree with it, but you should express so in a reply. If there is outstanding feedback and you are not actively working on it, your pull request may be closed.

Please refer to the peer review section below for more details.

Squashing Commits

If your pull request contains fixup commits (commits that change the same line of code repeatedly) or too fine-grained commits, you may be asked to squash your commits before it will be reviewed. The basic squashing workflow is shown below.

```
git checkout your_branch_name
git rebase -i HEAD~n
# n is normally the number of commits in the pull request.
# Set commits (except the one in the first line) from 'pick' to 'squash', save and
quit.
# On the next screen, edit/refine commit messages.
# Save and quit.
git push -f # (force push to GitHub)
```

Please update the resulting commit message, if needed. It should read as a coherent message. In most cases, this means not just listing the interim commits.

If you have problems with squashing or other git workflows, you can enable "Allow edits from maintainers" in the right-hand sidebar of the GitHub web interface and ask for help in the pull request.

Please refrain from creating several pull requests for the same change. Use the pull request that is already open (or was created earlier) to amend changes. This preserves the discussion and review that happened earlier for the respective change set.

The length of time required for peer review is unpredictable and will vary from pull request to pull request.

Rebasing Changes

When a pull request conflicts with the target branch, you may be asked to rebase it on top of the current target branch. The `git rebase` command will take care of rebuilding your commits on top of the new base.

This project aims to have a clean git history, where code changes are only made in non-merge commits. This simplifies auditability because merge commits can be assumed to not contain arbitrary code changes. Merge commits should be signed, and the resulting git tree hash must be deterministic and reproducible. The script in /contrib/verify-commits checks that.

After a rebase, reviewers are encouraged to sign off on the force push. This should be relatively straightforward with the `git range-diff` tool explained in the productivity notes. To avoid needless review churn, maintainers will generally merge pull requests that received the most review attention first.

Pull Request Philosophy

Patchsets should always be focused. For example, a pull request could add a feature, fix a bug, or refactor code; but not a mixture. Please also avoid super pull requests which attempt to do too much, are overly large, or overly complex as this makes review difficult.

Features

When adding a new feature, thought must be given to the long term technical debt and maintenance that feature may require after inclusion. Before proposing a new feature that will require maintenance, please consider if you are willing to maintain it (including bug fixing). If features get orphaned with no maintainer in the future, they may be removed by the Repository Maintainer.

Refactoring

Refactoring is a necessary part of any software project's evolution. The following guidelines cover refactoring pull requests for the project.

There are three categories of refactoring: code-only moves, code style fixes, and code refactoring. In general, refactoring pull requests should not mix these three kinds of activities in order to make refactoring pull requests easy to review and uncontroversial. In all cases, refactoring PRs must not change the behaviour of code within the pull request (bugs must be preserved as is).

Project maintainers aim for a quick turnaround on refactoring pull requests, so where possible keep them short, uncomplex and easy to verify.

Pull requests that refactor the code should not be made by new contributors. It requires a certain level of experience to know where the code belongs to and to understand the full ramification (including rebase effort of open pull requests).

Trivial pull requests or pull requests that refactor the code with no clear benefits may be immediately closed by the maintainers to reduce unnecessary workload on reviewing.

"Decision Making" Process

The following applies to code changes to the Bitcoin Core project (and related projects such as libsecp256k1), and is not to be confused with overall Bitcoin Network Protocol consensus changes.

Whether a pull request is merged into Bitcoin Core rests with the project merge maintainers and ultimately the project lead.

Maintainers will take into consideration if a patch is in line with the general principles of the project; meets the minimum standards for inclusion; and will judge the general consensus of contributors.

In general, all pull requests must:

- Have a clear use case, fix a demonstrable bug or serve the greater good of the project (for example refactoring for modularisation);
- Be well peer-reviewed;
- Have unit tests, functional tests, and fuzz tests, where appropriate;
- Follow code style guidelines (C++, functional tests);
- Not break the existing test suite;
- Where bugs are fixed, where possible, there should be unit tests demonstrating the bug and also proving the fix. This helps prevent regression.
- Change relevant comments and documentation when behaviour of code changes.

Patches that change Bitcoin consensus rules are considerably more involved than normal because they affect the entire ecosystem and so must be preceded by extensive mailing list discussions and have a numbered BIP. While each case will be different, one should be prepared to expend more time and effort than for other kinds of patches because of increased peer review and consensus building requirements.

