EIRENE coding and versioning rules – CONCEPT for the AD community “JuelichOrigin”

Last update: 24-06-24 (Pieter W. Groen)

Contents

[Introduction 2](#_Toc169689700)

[Rules concerning versioning (Git) 2](#_Toc169689701)

[Log for changes (‘change\_log.txt’ file) 3](#_Toc169689702)

[Merge requests 3](#_Toc169689703)

[Release Procedure 4](#_Toc169689704)

[Coding 4](#_Toc169689705)

[General 4](#_Toc169689706)

[Formatting 5](#_Toc169689707)

[Free format 5](#_Toc169689708)

[Capitals for Fortran keywords 5](#_Toc169689709)

[Format of constructs, procedures, functions, modules 5](#_Toc169689710)

[Derived type components 6](#_Toc169689711)

[File extensions 6](#_Toc169689712)

[.f and .F 6](#_Toc169689713)

[.f90, .F90 6](#_Toc169689714)

[Type names 6](#_Toc169689715)

[Variable names 6](#_Toc169689716)

[Procedure names 6](#_Toc169689717)

[Modules 7](#_Toc169689718)

[Module names 7](#_Toc169689719)

[Interfaces 7](#_Toc169689720)

[Best practices 7](#_Toc169689721)

[Compiler and preprocessor related 7](#_Toc169689722)

[Documentation 7](#_Toc169689723)

[In the code 8](#_Toc169689724)

[Outside of the code 8](#_Toc169689725)

[References 8](#_Toc169689726)

# Introduction

The **purpose** of this document is to state a set of coding rules for the development and maintenance of EIRENE, reference (1) inside the “JuelichOrigin” associated developer (AD) community. Moreover, this set of rules also states the code versioning, CI and documentation work flow. This AD community in charge of the reference EIRENE develop and master versions.

The **scope** of this set of rules mainly concerns the code that is either newly added or refactored. The bottom line being that *functionality must be preserved at any time*. We insist that any version committed and pushed to the repository even to a feature branch is at least compilable. For a merge request to the develop branch it is mandatory to have all CI tests and updates completed; moreover all the documentation must be up-to-date during such merge request. The latter of course includes any input/output changes, as well as the general idea of the update and references to the physical or mathematical basis – including a reference to the manual or literature (see “Log for changes” below). The rules listed below are strictly mandatory for all new files or large chunks of new code including formatting. On the other hand, the actions on general code updates, even “cosmetic” should be done with caution and in agreement with the whole AD community. For the refactored code parts one should see the rules below as “recommended if not affecting massively the repository history or doing some other potential harm to the work of other AD members”.

Rules concern the EIRENE code (as well as the code family related to EIRENE within the EIRENE repository, e.g. ModCR, and PLOUTOS) in general, whether it is to be used stand-alone or coupled with for instance the CFD codes line B2.5 (SOLPS-ITER package), Edge2D, EMC3 etc. One should respect the present guide by dealing with EIRENE itself and the interface parts as well as take care about the licenses and other rules related to the other package parts. For this reason, we aim to keep this guide aligned with, as much as possible, the SOLPS-ITER coding rules and other similar guidelines.

Some of the following rules are based on the CONTRIBUTING.MD of SOLPS-ITER (ref. (2)) and are adapted to the EIRENE specific situation. Also ref. (3) has been used as input to this document.

