FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Embedded Software Guidelines for C/C++ Projects

Version 0.1.7
FH Aachen
Institute of Applied Automation

IAAM

Contents

Preamble 4
Introduction 5
Software version control 5
  GIT 5
  Roles 5
  Tags 6
  Merge requests 7
  Merge request evaluation 8
  Issues 8
  CI/CD 9
Documentation 9
Changelog 11
Coding Style 12
  Vera++ 12
  KWStyle 13
Static analysis 13
  False positives 14
  When should an error be fixed? 14
    CPPCheck 14
    PVS Studio 14
    Codechecker 15
  Additional notes about the different SAT 15
    CPPCheck 15
    PVS studio 16
    Codechecker 16
Recommended Books 18
  C Language 18
  C++ Language 18
Source code documentation 20
  Comments at the beginning for source files 20
    Header file (.h,.hpp) 20
    Implementation file (.c,.cpp,.ino) 21
    Functions/Methods 22
    Enumerations 23

Structs 24
Variables 24
Referring to other parts of the program/files 25
CI/CD 25
Embedded docker container 25
Adding CI/CD to my project 25
Arduino Project 27
ROS 27
PlatformIO 27
Other 27

Preamble

This document is addressed to students who are working on embedded systems projects and are developing software with the C or C++ language. This document provides guidelines to improve the quality of code, reduce bugs and standardize programming styles for projects developed at the University of Applied Sciences of Aachen.

Introduction

This document contains software development guidelines for students that are developing embedded systems with the C/C++ language. The main objective is to standardize programming styles, increase the code quality and reduce bugs. As such, any student that is working on projects under the supervision of our department should strictly follow these guidelines for the evaluation of their projects. If these guidelines are not followed correctly the final evaluation or grade may be affected negatively.

Software version control

GIT

It is important that any project that requires developing software, uses a version control tool. At our university, we have a GitLab server hosted at https://git.fh-aachen.de/. This server uses the GIT software version control. Therefore, any software project that is developed by a student should be located on this server.

The student should be familiar with the git versioning software tool, and if necessary, should consult online material to learn how to use Git properly from the command line or graphical user interface.

Some useful links to start learning git:

- https://git-scm.com/docs/gittutorial
- https://thenewstack.io/tutorial-git-for-absolutely-everyone/
- https://rogerdudler.github.io/git-guide/
- https://www.geeksforgeeks.org/working-on-git-for-gui/

In addition, the student should be familiar with how Gitlab works. Not to be confused with Github which has similar functionalities but is not used in our server. Some useful links to getting started with GitLab are shown below:

- https://docs.gitlab.com/ee/topics/use_gitlab.html
- https://about.gitlab.com/get-started/
- https://www.youtube.com/watch?v=Jt4Z1vwtXT0&list=PLhW3qG5bs-L8YSnCiyQ-jD8XfHC2W1NL_
- https://www.youtube.com/watch?v=4lxvVj7wlZw

Roles

For each project, the supervisor or employee who manages the project for the students will create a group where the students will have developer access. For more information about the different GitLab, permissions refer to https://docs.gitlab.com/ee/user/permissions.html.

The developer role, allows students to push commits to any branch except the default one (normally called main or master). The main branch is only managed by the supervisor of the project or the FH employee who is leading the project. When the project has reached a specific milestone or objective, the student can request a review by doing a merge request. By doing a merge request the supervisor can evaluate if the changes are accepted and comply with these guidelines. As it will be further explained, if the merge request is rejected the student shall fix the problems indicated by the supervisor.

Tags

The student should create a tag when a certain objective has been accomplished and parts of the project are working. This allows us to navigate easily between the different versions of the project and to easily understand what have been the changes between versions. The tags should follow the format of the Semantic Versioning 2.0.0 specification. If you do not have experience with this specification please be sure to learn about it first.

An example of when to use a tag is described as follows. Assuming there is a project called “Autonomous Bicycle” and the software has the following functional requirements:

1. Exchange data with sensor ABC
2. Send data over WiFi via the MQTT protocol
3. Data Logging over an SD Card

Each time a requirement from the list is finished, the developer should increase the version of the software according to the Semantic Versioning 2.0.0 spec, and push a new tag.

The git tagging commands could look as follows, assuming the HEAD of the commit is pointing to the last commit where this functional requirement was completed.

v0.0.1

```
git tag -a v0.0.1 -m "* Sensor ABC communication complete" -m "* Tested sensor with PLC" -m "* First working version"
```

v0.0.2

```
git tag -a v0.0.2 -m "* MQTT communication added"
```

v0.1.0

```
git tag -a v0.0.3 -m "* SD Data Logging added" -m "* Maximum SD size 3GB" -m "* Note: Only SD Card Kingston tested"
```

Note that for tag v0.0.1 and tag v0.1.0, the tag message is multiline and for this reason, the example uses the `-m` argument multiple times. This could also be done by calling

```
git tag -a v1.2.3
```

This will open up your default code editor, where you could then easily write the message if it's too cumbersome to write via the command line.

The advantage of tagging is that one can navigate between the commits where the major changes have occurred. Also, if one requires to test a specific part of the project, one can use the tags to check out the specific commit instead of going through the complete commit history. In addition, if the project is not working, you could check out a tag that you know was working instead of trying to check the specific commit where the project worked before. You can consider tags as bookmarks or snapshots of the project.

