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

Last update: 09-02-24 (Pieter W. Groen)

Contents

[Introduction 2](#_Toc158993868)

[Rules concerning versioning (Git) 2](#_Toc158993869)

[Log for changes (‘change\_log.txt’ file) 3](#_Toc158993870)

[Merge requests 3](#_Toc158993871)

[Release Procedure 4](#_Toc158993872)

[Coding 4](#_Toc158993873)

[General 4](#_Toc158993874)

[Formatting 5](#_Toc158993875)

[Free format 5](#_Toc158993876)

[Format of constructs, procedures, functions, modules 5](#_Toc158993877)

[File extensions 5](#_Toc158993878)

[.f and .F 5](#_Toc158993879)

[.f90, .F90 5](#_Toc158993880)

[Variable names 5](#_Toc158993881)

[Procedure names 5](#_Toc158993882)

[Modules 6](#_Toc158993883)

[Module names 6](#_Toc158993884)

[Interfaces 6](#_Toc158993885)

[Best practices 6](#_Toc158993886)

[Compiler and preprocessor related 6](#_Toc158993887)

[Documentation 6](#_Toc158993888)

[In the code 6](#_Toc158993889)

[Outside of the code 7](#_Toc158993890)

[References 7](#_Toc158993891)

**Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.**

# Introduction

The **purpose** of this document is to state a set of coding rules for the development and maintenance of EIRENE (1). Moreover, a set of rules is stated for the code versioning workflow.

The **scope** of this set of rules mainly concerns the code that is either added newly or refactored. The bottom line being that *functionality must be preserved at any time*. We insist that any version committed and pushed to repository even to a feature branch is at least compliable. For a merge request to a developer branch it is mandatory to have all CI tests and updates completed; moreover all the documentation must be up-to-date by such merge request. 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 in general, whether it is will 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 the EIRENE itself and the interface parts as well as take care about the licenses and other rules related to the other package parts.

# Rules concerning versioning (Git)

1. [*Example* (2)] When working on a new code feature or bug fix, it is important to work in a branch forked from a development branch from the official distribution. These are the `develop`, `release/3.1.0`, and `feature/wg-release` branches. It is necessary to keep that forked branch abreast of any changes occurring in the reference branch, either by rebasing, 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. [*Example* (2)] Any pull request introducing changes to the physics results of the code that would affect some of the provided reference examples must contain pointers to newly converged occurrences of these affected examples.
3. [*Example* (2)] When introducing a new switch or extending its functionality, a description of this switch must be added to the documentation files, within the same commit or pull request. Changes to the default values of switches (see Coding > General > rule 3) need to be explicitly mentioned in the commit message.
4. Private branches are allowed, but should be kept under its own sub-folder structure (e.g. ‘private\_uid\_or\_acronym/private\_branch\_name’)
5. …

## 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. Optional the branch name as additional but not sufficient information (as branches are not persistent in git)
4. A detailed list of changes to the code since it branched off the official branch including:
   1. Purpose of the development
   2. Any changes to existing or adding of new input parameters
   3. Expected changes to existing simulations and evaluation of impact.
   4. …?

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 (e.g. ‘develop’, ‘release’, ‘feature’…) he 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 to the official branch (which were committed since he branched off) into his own feature branch and resolve conflicts as well as test his development under these conditions thoroughly. (This is debatable. One could also argue that conflict resolving should be done by the Reviewer. But then the reviewer must eventually generate the updated test-cases for the CI…)
4. The developer must thoroughly test their new development before requesting the merge.
5. 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).
6. The developer must provide updated test-cases for the CI if code results are changed.
7. 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.
8. Any new or changed input switches must be reflected in the EIRENE documentation prior to the merge request.
9. …?

## Release Procedure

1. Each Release must be tagged with an immutable tag that never moves. The format of the release-tag should always be the same (the explicit format to be discussed) and preferably be self-ordering (e.g., V1.2.1, V1.3.0, V1.3.1, … or Release-v1.2.1, Release-v1.3.0, ….). Another possible self-ordering release tag could be originating from the release date (used in JINTRAC/EDGE2D) like ‘Release-vddmmyyyy’ or ‘Release-vdd.mm.yyyy’ (e.g., Release-v15.02.2024 and for a bugfix Release-v15.02.2024-1)
2. Bugfixes to releases should be tagged individually by an immutable tag (either by a new release-tag or special bugfix-tag)
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 state, so that anytime a new release could be generated.
6. …?

# Coding

The following first concerns newly (i.e. with respect to Milestone version…) added code. The core of EIRENE that is not to be altered to the new style is discussed in the subsection "Preserving legacy code" below.

## 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 functionality.
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).
4. [*Example* (2)] 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 by an unsuspecting user.
5. [*Example* (2)] If modifying the code equations or introducing a new physics term, this shall be reflected in the physics model description chapter of the SOLPS-ITER manual, within the same commit or pull request.
6. [*Example* (2)] When introducing a new switch or extending its functionality, a description of this switch must be added to the documentation files. (In addition, see also Rules concerning versioning > rule 3).
7. …
8. Code should be written in compliance with the following (set of) standard(s) (3):
   1. …
9. In code documentation: use Doxygen. See section Documentation.
10. When changing code, update the Doxygen, as well as possibly affected links to… and/or entries in the manual.

## Formatting

### Free format

As of Fortran 90, see also (4):

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. …

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

Use capitals and naming for the following:

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*

## 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 been preprocessed (converted) with a pragma directive/macro.

### .f90, .F90

…

## Variable names

1. Use clear names that tell what the variable does

## Procedure names

Subroutines and functions should start with the prefix '*eirene\_*'.

## Modules

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

Start each module with 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*.

### Interfaces

…

## Best practices

See also (5).

1. Do **not** use
   1. GO TO statements
   2. COMMON blocks
2. IMPLICIT statements.  
   Do not use IMPLICIT statements other than IMPLICIT NONE. All named constants, variables and functions should be explicitly typed (5).  
   The IMPLICIT NONE statement should appear after the PROGRAM statement and before any type declaration statements.
3. INTENT attributes  
   INTENT(IN), INTENT(OUT), INTENT(INOUT)
4. SAVE statements  
   In modules …

## Compiler and preprocessor related

…

# Documentation

There is a distinction between documentation of the code (under the hood, for developers) and its use (user manual). In some cases there is a thin line between the two, so that their scopes should be defined. Moreover In code and outside of code documentation is linked via… When changing code, update all affected documentation, as well as possibly affected links to… and/or entries in the manual.

## In the code

Mainly for developers. Use the Doxygen (6). 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 the physics.

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

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

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