# Rules concerning versioning (Git)

1. When working on a new code feature or bug fix, one must work in a branch derived from a “develop” branch from the official distribution. It is necessary to keep that branch abreast of any changes occurring in the reference branch, either by merging or cherry-picking the appropriate commits, before submitting any pull requests. Doing so regularly decreases the amount of work required to inspect the pull request and ensures all relevant code updates can be introduced with minimal risk of errors.
2. We aim to maintain as full as possible CI coverage of the EIRENE. Any merge request introducing changes to the physics results of the code that would affect some of the provided reference examples must contain an updated set (including new ones added) of CI cases.  
   During the review of a merge request, the reviewer and developer(s) should follow this testing checklist and decide on ACTIONS to take for each of the related CI test cases:  
   1) DECIDE: is the test case still valid? (Does the code change change the assumptions?)  
   2) if the test case is no longer valid, DECIDE: should the test case be changed or removed?  
   3) if the test case should be changed, DEFINE it and IMPLEMENT it.  
   For each new feature introduced, if there are:  
   - new requirements  
   - changes to requirements  
   - untested assumptions in the codebase that the new code relies on  
   then:  
   1) DECIDE if a test case is needed for this requirement (or a changed one).  
   2) if so, DEFINE it and IMPLEMENT it.
3. When introducing a new input parameter or extending its functionality, a description of this switch/parameter must be added to the documentation files, within the same commit or pull request. Changes to the default values of switches/parameters (see Coding > General > rule 3) need to be explicitly mentioned in the commit message.
4. Private branches are allowed, but should be kept under their own sub-folder structure (e.g. ‘private\_uid\_or\_acronym/private\_branch\_name’). We recommend to name the branches with initials like “DVB/my\_new\_idea” and select the “delete branch” option when submitting the merge request.
5. Please commit often, even a few code lines. Also the merge requests should follow promptly – we should avoid long-term branching with painful effort on merging in the end. Each new feature should have an individual branch.

## Log for changes (‘change\_log.txt’ file)

Whenever a new feature or bug-fix is merged back to the official distribution branches (e.g. ‘develop’, ‘release’, ‘feature’…) an entry at the top of the ‘change\_log.txt’ file must be made. This makes it much easier for the Reviewer of the merge request to understand the new development. Also, any code-releases should be logged in this file, which makes it easier to compile ‘Release-Notes’ for the user community.

The entries in the ‘change\_log.txt’ file must contain:

1. The author(s) of the changes.
2. The git hash key of the last code commit.
3. Optionally the branch name as additional, but insufficient, information (as branches are not persistent in git), moreover we aim to delete most of the merged branches.
4. A detailed list of changes to the code since it branched off the official branch including:
   1. Purpose of the development.
   2. Bug fix or new feature (in standard form!).
   3. Any changes to existing or newly introduced input parameters.
   4. Any new output.
   5. Any new significant variables (in particular large structures and arrays, variable refactoring etc.)
   6. Expected changes to existing simulations and evaluation of impact. Mark the version as “Critical” if is expected to affect the results of a significant number of earlier simulations.
   7. Related CI cases, in particular if newly added.

See ‘Example\_change\_log\_from\_edge2d.txt’ file on INDICO as an example for a change-log used by EDGE2D.

## Merge requests

1. Whenever the developer requests a merge back to one of the official distribution branches (mainly the ‘develop’ branch), one must update the ‘change\_log.txt’ at the top of the file.
2. Merge requests should be as compact as possible. E.g., do not include three different features in one merge request, rather do three different merge requests. If merge requests build on top of each other mention this in the merge request to the Assignee and Reviewer.
3. Before the merge request, the developer should first merge any changes from the parent branch (which were committed since the moment he/she branched off) into his/her own feature branch and resolve conflicts.
4. If major new features are introduced, also a CI test-case must be provided to test these new features (preferably as computationally inexpensive as possible to keep the CI pipeline slim). The developer must provide updated test-cases for the CI if code results are changed, and make sure that the CI pipeline runs succesfully.
5. Whenever possible the changes should be ‘backwards compatible’ (e.g. default behavior of the code). If the changes are not ‘backwards compatible’ this must be clearly stated, so that this can be compiled in the next ‘Release Notes’ for the user community.
6. Any new or changed input switches or parameters must be reflected in the EIRENE documentation prior to the merge request.
7. The version should be updated using the “version.txt” file in the EIRENE root and the git hook.

## Release Procedure

1. Each Release must be tagged with an immutable tag. The format of the release-tag should always be the same, namely of the self-ordering type, using Semantic Versioning ("semver"), ref. (4): MAJOR.MINOR.PATCH, e.g. V1.2.1, V1.3.0, V1.3.1.  
   However, since tags are in principle free to use, *additional* separate tags can be used for comparison.
2. Bugfixes to releases should be tagged individually by an immutable tag (using *semver*).
3. For each release and bugfix, a “Release Note” must be compiled and distributed to the user-community.
4. Aim for at least one or two official releases per year.
5. The “develop” branch must be kept in a working state, so that it is good for productive simulations and for being a starting point for new branches at any time.