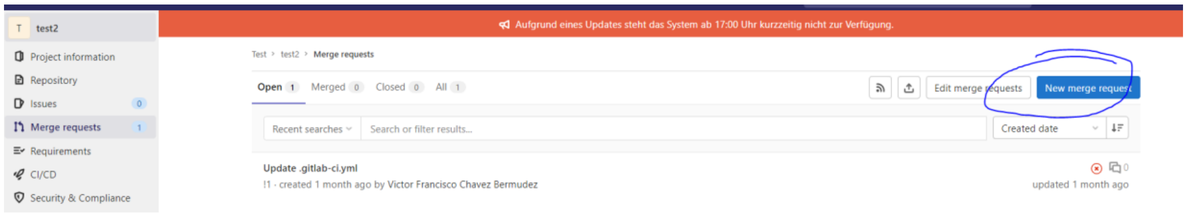
To checkout a specific tag you can use the `checkout` command with git.

```
git checkout v1.2.3
```

Merge requests

Merge requests allow pushing changes from one branch to another. In this case, the purpose of merge requests is two-fold. First of all, to evaluate the work of the student, and second to create a backup of the project in its current status. Merge requests should only be done when a certain objective has been accomplished and/or a code review is requested by the supervisor.

Merge requests can be done directly from the GitLab server by clicking Merge Request and clicking the New merge request button.

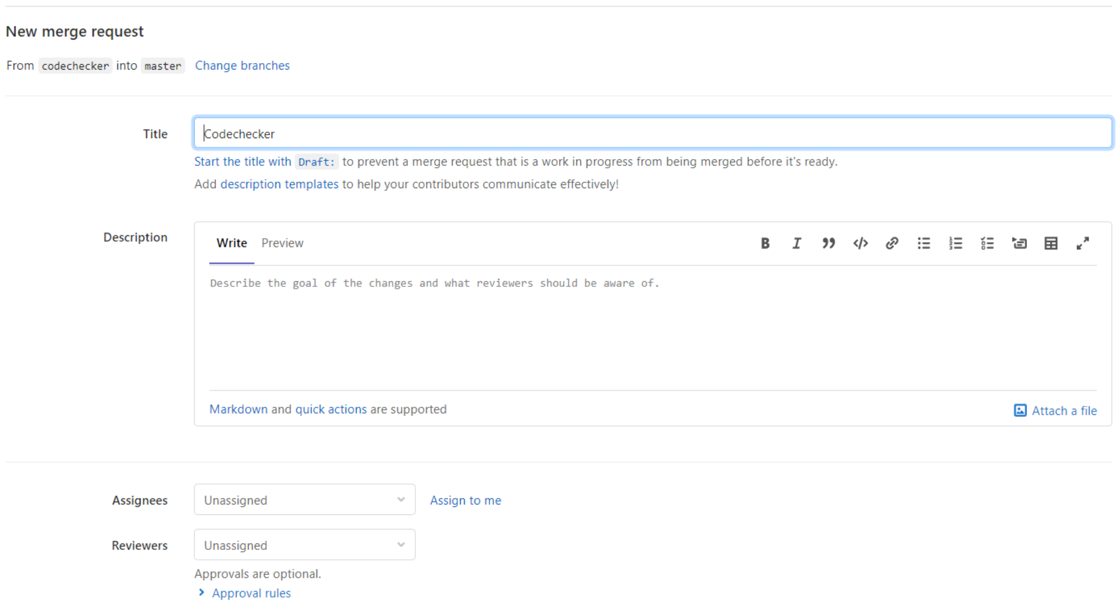


Aufgrund eines Updates steht das System ab 17:00 Uhr kurzzeitig nicht zur Verfügung.

Update .gitlab-ci.yml
!1 · created 1 month ago by Victor Francisco Chavez Bermudez
updated 1 month ago

The merge request requires certain information to be written before submission. An explanation of the REQUIRED fields to be completed by the student is as follows:

- Title: A short title that describes the purpose of the merge request
- Description: A complete description of what changes have been made to the software project.
- Assignees: Supervisor of the Software project
- Reviewers: Any additional employees or team members that will be participating in the code review.


New merge request

From codechecker into master Change branches

Title: Codechecker

Start the title with Draft: to prevent a merge request that is a work in progress from being merged before it's ready.

Add description templates to help your contributors communicate effectively!

Description: Describe the goal of the changes and what reviewers should be aware of.

Markdown and quick actions are supported

Assignees: Unassigned

Reviewers: Unassigned

Approvals are optional.

Merge request evaluation

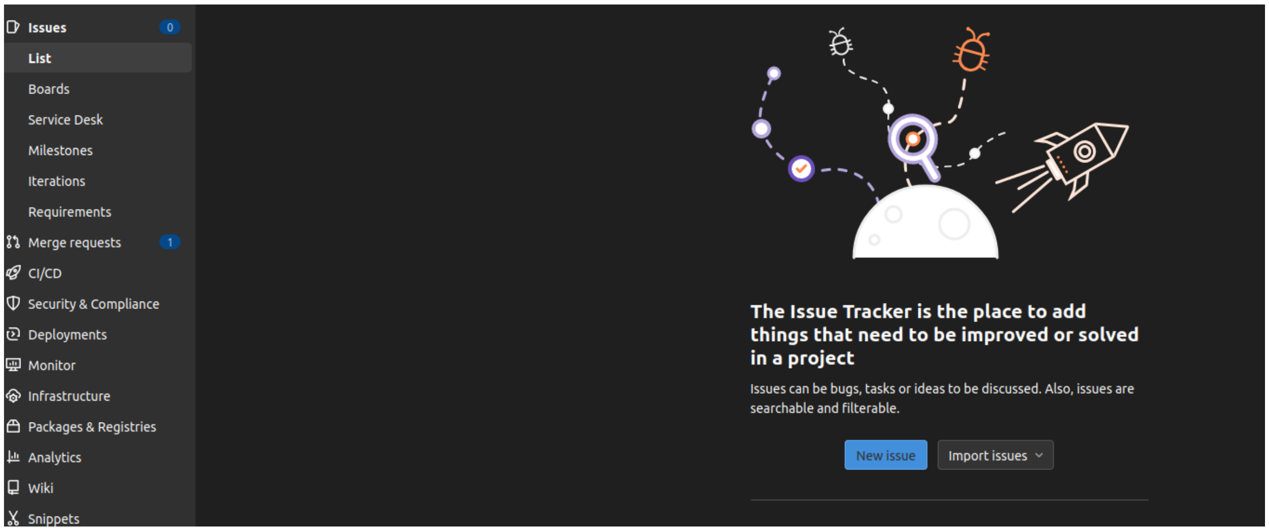
Once the Merge request is submitted, the supervisor will evaluate if the request can be merged to the main branch. The following conditions have to be met to accept the request:

