Code Contributions

==================

Do you have a new cool feature that you'd like to contribute to

Contiki? Or a fix for a bug? Great! The Contiki project loves code

contributions, improvements, and bugfixes, but we require that they

follow a set of guidelines and that they are contributed in a specific

way.

Additional rules apply for contributions of a new hardware platform.

General Advice

--------------

The chance of getting your pull request accepted increases considerably

if you adhere to the following rules in addition to the aforementioned

formatting and naming standards:

\* Ensure that all contributed files have a valid copyright statement

and an open-source license.

\* Do not bundle commits that are unrelated to each other -- create

separate pull requests instead.

\* Adhere to ISO C99 in all C language source files. Exceptions are

allowed for those platform-dependent source files that rely on the

extensions of a specific set of compilers.

\* Clean up the commit history. "git rebase -i" is useful for this purpose.

\* Do not include executable binary files, because they are usually

rejected for security reasons. Instead, provide instructions for how

to compile the file, so that a trusted member of the merge team can

commit it.

\* Write a descriptive pull request message. Explain the advantages and

disadvantages of your proposed changes.

\* Before starting to work on a major contribution, discuss your idea

with experienced Contiki programmers (e.g., on the contiki-developers

mailing list) to avoid wasting time on things that have no chance of

getting merged into Contiki.

Source code that goes into the mainline Contiki repository must be of

interest to a large part of the Contiki community. It must be

well-tested and the merge team must have confidence that the code can

be maintained over a longer period. See below for more details

pertaining to platform contributions.

Contributions that have been made in research projects, and typically

do not get maintained thereafter, are better suited for inclusion in

the Contiki projects repository.

Structuring Commits

-------------------

\* Write descriptive commit messages. They don't have to be very long,

but you should mention what the commit achieves. Commit messages

like "modified foo/bar.c" are not helpful, should not be used, and

are likely to result in you having to re-write them.

\* Please do not add / remove irrelevant new line markers. Don't remove

the new line marker at the EOF.

\* Please, make sure that your patch doesn't add lines with trailing

whitespaces. If you run uncrustify as discussed above, this should

get taken care of for you automatically.

\* More generally speaking, make sure that each commit in your history

only includes changes necessary to implement whatever it is the

commit is trying to achieve. All changes should be mentioned in the

commit message.

Code Formatting

---------------

We require that all code contributed to the Contiki tree follows the

same code formatting as the existing Contiki code. We are very strict

on this.

Code must be formatted according to

[contiki/doc/code-style.c](https://github.com/contiki-os/contiki/blob/master/doc/code-style.c).

The Contiki source tree contains scripts to assist with correct code formatting

and we recommend [Uncrustify](http://uncrustify.sourceforge.net/) as the

preferred auto formatter. Everything is under

[tools/code-style](https://github.com/contiki-os/contiki/tree/master/tools/code-style).

If you wish, you can format all changed resources in your working tree

automatically if the

[tools/code-style/uncrustify-changed.sh](https://github.com/contiki-os/contiki/blob/master/tools/code-style/uncrustify-changed.sh)

script is added as a [Git pre-commit

hook](http://git-scm.com/book/en/Customizing-Git-Git-Hooks) to your Git

configuration.

Here are some examples of what you can do:

\* To check a file's style without changing the file on disk, you can run this:

`./tools/code-style/uncrustify-check-style.sh <path-to-file>`

This script will only accept a single file as its argument.

\* To auto format a file (and change it on disk) you can run this:

`./tools/code-style/uncrustify-fix-style.sh <path-to-file>`

\* `uncrustify-fix-style.sh` will accept a space-delimited list of files as its argument. Thus, you can auto-format an entire directory by running something like this:

``./tools/code-style/uncrustify-fix-style.sh `find cpu/cc2538 -type f -name "\*.[ch]"` ``

This is \_not\_ a silver bullet and developer intervention is still required. Below are some examples of code which will get misformatted by uncrustify:

\* Math symbol following a cast to a typedef

```

a = (uint8\_t) ~P0\_1; /\* Cast to a typedef. Space gets added here (incorrect) \*/

a = (int)~P0\_1; /\* Cast to a known type. Space gets removed (correct) \*/

a = (uint8\_t)P0\_1; /\* Variable directly after the cast. Space gets removed (correct) \*/

```

\* `while(<condition>);` will become `while(<condition>) ;` (space incorrectly added after closing paren)

\* ??`asm("wfi");` becomes `asm ("wfi");`: A space gets added before the opening paren, because the `asm` keyword stops this from getting interpreted as a normal function call / macro invocation. This is only a problem with `asm`. For instance, ??`foo("bar");` gets formatted correctly.