# Coding

The following concerns newly (i.e. with respect to Milestone version…) added code.

A number of sources may serve as an inspiration, e.g. the Google C++ Style Guide, ref. (5), but they do not necessarily reflect the choices of style and rules in this document.

## General

1. The code standards from 2003 are mandatory, keeping to 2018 standard is recommended. Using the more novel code constructions should be properly discussed with other ADs.
2. Code changes must preserve earlier functionality (reverse compatibility).
3. In order for variables, procedures, modules, etc. to be regarded as being in its own EIRENE 'namespace' the appropriate prefix or suffix must be used in their names. This avoids conflicts with variables, procedures, etc. of the same name in other (coupled) code(s). For instance all EIRENE module files should start with “eirmod\_”.
4. Code that only works for a particular set of cases, and is not general, should be avoided whenever possible and must always be identified as such. The code should then include safeties and/or error/warning messages to prevent its unintentional use.
5. When modifying the code equations or introducing a new physics term, this shall be reflected in the *physics model* description chapter of the EIRENE manual, within the same commit or pull request.
6. When introducing a new switch/parameter or extending its functionality, a description of this switch/parameter must be added to the documentation files. (In addition, see also Rules concerning versioning > rule 3).
7. *In-code* documentation: use Doxygen, see section Documentation.
8. Always use *explicit* type declarations of variables, making use of the 'IMPLICIT NONE' statement.
9. Variables that are closely related , should be organized, and grouped into derived types, using the TYPE keyword (see also references (6) and (7)).
10. At present, it is recommended to avoid or limit as much as possible the use of “OOP features” in Fortran, such as *Abstract types and deferred bindings* (despite these being in the Fortran 2003 standard) because ongoing work on Algorithmic Differentiation with the TAPENADE tool does not support these. If necessary, their use should be restricted to code parts not interfering with the main solution routines.

## Formatting

### Free format

As of Fortran 90, see also ref. (8):

1. 132 characters per line.
2. '&' line continuation character. Split long (how long?) lines with this character.
3. '!' comment initiator.
4. Significant blanks:
   1. indentation of 2 spaces in
      1. the body of modules (except the CONTAINS statement)
      2. the body of procedures, do-loops, if-statements, …
      3. the indentations are cumulative (so the body of a do-loop in a function in a module has an indentation of 6 spaces)
   2. spacing of routine arguments (in call and declaration)
   3. we recommend to use automatic formatter “findent”.

### Capitals for Fortran keywords

Use capitals for **all Fortran keywords**, e.g. PROGRAM, END, TYPE, IMPLICIT NONE, etc..

### Format of constructs, procedures, functions, modules

Format loops and procedures as follows:

1. DO  
   (…)  
   END DO
2. MODULE *name*  
   (…)  
   END MODULE *name*
3. SUBROUTINE *name* (argument list)  
   (…)  
   END SUBROUTINE *name*
4. *TYPE* FUNCTION *name* (argument list)  
   (…)  
   END FUNCTION *name*

### Derived type components

Components of derived types are selected by using the percent sign (%): e.g.

particle%velocity

(For some compilers the dot (.) component selector also works, but is not recommended, because of potential conflicts.)

## File extensions

### .f and .F

There is a distinction between the extensions: .f (lower case) and .F (upper case): upper case. The upper case files (\*.F) are files that have to be preprocessed (with a pragma directive/macros).

### .f90, .F90

…

## Type names

When defining a derived type (using the TYPE keyword), the type name should begin with 'ei\_', and end in '\_t' to identify it as being a derived type, e.g.