1) **Software complies with these guidelines:** Any of the guidelines mentioned in this document should have been followed. To automatically detect if the guidelines are followed, CI/CD should be integrated into the repository. The specifics of the CI/CD are described later on, in this document. In general, the CI/CD will check that the coding style is correct and there are no major static analysis errors/warnings. If the student has not added CI/CD to his project, the supervisor has the right to close the merge request and accept the merge request until the student has done this. In the last section of this document, there are links to CI/CD templates that the student should use.

2) **No major software mistakes**: After a code review from the supervisor, if there aren't any comments about changes that have to be made, the commit is deemed to be safe to merge and assumed it complies with the requirements of the task. An example of when this might not be achieved is: The task is to send 2 bytes over USB and the student is sending 4 bytes instead or there was a misinterpretation of the requirements of the project.

Issues

A typical characteristic of software projects is that there are bugs in the code or the software does not work as expected. In addition, software developers might have questions regarding the implementation of the requirements or have a technical issues. Gitlab provides a way of tracking all of these problems in the form of Issues. If you encounter a problem while developing your project, cannot make the software work, or require to discuss a certain issue you should add a new issue in Gitlab so it can be discussed with your working colleagues or the supervisor.



The Issue Tracker is the place to add things that need to be improved or solved in a project

Issues can be bugs, tasks or ideas to be discussed. Also, issues are searchable and filterable.

An issue requires certain information to be written before submitting. An explanation of the REQUIRED fields to be completed by the student is as follows:

- Title: A short title that mentions the issue to be discussed.
- Description: A complete description of the issue. The description should be concise and explain what steps are needed to reproduce the issue, under what conditions it occurs, and a detailed description of the attempts that have been done to solve the problem. Please refer to the following links so you can properly describe your problem:
    - https://xyproblem.info/
    - https://blog.softexpert.com/en/how-to-describe-problem/
    - https://stackoverflow.com/help/how-to-ask
- Assignees: Persons that are involved in the issue.

CI/CD

Continuous Integration and Continuous development, better known as CI/CD allow for integration changes and verify the reliability of the code automatically each time a software change is made. Gitlab incorporates CI/CD, and each student software project should include it. What is important for us is that we can detect that the guidelines from this document are followed automatically. For this reason, the student should check the section CI/CD in this document.

Documentation

The main documentation of the software project is normally done as a separate report. Nevertheless, anything related to the Software (e.g. How to use the API, Example of usage, how to install) must be documented in the repository for anyone that will continue the project. The Student should include a Readme file with Markdown syntax. The readme should explain how you can compile the software, which requirements are needed, and any other detail related to how to use the software.

As a guideline, the student should have at least the following information in their README.md file:

- **Description**: A short description of the project.
- **Software Requirements**: Any software that is required to compile or use the project
- **Third Party Libraries:** Any libraries that are not developed by the student and are required to be downloaded. A link to the download URL of the library should be included, and a description of how it is used in the project. In case it's a Git repository**, the hash commit has to be included**. This is due to the fact, that a student might work with a library and 3 years later another student continues the project but the library has completely changed. With the hash commit, the next student can be sure that the project will still compile as if he had started the project 3 years ago.
- **Status:** Description of the parts of the software that work, do not work and those that are pending (a.k.a TODO List).
- **Conditions of Use:** Indicates any conditions necessary to operate the software correctly or under which circumstances the software has been tested
- **How to Use:** Any instructions on how to use the Software. This is typical of applications for Windows or Linux, where the software can be run from a command line or a GUI. In the case of a microcontroller, instructions should be indicated on which microcontroller is required or if a development board was used (e.g. Arduino Uno).
- **Notes**: Any remarks or comments about the software project that should be taken into account. For example: The Class ABC has presented a bug that only happens at 7:00 pm in Nepal.

An example of a simple Readme.md following this guideline looks as follows:

```
# My Final Software Project

## Description

This objective of this project is to create a submarine navigation algorithm for the FH Autonomous Navigation Project part
of the Mechatronics project of 2021 from Raul Mendoza and Sherly Pimmels.

## Software Requirements

-     Python 3.3.3
-     Linux 5.1
-     ROS 2
-     Arduino IDE 1.47

## Third party Libraries

-     DueTimer: Arduino Library for controlling a timer
          o    Link: https://github.com/ivanseidel/DueTimer
          o    Hash commit: 1ac2e5da962029fbe12b8faad3808b03694b0c9f
-     ROS 2 OPCUA: OPC UA communication for the ROS submarine
          o    Lnik: https://github.com/Mariunil/ros2-opcua
          o    Hash commit: 2857e21e56131a6f4f48e6674b46c1fefaaad140

## Status

### Working

* The submarine algorithm can navigate in straight paths

* Detection of whales

### Not Working

* Detection of Seals
```

* Navigation in depths greater than 1km

### TODO

* Improve feedback algorithm

## Conditions of Use

The software has been tested only in the lake Takahashi at the depth of 20 meters. Note: It is required to have a swimming permit to test the submarine at 10 pm in Fall.