Naming

------

We require that all code contributed to the Contiki tree follow the

Contiki source code naming standard:

\* File names are composed of lower-case characters and dashes. Like

this: simple-udp.c

\* Variable and function names are composed of lower-case characters

and underscores. Like this: simple\_udp\_send();

\* Variable and function names that are visible outside of their module

must begin with the name of the module. Like this:

simple\_udp\_send(), which is in the simple-udp module, declared in

simple-udp.h, and implemented in simple-udp.c.

\* C macros are composed of upper-case characters and underscores. Like

this: PROCESS\_THREAD().

\* Configuration definitions begin with the module name and CONF\_. Like

this: PROCESS\_CONF\_NUMEVENTS.

How to Contribute Code

----------------------

When your code is formatted according to the Contiki code style and

follows the Contiki naming standard, it is time to send it to the

Contiki maintainers to look at!

All code contributions to Contiki are submitted as [Github pull

requests](https://help.github.com/articles/using-pull-requests). Pull

requests will be reviewed and accepted according to the guidelines

found in the next section.

The basic guidelines to to start a Pull-Request:

\* Create a new branch for your modifications. This branch should be based on the latest contiki master branch.

\* If you already added the commits to another branch you can [cherry-pick](http://git-scm.com/docs/git-cherry-pick) them onto your new branch.

\* Push the new branch to github.

\* Raise the new Pull Requests on this new branch. Raising a Pull Request for the master branch is almost always a bad idea.

\* If changes are requested do not close the pull request but rewrite your history. [Details about rewriting your history](http://git-scm.com/book/en/Git-Tools-Rewriting-History)

\* You now force-push the changes to github. The pull-request is automatically updated.

In Git terminology this is equivalent to:

\* Make sure you have the original contiki repo as origin.

```bash

$ git remote -v

contiki-orig https://github.com/contiki-os/contiki.git

```

\* If not add it

```bash

$ git remote add contiki-orig https://github.com/contiki-os/contiki.git

```

\* Make sure you have the latest version of your remotes

```bash

$ git remote update

```

\* Create a new branch "my\_new\_feature" based on the latest contiki master branch

```bash

$ git checkout contiki-orig/master -b my\_new\_feature

```

\* Add your work. For example by cherry-picking your changes from another branch.

```bash

$ git cherry-pick <HASH OF COMMIT>

```

\* Push to \_your\_ github repository

```bash

$ git push origin my\_new\_feature

```

\* Make a Pull Request for that branch

\* Rewrite your history if requested

```bash

$ git rebase -i contiki-orig/master

```

\* As rewriting your history can break things you must force-push the changes. \*\*Warning\*\*: Force-pushing normally is dangerous and you might break things. Make sure you are never force-pushing branches other people are supposed to work with.

```bash

$ git push origin my\_new\_feature -f

```

\* NOTE: To avoid all the pain of selectively picking commits, rebasing and force-pushing - begin your development with a branch OTHER THAN your master branch, and push changes to that branch after any local commits.

Pull Request Merging Policy

---------------------------

Pull requests (PRs) are reviewed by the [merge team](https://github.com/orgs/contiki-os/people).

Generally, PRs require two "+1" before they can be merged by someone on the merge team.

The since Contiki 3.0, the merging policy is the following:

\* The PR receives \*\*one "-1"\*\* from a merge team member (along with specific feedback). The PR is closed. A "-1" must be accompanied with a clear explanation why the PR will not be considered for inclusion.

\* The PR receives \*\*two "+1"\*\* from merge team members. The PR is merged.