TYPE :: ei\_particle\_t

REAL (KIND = R8) :: mass

…

END TYPE ei\_particle\_t

## Variable names

1. Use clear names that tell what the variable does.
2. Do not use implicit variables.
3. Although Fortran is case insensitive, readability is enhanced when mixing cases (it also distinguishes variable names from keywords).  
   Use 'camelCase', e.g. “particleVelocity”.  
   Sometimes exceptions to this rule are justified.
4. Make variable names as explicit as possible, so rather 'particleVelocity' than 'pV' or 'partVel'. When names are getting too long, abbreviations should still result in clear names, so in that case rather 'partVel' than 'prtVl' for example.

## Procedure names

Subroutines and functions should start with the prefix '*eirene\_*'. Further categorization is possible using underscores, e.g. '*eirene\_user\_*' for user routines.

## Modules

All EIRENE module names should start wirh “eirmod\_”.

Separate functionality is put in modules that reside in the 'src/modules' directory.

Start each module with the IMPLICIT NONE statement. The file that imports the module uses the IMPLICIT NONE statement immediately after the USE *eirmod\_name* statement.

Whenever adding the use of modules in the code, explicitly state with “ONLY” which variables/functions are used from the module (“USE *eirmod\_name*, ONLY: *var\_name”* statement)). This avoids name-clashes and makes it much more understandable from where certain variables are originating (especially for new developers/users).

### Module names

Use a clear name, eventually with underscores, in lower case, starting with the prefix '*eirmod\_*' e.g. *eirmod\_json*.

Module names are included in the END MODULE statement.

### Interfaces

…

## Best practices

See also ref. (9).

1. Do **not** use
   1. GO TO statements
   2. COMMON blocks
   3. IMPLICIT variable typing.
2. Do not use IMPLICIT statements other than IMPLICIT NONE. All named constants, variables and functions should be explicitly typed, see ref. (9).  
   The IMPLICIT NONE statement should appear after the PROGRAM statement and before any type declaration statements.
3. Use explicitly INTENT attributes  
   INTENT(IN), INTENT(OUT), INTENT(INOUT)
4. When a module declares sharable data, the SAVE statement guarantees that that data is preserved between references in different procedures, ref. (9).  
   Although this can be beneficial, it may also lead to unwanted global behavior. Therefore the SAVE statement should *only* be used *when necessary*. When used, must be made explicit in the documentation (e.g. in an overview of all such variables).

## Compiler and preprocessor related

…

# Documentation

There is a distinction between documentation parts of the code aimed for developers and for its users (user manual). In some cases there is a thin line between the two, so that their scopes should be defined. The Manual is the main documentation for both. However, we don’t object using Doxygen by the developers for technical details. Any changes to the code, should be mimicked by the documentation – by the main “book-like” manual as well as Doxygen. We don’t insist on having Doxygen in every routine, however, in case it is provided it must be kept up-to-date – wrong document is even worse than none.

## In the code

Mainly for developers. Use Doxygen, ref. (10). This means that at least the following should be described … in the following format ….

From Doxygen, the following entries are generated:

* PDF …
* HTML code that is published via …

[DB: generally ok, but definitely needs to be more clear and detailed…]

## Outside of the code

Manual. Mainly for users. Input files, interface description, reference to parameters. Emphasis on the physics.

This LaTex documentation is part of the Eirene repository.

# References

1. **Reiter, D.** The EIRENE Code User Manual.

2. **ITER Organisation.** CONTRIBUTING.MD. December 18, 2023.

3. **Emil Løvbak, Xavier Bonnin, Oskar Lappi, Huw Leggate.** EIRENE formatting. 05-05-2023.

4. **https://semver.org/.**

5. **Google.** https://google.github.io/styleguide/cppguide.html#General\_Naming\_Rules.

6. **Gonzalez, Jorge.** Variable Grouping (Presentation Code Camp 2021).

7. —. Variable Grouping (Presentation Code Camp 2022).

8. **A, Marshall.** https://www.mrao.cam.ac.uk/~pa/f90Notes/HTMLNotesnode44.html.

9. **Chapman, S.J.** Fortran 95/2003 For Scientists and Engineers Third Edition. 2008.

10. **Doxygen.** https://www.doxygen.nl/.