The algorithm works only in sweet water and was not tested in the Ocean.

## How to Use

To use the software, first it has to be compiled with the Arduino IDE 3.23. The generated hex file has to be uploaded via serial communication. If using linux the command “submarine go” has to executed before uploading to the main board.

A list of complete commands is written below….

## Notes

The project has been developed as a proof-of-concept and therefore does not have all the features of an autonomous submarine. There was a short-circuit caused due to a current leak in the main board v3.23, for this reason there is a temporal function in the software called “ShortCircuitFix()” that should be called in the meantime.

Changelog

A changelog is a file that is used to track the progress of the report for each version of the project. The student must include a Changelog in their repository for each new version that is released. The changelog should be located at the root of the GIT repository of the project. The file should be called Changelog.txt and follow the guidelines from

https://keepachangelog.com/en/1.0.0/

For each new version, the student should write the following types of changes in the changelog.

- **Added** for new features.
- **Changed** for changes in existing functionality.
- **Deprecated** for soon-to-be removed features.
- **Removed** for now removed features.
- **Fixed** for any bug fixes. If the bug fix was done as part of a Gitlab Issue you should post as well the Issue # and URL from GitLab.
- **Security** in case of vulnerabilities.

Please note that not all of the categories are required for each version and it depends on what has happened in the new version that you release.

Coding Style

To standardize the code style of the projects, a set of rules have been curated from the Vera++ software and KWStyle.

Vera++

Vera++ is a software tool for verification, analysis, and transformation of C++ source code. Information about all the vera++ rules is found here.

The set of rules required for these guidelines are the following:

- F001
- F002
- L003
- L004 – Note: max-line-length=125
- L005
- L006
- T001
- T002
- T003
- T004
- T005
- T006
- T007
- T009
- T010
- T011
- T012
- T015
- T017
- T018
- T019

These rules can be verified by downloading the vera++ software and installing a Rules profile in the installation directory: *your_vera_install_path/profiles/*

For ease of use, the profile can be downloaded from our git repository:

https://git.fh-aachen.de/embeddedutils/vera-fh-profile/-/blob/main/FH.tcl

After installing the profile execute vera++ from the command line as follows for a single file:

```
vera++ file_to_analyze.cpp -p FH.tcl
```

Note: This assumes the vera++ executable is already added to your PATH environment.

And for multiple files, create a text file separated by a new line for each file you want to analyze and run the command as:

```
vera++ -i TXT_FILE_with_all_my_sources.txt -p FH.tcl
```

After executing vera++, the software will output all the rules that were violated.

KWStyle

KWstyle is a source style checker developed by Kitware (developer of CMake). This open-source software has a set of rules that can be configured. Documentation about all the rules is found here. For this document, the following rules shall be followed:

Rule Code	Additional Info
LEN	125 lines max
SEM	-
EOF	-
ESP	-
EML	-
SPL	-
FLN	450 lines max per function. Violating this rule indicates that the function is doing more tasks than it's supposed to or it could be simplified into smaller parts. If the function cannot be simplified or divided into smaller parts, the justification should be discussed with the supervisor.

The set of rules for use with the software can be downloaded from:

https://git.fh-aachen.de/embedded-guidelines/cfg/kwstyle_config

How to run Kwstyle with the set of rules

```
git clone https://git.fh-aachen.de/embedded-guidelines/cfg/kwstyle_config    KWStyle -D input_files.txt -xml kwstyle_config/kwstyle_embedded_guidelines.xml -html kwsytle_html
```

The previous two commands download the rules from git and execute KWstyle with our embedded guidelines rules. The output is an HTML webpage in the folder kwstyle_html. This example also assumes you have a text file separated by a new line for each file that you want to analyze.

Static analysis

To reduce the number of bugs and mistakes that a software project has, the student must use static analysis tools (SAT). These tools allow identifying without running the code, possible errors, and warnings that the code might have. For this guideline, the following static analysis tools must be used:

- CPPCheck (Open Source)

- Codechecker (Open Source, Only for Linux)
- PVS Studio (Requires a License, but a free student version is available)

The reason for using different SAT is that not all tools capture the same errors, and allow to discover more information about possible mistakes in the software. The reports generated from these tools will be used for the evaluation of the work of the student. The student does not require to send the reports to the supervisor, since the Gitlab server for each student software project will be configured to automatically generate the reports (see CI/CD).

However, the student should install the aforementioned tools in their computer to check easily for any warnings or errors each time they compile their code. This is due to the reason that the GitLab ci/cd is a remote server, and you would have to submit a new commit to check for the reports, which is slower. In contrast, if you have the tools locally you can instantly check any errors rather than waiting for the GitLab server and overloading it.

False positives

SAT software detects warnings or errors in a software project. However, sometimes the reports might include false positives. This means that the tool detected errors that are not in reality. For this reason, whenever the student thinks there is a false positive, they can ask the supervisor for clarifying if the error is a false positive and if they should ignore it.

When should an error be fixed?

