Jimmy Song Follow
Bitcoin Developer and Entrepreneur
Jul 31 · 9 min read

A Gentle Introduction to Bitcoin Core Development

If you're a developer and you own any Bitcoin at all, contributing to Bitcoin Core can be one of the best things you can do to help out your investment. In this article, I'm going to give a gentle, step-by-step overview of how to contribute to Core development.

So You Want to Be a Core Developer...

Before we get into the actual nitty-gritty of contributing to Core, there are a few caveats to get out of the way.

First and foremost, Bitcoin Core is a meritocracy. As a noob, you are unlikely to get a crazy proof-of-work change pull request merged into Core. Like any meritocracy, you start with zero reputation and work your way up.

The good news is that nobody cares about your background. Whether you're a 14-year-old in India or a 45-year-old CTO of a Fortune-500 company, the only thing that really matters is the quality of your work.

The bad news is that you have to be willing to leave your ego at the door. Nobody cares how many years of experience you have or what great ideas you have for fixing Bitcoin. The quality of your code, reviews, documentation and testing are what count.

Second is that you should set your expectations accordingly. Notable Core developers like Pieter Wuille, Cory Fields and Gregory Maxwell have earned their respect through many *years* of blood, sweat and tears. Adding a PR that corrects a spelling error is not going to win you a Pieter Wuille-level of respect. Good work will earn you recognition and respect, but only after you've been producing for a while.

Third is that this is not an easy road. Being the top developer at company X will not necessarily make you a good Core developer. There's a variety of reasons for this, but in general, your work has to meet some

pretty high standards of testing, documentation and code review not often seen in even very competent technical companies.

All that said, if you have the humility, drive and desire for excellence, contributing to Core will make you a better developer, code reviewer, documenter and tester.

Prerequisites

You will need some knowledge and skills to begin.

Bitcoin Core is mainly in two languages, C++ and Python. You're most likely going to have to learn at least some of both if you hope to contribute.

Source control is managed through Git. At a minimum, you should know how to fetch from origin, how to make topic branches and how to rebase. If you're testing someone else's code, you should also know how to add multiple repositories to your local development environment so you can fetch and test the code.

All changes to Core are merged on a PR by PR basis on Github, so you will need a GitHub account.

Lastly, you'll need to know how to install and remove packages on your platform. The instructions are pretty thorough, but it helps, for instance, to be able to add and remove ZMQ as needed.

Starting Out

The first thing that you'll want to do when starting out is to read some documentation. The README and the contribution guidelines are good places to start.

Afterward, head over to the doc directory and read the README there. All the documentation in the doc directory is described in the README, so you can refer back to it if you get lost or confused at any point.

Note that you don't have to understand everything in every document I've suggested. There are a lot of very nice people that can help you on

IRC, StackExchange and Slack if you encounter something that you don't understand.

Building From Source

Now that you've read the basics of how development is supposed to work, start by building bitcoin from source. First, clone the bitcoin Git repository:

```
git clone git@github.com:bitcoin/bitcoin
```

Next step will be to set up your development environment. This will depend heavily on the platform you're on, but compiling is something you'll need to do frequently so it's well worth getting this part down.

Additionally, you'll want to run all the integration tests, so be sure to turn on the GUI and ZMQ when going through the instructions below.

- *nix instructions are here.
- osx instructions are here.
- windows instructions are here.
- openBSD instructions are here.

As you set up your environment, if something doesn't work, Google the errors first before submitting a documentation PR. As mentioned before, IRC, StackExchange and Slack are good resources, but please don't be that guy that asks really simple questions and drains everyone's time.

Testing

Now that you've compiled everything, the next step is to test out the software. Thankfully, Bitcoin Core has a variety of unit and integration tests to check that the software you just compiled works properly.

First, run the unit tests located here. Unit tests are compiled along with everything else, so all you need to do is run the binary that results.

Check that all the tests pass. If they don't there's probably some instruction that was missed.

If all your unit tests pass, run the integration tests located here. You'll want to run the extended tests. The pruning test in particular takes a long time to run, so you will want to exclude that test when running integration tests in the future.

Again, check that all the tests pass. You will see a lot of dots until a summary shows up at the end. If something doesn't pass, you probably missed some instruction, though at times, some integration tests can be a bit fickle based on RAM and CPU.

Contribution Categories

Now that you have your system set up, you can begin to contribute!

You may think that contribution means adding a bunch of code, sending in a PR and getting the glory. In actuality, a lot of the work is around the reviewing and testing code that other people submit. It helps to understand the steps involved in how a PR gets merged.

1. Someone creates a change and submits the code via a Pull Request (PR).
2. One or more people review the code.
3. One or more people test the code.
4. When enough people have reviewed and tested the code, a maintainer will merge the code—only a handful of people can do this.

Most people think of contributing to an open source project as only contributing code, but in reality, testing and review are more important to the success of the project. Lack of review and testing is often the reason for security flaws in many projects as we saw in the recent Ethereum Parity Bug.

