# Contributing to open62541

Contributions to open62541 include code, documentation, answering user

questions, running the project's infrastructure, and advocating for all types of

open62541 users.

The open62541 project welcomes all contributions from anyone willing to work in

good faith with other contributors and the community. No contribution is too

small and all contributions are valued.

This guide explains the process for contributing to the open62541 project's core

repository and describes what to expect at each step. Thank you for considering

these point.

Your friendly open62541 community!

## Code of Conduct

The open62541 project has a [Code of Conduct](./CODE\_OF\_CONDUCT.md) that \*all\*

contributors are expected to follow. This code describes the \*minimum\* behavior

expectations for all contributors.

## Pull Requests

Everybody can propose a pull request (PR). But only the core-maintainers of the

project can merge PR.

### Minimal requirements for a PR

The following are the minimal requirements that every PR needs to meet.

- \*\*Pass Continuous Integration (CI)\*\*: Every PR has to pass our CI. This

includes compilation with a range of compilers and for a range of target

architectures, passing the unit tests and no detected issues with static code

analysis tools.

- \*\*Code-Style\*\*: Please consider the

[Code-Style](https://github.com/open62541/open62541/wiki/Code-Style)

recommendations when formatting your code.

- \*\*Signed CLA\*\*: Every contributor must sign the Contributor License Agreement

(CLA) before we can merge his first PR. The signing can be done online. The

link automatically appears on the page of the first PR. In addition, the CLA

text can be accessed [here](https://cla-assistant.io/open62541/open62541).

## Commit and PR Hygiene

We have very precise rules over how our git commit messages can be formatted. This leads to \*\*more

readable messages\*\* that are easy to follow when looking through the \*\*project history\*\*. But also,

we use the git commit messages to \*\*generate the change log\*\*.

This convention is identical to the [Conventional Commits](https://www.conventionalcommits.org) specification or the one used by Angular.

### Commit Message Format

Each commit message consists of a \*\*header\*\*, a \*\*body\*\* and a \*\*footer\*\*. The header has a special

format that includes a \*\*type\*\*, a \*\*scope\*\* and a \*\*subject\*\*:

```text

<type>(<scope>): <subject>

<BLANK LINE>

<body>

<BLANK LINE>

<footer>

```

The \*\*header\*\* is mandatory and the \*\*scope\*\* of the header is optional.

Any line of the commit message cannot be longer 100 characters! This allows the message to be easier

to read on GitHub as well as in various git tools.

The footer should contain a [closing reference to an issue](https://help.github.com/articles/closing-issues-via-commit-messages/) if any.

Samples: (even more [samples](https://github.com/angular/angular/commits/master))

```text

docs(server): add function documentation

```

```text

fix(core): fix parsing of endpoint url

Parsing of endpoint urls now also supports https

```

### Revert

If the commit reverts a previous commit, it should begin with `revert: `, followed by the header of the reverted commit. In the body it should say: `This reverts commit <hash>.`, where the hash is the SHA of the commit being reverted.

### Type

Must be one of the following:

- \*\*build\*\*: Changes that affect the build system or external dependencies

- \*\*ci\*\*: Changes to our CI configuration files and scripts (example scopes: travis, appveyor, fuzz)

- \*\*docs\*\*: Documentation only changes

- \*\*feat\*\*: A new feature

- \*\*fix\*\*: A bug fix

- \*\*perf\*\*: A code change that improves performance

- \*\*refactor\*\*: A code change that neither fixes a bug nor adds a feature

- \*\*style\*\*: Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)

- \*\*test\*\*: Adding missing tests or correcting existing tests

### Scope

The scope is optional, but recommended to be used. It should be the name of the component which is affected (as perceived by the person reading the changelog generated from commit messages).

The following is the list of supported scopes:

- \*\*arch\*\*: Changes to specific architecture code in `root/arch`

- \*\*client\*\*: Changes only affecting client code

- \*\*core\*\*: Core functionality used by the client and server

- \*\*ex\*\*: Example code changes

- \*\*mt\*\*: Changes specifically for multithreading

- \*\*nc\*\*: Nodeset compiler

- \*\*pack\*\*: Packaging setting changes

- \*\*plugin\*\*: Change to any (optional) plugin

- \*\*pubsub\*\*: Changes to the pubsub code

- \*\*sec\*\*: Changes to security, encryption, etc.

- \*\*server\*\*: Changes only affecting server code

### Subject

The subject contains a succinct description of the change:

- use the imperative, present tense: "change" not "changed" nor "changes"

- don't capitalize the first letter

- no dot (.) at the end

### Body

Just as in the \*\*subject\*\*, use the imperative, present tense: "change" not "changed" nor "changes".

The body should include the motivation for the change and contrast this with previous behavior.

### Footer

The footer should contain any information about \*\*Breaking Changes\*\* and is also the place to

reference GitHub issues that this commit \*\*Closes\*\*.

\*\*Breaking Changes\*\* should start with the word `BREAKING CHANGE:` with a space or two newlines. The rest of the commit message is then used for this.

## General commit hygiene

We are using the [Conventional Commits](https://www.conventionalcommits.org) specification (see previous section).

These sites explain a core set of good practice rules for preparing a PR:

- https://wiki.openstack.org/wiki/GitCommitMessages

- https://nvie.com/posts/a-successful-git-branching-model/

The following points will be especially looked at during the review.

- \*\*Separation of Concerns\*\*: Small changes are much easier to review.

Typically, small PR are merged much faster. For larger contributions, it might

make sense to break them up into a series of PR. For example, a PR with a new

feature should not contain other commits with only stylistic improvements to

another portion of the code.

- \*\*Feature Commits\*\*: The same holds true for the individual PR as well. Every

commit inside the PR should do one thing only. If many changes have been

applied at the same time, `git add --patch` can be used to partially stage and

commit changes that belong together.

- \*\*Commit Messages\*\*: Good commit messages help in understanding changes.

See previous section.

- \*\*Linear Commit History\*\*: Our goal is to maintain a linear commit history

where possible. Use the `git rebase` functionality before pushing a PR. Use

`git rebase --interactive` to squash bugfix commits.

### Review Process

These labels can be used for the PR title to indicate its status.

- `[WIP]`: The PR is work in progress and at this point simply informative.

- `[Review]`: The PR is ready from the developers perspective. He requests a review from a core-maintainer.

- `[Discussion]`: The PR is a contribution to ongoing technical discussions. The PR may be incomplete and is not intended to be merged before the discussion has concluded.

The core-maintainers are busy people. If they take especially long to react,

feel free to trigger them by additional comments in the PR thread. Again, small

PR are much faster to review.

It is the job of the developer that posts the PR to rebase the PR on the target

branch when the two diverge.

### Changes to the public API

The \*public\* API is the collection of header files in the /include folder.

Changes to the public API are under especially high scrutiny. Public API changes

are best discussed with the core-maintainers early on. Simply to avoid duplicate

work when changes to the proposed API become necessary.

You can create a special issue or PR just for the sake of discussing a proposed

API change. The actual implementation can follow later on.