In general, for the evaluation of the project all the errors should be fixed for the respective SAT as follows:

CPPCheck

Severity	Description
Error	Should be immediately fixed.
Warning	Should be immediately fixed.
Style	Should be consulted with the supervisor if they should be fixed.
Info	Should be consulted with the supervisor if they should be fixed.

PVS Studio

Severity	Description
High	Should be immediately fixed.
Medium	Should be immediately fixed.
Low	Should be immediately fixed.

Note: PVS studio Allows detection of bugs for different platforms. For the analysis, please choose the following analyzers from PVS Studio:

- General Analysis
- Optimization

For more information about the analyzers click here.

Codechecker

Severity	Description
High (H)	Should be immediately fixed.
Medium (M)	Should be immediately fixed.
Low (L)	Should be immediately fixed.

Additional notes about the different SAT

CPPCheck

CPPCheck can be used directly from a GUI or command line. The manual of the tool is located here. CPPCheck is the simplest tool to use as it does not require compiling the code to detect the files to analyze. There are different tutorials online, here is one for example.

CPPcheck in addition is compatible with Linux and Windows.

It is worth mentioning that not all of the rules from cppcheck are mandatory for these guidelines. For this reason, a custom set of rules has been made available at

https://git.fh-aachen.de/embedded-guidelines/cfg/cppcheck-cfg

To apply these rules in cppcheck you can execute the following command from a terminal:

```
cppcheck --suppressions-list=cppcheck_suppresion.txt --force--enable=all YOUR_SRC_DIR_TO_ANALYZE
```

How to deal with false positives

In the case CPPCheck reports issues with your code that are false positives you can follow the recommendations from the manual in the section Inline suppressions

https://cppcheck.sourceforge.io/manual.pdf

It is important to mention that you SHOULD ONLY suppress warnings by the methods discussed in the link above after consulting with your project supervisor.

PVS studio

To use PVS studio, you can use the free version, which requires adding a specific comment in each source file. To automate the task of adding the comments, we have a script that allows you to add the comments with python. Please refer to this repository for more information.

PVS-Studio can be used from the command line or its Monitoring GUI.

How to use PVS Studio:

- Windows
- Linux/MacOS

The complete documentation can be found here.

How to deal with false positives

In the case PVS-Studio reports issues with your code that are false positives you can follow the recommendations from PVS-Studio in the section **Manual false positives suppression** from the following site:

https://pvs-studio.com/en/docs/manual/0017/

It is important to mention that you SHOULD ONLY suppress warnings by the methods discussed in the link above after consulting with your project supervisor.

Codechecker

Codechecker is an open-source static analyzer tool developed by Ericsson. To use Codechecker, you require Linux. If you do not have Linux you can use the docker container we have at the FH to analyze your project with our CI/CD (refer to CI/CD).

Follow the instructions from https://github.com/Ericsson/codechecker to install Codechecker.

An example of how to use it from the command line

1) CodeChecker log -b "Your Command to Compile the project" -o codechecker.log
2) CodeChecker analyze codechecker.log --ctu --output codechecker_analysis --enable sensitive
3) CodeChecker parse --export html --output codechecker_html codechecker_analysis

Note: depending on the compiler, the following env. Variable should be set before executing the commands above CC_LOGGER_GCC_LIKE. This variable should be set to the correct compiler used for the project.

Example of some possible values for this variable (including double quotes)

Target	CC_LOGGER_GCC_LIKE variable value Including double quote

Linux application	“gcc:g++”
Microcontroller with Cortex CPU	“arm-none-eabi-gcc:arm-none-eabi-g++”
Microcontroller with AVR CPU	“avr-gcc:avr-g++”

To set the value in the command line you can run for example:

```
export CC_LOGGER_GCC_LIKE “gcc:g++”
```

It is worth mentioning that not all of the rules from codechecker are mandatory for these guidelines. For this reason, a custom set of rules has been made available at

https://git.fh-aachen.de/embedded-guidelines/cfg/codechecker-cfg

You can use the suppression file FH-Codechecker.json from this repository to filter the rules that should be disabled to comply with these guidelines. To use the file you can invoke codechecker as follows:

```
CodeChecker analyze YourCompileLog --ctu --enable extreme --output OutputDir --config codechecker-cfg/FH-Codechecker.json
```

How to deal with false positives

In the case the Clang static analyzer (clangsa) reports issues with your code that are false positives you can follow the recommendations from this site:

https://clang-analyzer.llvm.org/faq.html

In the case the warning does not come from clangsa, and is instead from clang tidy (clangtidy), to suppress warnings you can follow the instructions from this link:

https://clang.llvm.org/extra/clang-tidy/#suppressing-undesired-diagnostics

It is important to mention that you SHOULD ONLY suppress warnings by the methods discussed in the links above after consulting with your project supervisor.

Recommended Books

This section includes recommended books to deepen your knowledge of the C and C++ language. Specific sections are mentioned to point out important topics that should be known by any student developing projects with these guidelines. It is recommended to focus on specific section or topics you want to learn as the books might or could cover a broad category of topics. Most of these books can be obtained from the FH Aachen library either from a digital or hardcopy.

C Language

- Modern C, Jens Gustedt
- C: A reference Manual, Harbison & Steele
- Deep C Secrets, Peter van der Linden
- Object-Oriented Programming With ANSI-C, Axel Schreiner

C++ Language

C++ is a language that has been around since 1985 and as such has changed over time. Currently there are 6 specifications (C++98, C++03, C++11, C++14, C++17, C++20). Each of them refers to the year when they were published and each version has added new features, idioms, and syntax. For this reason, any of the best practices mentioned in this section should be doubled check as they might only apply to a specific version of C++.