\* The PR was inactive for \*\*two months\*\*. A team member may either:

\* Comment "Is there any interest for this PR? Is there any work pending on it? If not I will close it in \*\*one month\*\*." Back to initial state in case of activity, close otherwise.

\* Comment "I approve this PR. If nobody disapproves within \*\*one month\*\*, I will merge it." Back to initial state in case of activity, merge otherwise.

There is an exception to the rule.

Code that requires esoteric expertise such as some applications, platforms or tools can be merged after a single "+1" from its domain expert.

Travis / Regression testing

---------------------------

[Travis](https://travis-ci.org/) is a service that runs regression

tests. If you make a pull-request for Contiki this is automatically

forwarded to Travis and regression tests are run. A box with

information about the state of you pull request should show up after a

minute or two.

If the test fails it is likely that something is wrong with your

code. Please look carefully at the log. It might also be that some

package on the testing VM was updated and causes the build to fail. If

you are sure that is is not your code causing the tests to fail start

a new issue describing the problem. Also note this in your pull

request.

You can also register at [Travis](https://travis-ci.org/) for

free. Once you activated your Contiki repository, every push will be

tested at Travis. The configuration is part of the contiki repository

and testing will therefore work out-of-the-box. At Travis you then get

an overview of the state of each of your branches.

New Platforms

-------------

A new hardware port will be considered for inclusion in mainline Contiki

if it satisfies the following rules:

\* There must be at least one person willing and committed to maintain it.

They may but do not have to be the people who wrote the code. Similarly,

they may but do not have to be affiliated with the hardware manufacturer.

In the first instance, code maintenance would mean keeping the port up to

speed by submitting pull requests as Contiki moves forward. In the longer

term, people who maintain a reasonable level of commitment and who demonstrate

that they know what they're doing may be invited to become repo collaborators.

\* The hardware must be commercially available and of interest to a wide audience.

In other words, ports for bespoke hardware built for e.g. a specific project /

a single customer / niche markets are more suitable for a Contiki fork.

\* The code must strictly adhere to the Contiki code style, as discussed above.

\* The new files must have a clear copyright notice and license header. Contiki's

preferred software license is the

[3-clause BSD](http://opensource.org/licenses/BSD-3-Clause).

Other licenses may also be considered

as long as they are compatible with the 3-clause BSD (e.g. the Apache 2.0 license).

Conversely, code distributed under GPL cannot be considered. The same applies to

bespoke licenses, such as those allowing use or redistribution only together with

certain kinds of hardware.

\* The port must demonstrate a certain degree of completeness and maturity. Common sense

applies here.

\* The port must be accompanied by examples demonstrating basic functionality. This could

be a set of examples under `examples/<new-hardware-port>` and/or documentation of

which existing examples are meant to work.

\* The port must provide compile regression tests by extending the existing travis

integration testing framework. Again, we can't specify explicitly

what those tests should be, but something more interesting than hello-world is expected.

\* The work must be documented. The documentation could be README.md files

under the platform / cpu / example dirs or wiki pages. Doxygen comments are

also encouraged. The documentation should include:

\* A getting started guide, including a list of tools required to use the platform

(e.g. toolchain, software to program the device), where to get them from and brief notes

how to install them (can simply be a list of links to external guides)

\* A list of things which will work off the shelf

\* A list of things which are not meant to work, if any

\* Additional reading resources (e.g. datasheets, hardware user guides, web resources)

\* A ToDo list, if applicable.

\* It must be possible to use the port using free software. We do not discourage the

use of commercial software (e.g. support for a commercial toolchain), quite the opposite.

However, we will insist on the existence of a free alternative for everything.

After the port has been accepted, things meant to work off the shelf should

keep working off the shelf as Contiki moves forward.

We appreciate that, for many people, contributing to Contiki is a spare time

activity and our expectations from port maintainers take this into

consideration. All we ask from maintainers is to comment on and address

relevant pull requests at a reasonable frequency and to make sure travis keeps

passing. In other words, we just want platforms to stay healthy over time and

to thus avoid becoming very broken / obsolete.