CONTRIBUTING

The libuv project welcomes new contributors. This document will guide you through the process.

FORK

Fork the project on GitHub and check out your copy.

```
$ git clone https://github.com/username/libuv.git
$ cd libuv
$ git remote add upstream https://github.com/libuv/libuv.git
```

Now decide if you want your feature or bug fix to go into the master branch or the stable branch. As a rule of thumb, bug fixes go into the stable branch while new features go into the master branch.

The stable branch is effectively frozen; patches that change the libuv API/ABI or affect the run-time behavior of applications get rejected.

In case of doubt, open an issue in the issue tracker, post your question to the libuv discussions forum, or message the libuv mailing list.

Especially do so if you plan to work on something big. Nothing is more frustrating than seeing your hard work go to waste because your vision does not align with that of the project maintainers.

BRANCH

Okay, so you have decided on the proper branch. Create a feature branch and start hacking:

```
$ git checkout -b my-feature-branch -t origin/v1.x
```

(Where v1.x is the latest stable branch as of this writing.)

CODE

Please adhere to libuv's code style. In general it follows the conventions from the Google C/C++ style guide. Some of the key points, as well as some additional guidelines, are enumerated below.

- Code that is specific to unix-y platforms should be placed in `src/unix`, and declarations go into `include/uv/unix.h`.
- Source code that is Windows-specific goes into `src/win`, and related publicly exported types, functions and macro declarations should generally be declared in `include/uv/win.h`.
- Names should be descriptive and concise.
- All the symbols and types that libuv makes available publicly should be prefixed with `uv_` (or `UV_` in case of macros).
- Internal, non-static functions should be prefixed with `uv__`.
- Use two spaces and no tabs.
- Lines should be wrapped at 80 characters.
- Ensure that lines have no trailing whitespace, and use unix-style (LF) line endings.

- Use C89-compliant syntax. In other words, variables can only be declared at the top of a scope (function, if/for/while-block).
- When writing comments, use properly constructed sentences, including punctuation.
- When documenting APIs and/or source code, don't make assumptions or make implications about race, gender, religion, political orientation or anything else that isn't relevant to the project.
- Remember that source code usually gets written once and read often: ensure the reader doesn't have to make guesses. Make sure that the purpose and inner logic are either obvious to a reasonably skilled professional, or add a comment that explains it.

COMMIT

Make sure git knows your name and email address:

```
$ git config --global user.name "J. Random User"
$ git config --global user.email "j.random.user@example.com"
```

Writing good commit logs is important. A commit log should describe what changed and why. Follow these guidelines when writing one:

1. The first line should be 50 characters or less and contain a short description of the change prefixed with the name of the changed subsystem (e.g. "net: add localAddress and localPort to Socket").
2. Keep the second line blank.
3. Wrap all other lines at 72 columns.

A good commit log looks like this:

```
subsystem: explaining the commit in one line

Body of commit message is a few lines of text, explaining things
in more detail, possibly giving some background about the issue
being fixed, etc etc.

The body of the commit message can be several paragraphs, and
please do proper word-wrap and keep columns shorter than about
72 characters or so. That way `git log` will show things
nicely even when it is indented.
```

The header line should be meaningful; it is what other people see when they run `git shortlog` or `git log --oneline`.

Check the output of `git log --oneline files_that_you_changed` to find out what subsystem (or subsystems) your changes touch.

REBASE

Use `git rebase` (not `git merge`) to sync your work from time to time.

```
$ git fetch upstream
$ git rebase upstream/v1.x  # or upstream/master
```

TEST

Bug fixes and features should come with tests. Add your tests in the `test/` directory. Each new test needs to be registered in `test/test-list.h`.

If you add a new test file, it needs to be registered in three places:

- `CMakeLists.txt`: add the file's name to the `uv_test_sources` list.
- `Makefile.am`: add the file's name to the `test_run_tests_SOURCES` list.

Look at other tests to see how they should be structured (license boilerplate, the way entry points are declared, etc.).

Check README.md file to find out how to run the test suite and make sure that there are no test regressions.

PUSH

```
$ git push origin my-feature-branch
```

Go to https://github.com/username/libuv and select your feature branch. Click the 'Pull Request' button and fill out the form.

Pull requests are usually reviewed within a few days. If there are comments to address, apply your changes in a separate commit and push that to your feature branch. Post a comment in the pull request afterwards; GitHub does not send out notifications when you add commits.