Development

This document is primarily for developers only.

General principals

1. Name everything well.
2. Strike a balance between simplicity and not-repeating code.
3. Methods that start with the word 'find' can return a null, methods that start with 'get' should not.
4. Keep methods short - it makes it easier to test.
5. Don't be afraid of moving code to a new file - it helps to reduce test dependencies.
6. Avoid noise-words in variable names, like 'data' or 'info'. Think about what you're naming and name it well. Don't be afraid to rename anything.
7. Avoid comments that describe what the code is doing, the code should describe itself. Comments are useful however for big-picture purposes and to document content of variables.
8. If you need to document a variable do it at the declaration, don't copy the comment to the `extern` usage since it will lead to comment rot.
9. Seek advice from other developers - know you can always learn more.
10. Be professional - attempts at humor or slating existing code in the codebase itself is not helpful when you have to change/fix it.
11. Know that there's always more than one way to do something and that code is never final - but it does have to work.

Before making any code contributions, try to comply with the coding style (CodingStyle.md). It has a lot of sound advice.

It is also advised to read about clean code, here are some useful links:

- http://cleancoders.com/
- http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29
- http://en.wikipedia.org/wiki/Code_smell
- http://en.wikipedia.org/wiki/Code_refactoring
- http://www.amazon.co.uk/Working-Effectively-Legacy-Robert-Martin/dp/0131177052

Unit testing

Ideally, there should be tests for any new code. However, since this is a legacy codebase which was not designed to be tested this might be a bit difficult.

If you want to make changes and want to make sure it's tested then focus on the minimal set of changes required to add a test.

Tests currently live in the `test` folder and they use the google test framework.
The tests are compiled and run natively on your development machine and not on the target platform.
This allows you to develop tests and code and actually execute it to make sure it works without needing a development board or simulator.

This project could really do with some functional tests which test the behaviour of the application.

All pull requests to add/improve the testability of the code or testing methods are highly sought!

Note: Tests are written in C++ and linked with with firmware's C code. All code is also instrumented using gcov to make test coverage analysis possible.

Running the tests.

The tests and test build system is very simple and based off the googletest example files, it will be improved in due course. From the root folder of the project simply do:

```
make test
```

You can also do:

```
make junittest
```

This will build a set of executable files in the `obj/test` folder, one for each `*_unittest.cc` file. It will stop after first compile/build error. If you want it to continue with the next test module you can use `make -k test`.

After they have been executed by the make invocation, you can still run them on the command line to execute the tests and to see the test report. Test reports will also be produced in form of junit XML files, if tests are built and run with the "junittest" goal. Junit report files are saved in obj/test directory and has the following naming pattern test_name_results.xml, for example: obj/test/battery_unittest_results.xml

You can also step-debug the tests in eclipse and you can use the GoogleTest test runner to make building and re-running the tests simple.

The tests are currently always compiled with debugging information enabled, there may be additional warnings, if you see any warnings please attempt to fix them and submit pull requests with the fixes.

Tests are verified and working with GCC 4.9.3

Test coverage analysis

There are a number of possibilities to analyse test coverage and produce various reports. There are guides available from many sources, a good overview and link collection to more info can be found on Wikipedia:

https://en.wikipedia.org/wiki/Gcov

A simple report for a single test can for example be made using this command:

```
gcov -s src/main/sensors -o obj/test/ battery_unittest.cc
```

To produce an coverage report in xml format usable by the Cobertura plugin in Jenkins requires installation of a Python script called "gcovr" from github:

https://github.com/gcovr/gcovr/tree/dev

Example usage in Jenkins:

```
/gcovr-install-path/gcovr/scripts/gcovr obj/test --root=src/main -x >
coverage.xml
```

There are many other ways to prodice test coverage reports in other formats, like html etc etc.

Using git and github

Ensure you understand the github workflow:
https://guides.github.com/introduction/flow/index.html

Please keep pull requests focused on one thing only, since this makes it easier to merge and test in a timely manner.

If you need help with pull requests there are guides on github here:

https://help.github.com/articles/creating-a-pull-request/

The main flow for a contributing is as follows:

1. Login to github, go to the cleanflight repository and press `fork`.
2. Then using the command line/terminal on your computer: `git clone <url to YOUR fork>`
3. `cd cleanflight`
4. `git checkout master`
5. `git checkout -b my-new-code`
6. Make changes
7. `git add <files that have changed>`
8. `git commit`
9. `git push origin my-new-code`
10. Create pull request using github UI to merge your changes from your new branch into `cleanflight/master`
11. Repeat from step 4 for new other changes.

The primary thing to remember is that separate pull requests should be created for separate branches. Never create a pull request from your `master` branch.

Once you have created the PR,
every new commit/push in your branch will propagate from your fork into the PR in the main github/cleanflight repo.
Checkout another branch first if you want something else.

Push will often fail if you edit or squash commits in a branch already pushed. Never do such things after creating the PR.

Later, you can get the changes from the cleanflight repo into your `master` branch by adding cleanflight as a git remote and merging from it as follows:

1. `git remote add cleanflight https://github.com/cleanflight/cleanflight.git`
2. `git checkout master`
3. `git fetch cleanflight`
4. `git merge cleanflight/master`
5. `git push origin master` is an optional step that will update your fork on github

You can also perform the git commands using the git client inside Eclipse. Refer to the Eclipse git manual.