- Real-Time C++, Christopher Kormanyos

  Recommended sections

  - An Easy Jump Start in Real-Time C++
  - Object-Oriented Techniques for Microcontrollers
  - C++ Templates for Microcontrollers
  - Optimized C++ Programming for Microcontrollers
  - The Right Start
  - Floating-Point Mathematics
  - Fixed-Point Mathematics
- Effective C++ 55 Ways to Improve Your Programs and Designs, Scott Meyers

  Recommended sections

  - Accustoming yourself to C++
  - Constructors, Destructors, and Assignment Operators
  - Design and declarators
  - Template and Generic programming
- Effective Modern C++ 42 Specific Ways to improve your use of C++11 and C++14, Scott Meyers

  Recommended sections
  - Deducing types
  - auto
  - Moving to Modern C++
  - Lambda Expressions
- C++ For Embedded Systems, Arkady Miasnikov

- The C++ Standard Library, A tutorial and reference, Nicolai M. Josuttis
- Modern C++ Programming cookbook, Marius Bancila
  Recommended sections

  Chapter 1:

  - Using-auto whenever possible
  - Understanding the various forms of non-static member initialization
  - Using scoped enumeration
  - Using override and final for virtual methods
  - Using range-based for loops to iterate on a range
  - Using unnamed namespaces instead of static globals

  Chapter 2

  - Converting between numeric and string types
  - Limits and other properties of numeric types

  Chapter 3

  - Defaulted and deleted functions

  Chapter 4

  - Conditionally compiling your source code
  - Using the indirection pattern for preprocessor stringification and concatenation
  - Performing compile-time assertion checks with static_assert
  - Selecting branches at compile time with constexpr if
  - Providing metadata to the compiler with attributes

  Chapter 9

  - Ensuring constant correctness for a program
  - Creating compile-time constant expressions
  - Performing correct type casts
  - Using unique_ptr to uniquely own a memory resource
  - Using shared_ptr to share a memory resource
  - Consistent comparison with the operator <=>

  Chapter 10

  - Implementing the pimpl idiom
  - Implementing the named parameter idiom
  - Separating interfaces and implementations with the non-virtual interface idiom
  - Handling friendship with the attorneyclient idiom
  - Static polymorphism with the curiously recurring template pattern
  - Implementing a thread-safe singleton

Source code documentation

This section should not be confused with the subsection Documentation of the section Software Version Control in this document. The purpose of this section is to define how the student should document the source files of their project (.c, .cpp,.ino) with comments.

Normally the documentation of the project is done as a separate report file where the main aspects of the design, requirements, UML diagrams, etc. are described. In contrast, source code documentation refers to the internal documentation written in the source code that should serve as a reference for anyone that wants to understand the source code implementation and modify it. This typically would be a software developer or in our case any student that will continue working on this project. Source code documentation is important as it allows anyone to understand implementation decisions made by the software developer and how to properly use any functions, classes, interfaces, etc.

To standardize how comments in source files for documentation are written, the student should follow the syntax from the Doxygen project. Doxygen is a source code documentation that supports C/C++ and can generate HTML output from comments in source files. The complete syntax and information on how to use Doxygen can be seen here:

- https://www.doxygen.nl/manual/index.html
- https://www.youtube.com/watch?v=TtRn3HsOm1s
- https://www.youtube.com/watch?v=5G1zUpNFmEY
- https://aaronbloomfield.github.io/pdr/tutorials/11-doxygen/index.html

The following subsections include guidelines based on the Doxygen syntax that the student has to follow for their source files.

Comments at the beginning for source files

Each source file should start with a block comment that includes information about the author of the file, the year it has been created, and any other information that can describe the purpose of the file. A template of what the student should add to a file is included (Doxygen Syntax).

Header file (.h,.hpp)

```
/*!***********************************************************
 * @file      NameOfYourFile.h
 * @author     Victor Chavez
 * @date      Mar 3, 2022
 *
 * @brief
 * A Short description of the purpose of the header file
 *
 * @details
 * Any details that should be mentioned about this header file <br>
 * Note that \<br\> is just an HTML line break as doxygen supports <br>
 * using HTML inside comments. <br>
 * Examples of what should be mentioned in details: <br>
 * -Reference to external links<br>
 * -Special notes about the design of Class/Interfaces/API<br>
 *
 * @par Dependencies
 * - language:          C++11
 * - OS:                      None
 * @ingroup GroupName
 *******************************************************************/
```

Notes:

Language should specify the programming language of the source file and the specific version that is required to compile the file (e.g., C++98, C++11, C17, C99).

OS should specify if the source file requires a specific operating system. The OS can be e.g. Linux, Windows, or for an embedded system it could relate to an RTOS such as FreeRTOS, Zephyr, ThreadX, uc-os, vx-works,etc.

GroupName should be a specific group in which this source file should be grouped. This allows organizing the source files in case multiple files are part of the same implementation. For example if you have MyClass.h and divide the implementation in Source1.cpp, Source2.cpp and Source3.cpp you can add all of these .cpp files to the same group. To define the group you can put at the end of your header file the following

```
/*!*******************************************
 * @defgroup MyAwesomeGroup The Awesome Group with spaces
 * This is the group that collects Source1.cpp, Source2.cpp,Source3.cpp and MyClass.h
 *********************************************/
```

Implementation file (.c,.cpp,.ino)

```
/*!********************************************************
 * @file       MySource.ino
 * @author      Victor Chavez
 * @date       Mar 3, 2022
 *
 * @brief
 * A Short description of the implementation file. For example
 * This file contains the implementation for MyCoolProject Class
 * based on the requirements documents of the Mechatronics project
 * Autonomous Submarine.
 *
 * @details
 * Any details that should be mentioned about the implementation<br>
 * Examples of what should be mentioned in details: <br>
 * - Special notes for compiling
 * - Obscure details that are easily forgotten.
 * - Remarks for any future developer that will have to continue
 *    developing this code
 *
 * @par Dependencies
 * - language:           C++11
 * - OS:                         None
 * @ingroup GroupName
 ******************************************************************/
```

Notes:

Language should specify the programming language of the source file and the specific version that is required to compile the file (e.g., C++98, C++11, C17, C99).