Review and testing are also critical as it's often hard to have an adversarial mindset that covers all cases. Having many eyes look and test the code helps to uncover possible ways in which the code can be exploited.

Finally, not only are reviewing and testing good for the project, but they're good for you! Reviewing and testing will force you to learn more about the code base and will give you a much deeper understanding than even writing code will do.

Starting Out

To get used to the contribution process, you should *not* start by adding lots of PRs. Instead, as a new developer with no reputation, your best bet is to start off reviewing and testing other people's work. Reviewing and testing tend to be bottlenecks, so you can contribute and earn some reputation in the community at the same time.

It's worth mentioning here that part of the reason why Greg Maxwell has such a great reputation among developers is that he is a really good reviewer and tester. He has world-class talent when it comes to finding possible ways something can break, and he reviews and tests a lot more code than writes. I promise you will appreciate him even more after you've reviewed and tested some code yourself.

Additionally, code is generally written once, but it's read a ton of times. Review is therefore a much more important step as it reveals how "readable" the code is. The general rule of thumb is that for every pull request you do, you should be reviewing about 3 pull requests. At the beginning you may want this ratio of reviews to pull requests to be even higher.

How To Review

Generally, you will not be able to give a good review of a Pull Request until you understand what the code is doing. As they say in coding, writing is a lot easier than reading, so take some time to really understand the code.

Oftentimes, to get a grasp of what is going on in a PR, you will have to look up the functions and methods that are being used and carefully examine the context of the code. If you get confused and the person whose code you're reviewing is on IRC, you may want to ask questions directly. Most of the time, PR authors are very appreciative of reviewers and will be happy to help you.

Much like reviewing an article or a book, there are a lot of things that you should be looking out for, namely:

- Is the code doing what it should be doing?
- Is the code tested sufficiently?
- Are all the comments around the code helpful and accurate?
- Is the code clear and easy to modify later?

As a general rule, if you disagree with how something is done, it is better to assume you don't understand what's going on rather than launching into a lecture. Empathy and tact will get you a long way here. Remember, you are in many ways commenting on someone's baby. Ask questions and be gentle, at least at first. Oftentimes, these coders don't know you or your intentions. Be very clear when something is a minor issue of style vs. a major issue that may break something. It's much better to sound like a student trying to get a grasp of what the improvement is than a gatekeeper of sorts.

Once you are sure what it is the code is trying to achieve, then you can comment on whether this is a feature worth doing. Until you've built up some reputation, stay away from any comments that could be perceived negatively.

When you are done reviewing the code, comment on the PR with the appropriate peer review comment. If you want to NACK, start over and assume you don't understand what's going on, converse with the author and ask questions until you're positive this is a bad PR. Even then, converse with more experienced people to make sure.

How To Test

In order to test properly, you'll have to download the code from the pull request, compile again and run the tests. If you can think of a way to test the feature in some way manually, so much the better, but that is not required.

Most of the time, the pull request will include one or more tests. If the coder didn't provide test coverage, try to understand why—refactors often don't for obvious reasons. If you think there should be a test, you can write "this needs a test" in the PR. Better yet, write the test yourself

and let the author know where it can be cherry-picked in the PR! This is an excellent way to build up some good will from people whose code you're reviewing.

Your job as a tester is to make sure the tests pass and that they sufficiently cover the functionality introduced. A good comment in a review might be "It would be nice if such-and-such situation could be covered by a test that does this-and-this".

After testing, be sure to note in the PR that you've tested.

Better Pull Requests

Eventually, you will get to a point when you feel comfortable creating pull requests of your own. A pull request can be anything from adding documentation to consensus-critical functionality. Whatever the change, the key to making good pull requests is to make it easy to understand and painless to review.

To that end, please don't torture your reviewers by making your PR a single commit with 3000 lines of changes. Separate those changes into easy-to-review commits of less than 300 lines (or maybe less than 100!) and grouped appropriately. As an example, you should put formatting changes into one commit, actual coding changes into another and large moves of code blocks in yet another.

Take pains to explain exactly what you're doing and the reasoning behind what you've done in each commit. I can't emphasize this part enough. Bend over backwards to make everything in your PR easy to understand! If you don't, you won't get anybody reviewing your code.

When reviewers comment on something or suggest changes, try to understand what the reasoning is. If you don't understand, ask until it's clear what they want you to do. If you agree, make the appropriate changes and let your reviewers know, but if not, make sure you have a mature, tactful dialogue with them to see how you can get their ACKs.

PR Ideas When Starting Out

Here are some PR ideas that you can contribute right away (remember, review 3x the PR's you submit!)

- Making documentation, particularly about setup, clear
- Writing unit or integration tests for something that is not tested yet
- Writing comments in undocumented code to help other people find their way

Conclusion

The software development practices used by Bitcoin Core are often not followed in other environments. Most people coming into Core development find the process overly restrictive and constraining. I assure you there is a reason for every step.

Remember to be courteous, gentle and tactful. Come in with a humble attitude and a desire to learn and that will set you on the path to not only becoming a better developer, but also a force for good in the Bitcoin community.