Peer Review

Anyone may participate in peer review which is expressed by comments in the pull request. Typically reviewers will review the code for obvious errors, as well as test out the patch set and opine on the technical merits of the patch. Project maintainers take into account the peer review when determining if there is consensus to merge a pull request (remember that discussions may have been spread out over GitHub, mailing list and IRC discussions).

Code review is a burdensome but important part of the development process, and as such, certain types of pull requests are rejected. In general, if the **improvements** do not warrant the **review effort** required, the PR has a high chance of being rejected. It is up to the PR author to convince the reviewers that the changes warrant the review effort, and if reviewers are "Concept NACK'ing" the PR, the author may need to present arguments and/or do research backing their suggested changes.

Conceptual Review

A review can be a conceptual review, where the reviewer leaves a comment

- `Concept (N)ACK`, meaning "I do (not) agree with the general goal of this pull request",
- `Approach (N)ACK`, meaning `Concept ACK`, but "I do (not) agree with the approach of this change".

A `NACK` needs to include a rationale why the change is not worthwhile. NACKs without accompanying reasoning may be disregarded.

Code Review

After conceptual agreement on the change, code review can be provided. A review begins with `ACK BRANCH_COMMIT`, where `BRANCH_COMMIT` is the top of the PR branch, followed by a description of how the reviewer did the review. The following language is used within pull request comments:

- "I have tested the code", involving change-specific manual testing in addition to running the unit, functional, or fuzz tests, and in case it is not obvious how the manual testing was done, it should be described;
- "I have not tested the code, but I have reviewed it and it looks OK, I agree it can be merged";
- A "nit" refers to a trivial, often non-blocking issue.

Project maintainers reserve the right to weigh the opinions of peer reviewers using common sense judgement and may also weigh based on merit. Reviewers that have demonstrated a deeper commitment and understanding of the project over time or who have clear domain expertise may naturally have more weight, as one would expect in all walks of life.

Where a patch set affects consensus-critical code, the bar will be much higher in terms of discussion and peer review requirements, keeping in mind that mistakes could be very costly to the wider community. This includes refactoring of consensus-critical code.

Where a patch set proposes to change the Bitcoin consensus, it must have been discussed extensively on the mailing list and IRC, be accompanied by a widely discussed BIP and have a generally widely perceived technical consensus of being a worthwhile change based on the judgement of the maintainers.

Finding Reviewers

As most reviewers are themselves developers with their own projects, the review process can be quite lengthy, and some amount of patience is required. If you find that you've been waiting for a pull request to be given attention for several months, there may be a number of reasons for this, some of which you can do something about:

- It may be because of a feature freeze due to an upcoming release. During this time, only bug fixes are taken into consideration. If your pull request is a new feature, it will not be prioritized until after the release. Wait for the release.
- It may be because the changes you are suggesting do not appeal to people. Rather than nits and critique, which require effort and means they care enough to spend time on your contribution, thundering silence is a good sign of widespread (mild) dislike of a given change (because people don't assume *others* won't actually like the proposal). Don't take that personally, though! Instead, take another critical look at what you are suggesting and see if it: changes too much, is too broad, doesn't adhere to the developer notes, is

dangerous or insecure, is messily written, etc. Identify and address any of the issues you find. Then ask e.g. on IRC if someone could give their opinion on the concept itself.

- It may be because your code is too complex for all but a few people, and those people may not have realized your pull request even exists. A great way to find people who are qualified and care about the code you are touching is the Git Blame feature. Simply look up who last modified the code you are changing and see if you can find them and give them a nudge. Don't be incessant about the nudging, though.
- Finally, if all else fails, ask on IRC or elsewhere for someone to give your pull request a look. If you think you've been waiting for an unreasonably long time (say, more than a month) for no particular reason (a few lines changed, etc.), this is totally fine. Try to return the favor when someone else is asking for feedback on their code, and the universe balances out.
- Remember that the best thing you can do while waiting is give review to others!

Backporting

Security and bug fixes can be backported from `master` to release branches. If the backport is non-trivial, it may be appropriate to open an additional PR to backport the change, but only after the original PR has been merged. Otherwise, backports will be done in batches and the maintainers will use the proper `Needs backport (...)` labels when needed (the original author does not need to worry about it).

A backport should contain the following metadata in the commit body:

```
Github-Pull: #<PR number>
Rebased-From: <commit hash of the original commit>
```

Have a look at an example backport PR.

Also see the backport.py script.

Copyright

By contributing to this repository, you agree to license your work under the MIT license unless specified otherwise in `contrib/debian/copyright` or at the top of the file itself. Any work contributed where you are not the original author must contain its license header with the original author(s) and source.