OS should specify if the source file requires a specific operating system. The OS can be e.g. Linux, Windows, or for an embedded system it could relate to an RTOS such as FreeRTOS, Zephyr, ThreadX, uc-os, vx-works,etc.

GroupName should be a specific group in which this source file should be grouped. This allows organizing the source files in case multiple files are part of the same implementation. For example, if you have MyClass.h and divide the implementation in Source1.cpp,Source2.cpp and Source3.cpp you can add all of these .cpp files to the same group. To define the group you can put at the end of your header file the following

```
/*!*******************************************
 * @defgroup MyAwesomeGroup The Awesome Group with spaces
 * This is the group that collects Source1.cpp, Source2.cpp,Source3.cpp and MyClass.h
 *********************************************/
```

Functions/Methods

Functions/Methods should be documented to describe how to use them. This allows understanding how to call the function/method properly without having to read the complete source code and guess how it works. Functions and methods should be commented in the HEADER file (.h/.hpp) and follow the next template:

```
/*! @brief Briefly describe the function
   @details Any details of the function that should
                              Mention when it should be used, any type
               Of conditions for its use, why is it used,
                              And if possible a short example to clarify
                              Any ambiguity with another similar function
            Or in case it's a complex function that not even
                              The author can remember how to use it without
                              Writing this comment.

   @param[in] Param1 Description of Parameter 1.
   @param[out] Param2 This is an output parameter
   @retval 0: Description for a return value of zero
   @retval >0: Description for a return value > zero
*/
```

Example:

```
/*! @brief Verifies that A > B
   @details Verifies that any number A that is provided
```

```
         from the Plank algorithm is greater than
                         a number B that originates from the
                    RGB matrix in the display.

  @param[in] a This is the number A that originates from the
                                    Plank algorithm. The unit of this parameter is
                                    In microvolts. The range of this parameter is
                                    From  -23 to 4500.
  @param[in] b This is the number B that originates from the
                                    RGB matrix. The units are in Amperes. The
                                    Maximum value should not exceed 4300.
  @param[out] c This is an output parameter that returns the
           Time it took the verification to processs.
           If a null value is given the verificiation is
           Cancelled.
  @retval 0: The calculation had an error
  @retval >0: The calculation was successful, the retval is
                         In microPixels
*/
int Verify(int a, int b, int * c);
```

The example above gives a good idea of how functions should be commented. Something worth mentioning is that even though the function has a return value that can be considered as an output, the parameters have an [in],[out] notation. The reason is that sometimes, functions or methods may include pointers that are used to modify variables outside the scope of the function/method with the purpose to give an output. For this example, parameter C is an output, but if for some reason parameter C was used to read a value only it could be changed to @param[in] to denote the actual intent. The @retval syntax allows describing possible outcomes of the function and what values can be expected.

Enumerations

Enumerations allow to store integer constants that are readable and easy to maintain. Students sometimes make the mistake of using obscure names or ambiguous naming conventions that no one else but them can understand. For this reason, each enumeration value and a description of the purpose of the enumeration should be written.

Enumerations should be commented on as follows:

```
/*! Description of the enumeration */
enum MyEnum
{
  EnumValue1=0,   /*!<Description of the first value*/
  EnumValue2,           /*!<Description of the second value*/
};
```

Example:

```
/*! Description of the possible
*  car types that the
* object detection algorithm can detect
*/
enum CarType
{
  Tesla=0,   /*!<Detection of Tesla S and Tesla XL models*/
  Ford       /*!<Detection of Ford Fiest and Ford Subaru models */
};
```

Structs

Structs allow collecting variables, functions, and other data types inside an organized unit called Struct. Sometimes students create structs with obscure names, ambiguous meanings or simply without any information on why they were created. For this reason, structures should be documented to allow anyone to know how to use them.

Structures should be commented on as follows:

```
/*! Description of the struct */
struct Mystruct
{
  int member1;   /*!<Description of the first member*/
  int member2;              /*!<Description of the second member*/
};
```

Example:

```
/*! Description of the
*    features of a microcontroller
*/
struct MicroFeatures
{
  Int clock_speed,   /*!<The clock speed of the microcontroller
                                          * unit is in Hz*/
  Long max_ram;      /*!<Maximum RAM size in KB */
};
```

Variables

The purpose of commenting variables is that anyone can understand its purpose in the program. Sometimes students create obscure variables that even they do not know their real purpose after reading trying to explain their code the next day. For this reason, they should be commented to give more information about their actual purpose, and if appropriate its type of unit and when can it be used.

Variables should be commented on as follows

```
Int myVar;      /*!<Description of the Variable */
```

Example

```
Int SuperSecretCount;      /*!<Stores the super secret number each time
                                                 *  a platypus swims in the lake */
```

Referring to other parts of the program/files

Doxygen automatically links all of the files, functions, classes,etc. If you need to refer to a function a variable of a Class, etc you can easily do it in the Doxygen comments. More information is provided here:

https://www.doxygen.nl/manual/autolink.html

CI/CD

To automate the process of checking that these guidelines are followed, we have developed a docker container that can check for rules violated by the SAT software and style checks. You should be familiar with GitLab CI/CD to understand this section. A couple of links for a more detailed understanding of Gitlab CI/CD are provided below:

- https://www.youtube.com/watch?v=jUiKi6FWYrg&list=PLhW3qG5bs-L8YSnCiyQ-jD8XfHC2W1NL_&index=7
- https://docs.gitlab.com/ee/ci/quick_start/
- https://www.youtube.com/watch?v=B68jcGfH4C8
- https://www.youtube.com/watch?v=34u4wbeEYEo&list=PLaFCDlD-mVOlnL0f9rl3jyOHNdHU--vlJ
- https://www.youtube.com/watch?v=Jav4vbUrqII

Embedded docker container

To automate the process of the embedded guidelines of this document we use a Docker container that already has installed the necessary tools/software described in the sections above. The source files of the container are located at https://git.fh-aachen.de/embeddedtools/embedded-tools-image. The student can check this repository to understand how we created the container and understand how the Gitlab CI/CD uses the container.

This container can be integrated into your GitLab repository by creating a ***.gitlab-ci.yml*** file in the root directory of it. To use the image, you can add the following line to your Gitlab CI/CD job:

```
image: registry.git.fh-aachen.de/embeddedtools/embedded-tools-image:core
```

More information about using docker containers in GitLab CI/CD is found here.

Adding CI/CD to my project

Depending on the toolchain and compiler you require for the project there are different ways to set up your project to check that it complies with the guidelines of this document. The easiest way is to use one of the templates suggested in the next section to get started. The templates require some modifications for the specific project and the student should check the README.md file of each template for a proper understanding of how to apply it to the project they are working with.

It is REQUIRED that the student uses one of the templates below to evaluate their project. If they have not done this, the supervisor reserves his right to evaluate the project until he applies CI/CD into his GitLab repository.

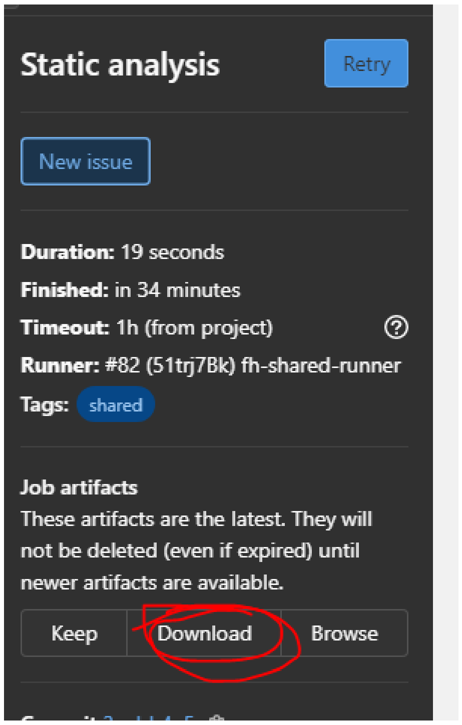
Once the CI/CD is applied to the project, you will notice that each time you make a new commit a new pipeline is created. More information about pipelines is provided here:

https://docs.gitlab.com/ee/ci/pipelines/

The pipeline activated by the .gitlab-ci,yml of these guidelines have the following stages:

- Style: Checks for coding styles by using the tools Vera++ and KW Style. If any of the rules provided in this document are violated, the pipeline will fail.
- Compile: Checks if the project can be compiled successfully. If not this stage will fail. This job allows checking if the student has uploaded broken code and that it can compile correctly.
- Quality: This stage analyzes the quality of your code by means of different tools. The results of this stage are used by the supervisor to review the quality of your development. If the supervisors notices major issues or problems based on these reports, the student will have to fix them.
  - Complexity: Calculates the cyclomatic complexity of your project. In addition, it uses the tool cppclean to find any problems in your code.
  - Duplicates: Checks if there are any parts of your code that is duplicated. If the report finds duplications it probably means that you should simplify your code or create a function that can be called in different parts of your code.
  - Metrics: Reports metrics from your code such as number of lines, max indentation, number of comments.
  - Security: Checks for code that could cause security issues if not fixed. For example buffer overflows, stack overflow, and any other security weaknesses.
- Static Analysis: This stage uses the tools cppcheck, codechecker, and pvs studio to evaluate your code. Based on the rules provided in this document, it will provide a report and create an error if any of them are violated.
- Docs: Creates the Source code documentation based on Doxygen described in the section “Source code documentation”

If any of these stages fails, you can get a report of your mistakes and possible issues by downloading the artifacts created by the pipeline.



Duration: 19 seconds
Finished: in 34 minutes
Timeout: 1h (from project)
Runner: #82 (51trj7Bk) fh-shared-runner
Tags: shared

Job artifacts
These artifacts are the latest. They will not be deleted (even if expired) until newer artifacts are available.

Figure 1 How to download static analysis reports

Arduino Project

If your project is using the Arduino IDE or its SDK you can use any of the templates below to follow the guidelines of this document:

- Sketch: If you are developing a sketch, i.e., an application with Arduino you can use this template.
- Library: If you are developing an Arduino library, you can use this template.

ROS

If your project consists of a ROS package, you can use the following template to comply with these guidelines.

PlatformIO

If your project is using PlatformIO, you can use the following template to comply with these guidelines.

Other

If the C/C++ project you are working with is not described take a look at the other type of projects that are described above. They can be used as a reference to create a new template for a specific project that is not listed here. Please discuss with the supervisor of your project how you should incorporate it.