A Developer Guide to the PROCESS Fusion Reactor Systems Code

The PROCESS Team: P. J. Knight, M. D. Kovari, H. Lux, J. Morris, S. I. Muldrew

Culham Centre for Fusion Energy/ United Kingdom Atomic Energy Authority
Culham Science Centre, Abingdon, Oxon, OX14 3DB, UK

15 July 2019 PROCESS version: 1.0.16

Contents

List of Tables

Chapter 1

Summary

This document is describing all aspects of the PROCESS code relevant to developers who add to the PROCESS code. Users who do not edit the PROCESS code do not need to read this document.

Chapter 2

Changing the Source Code: New Models, Variables and Constraints

It is often useful to add extra features to the code in order to model new situations. This chapter provides instructions on how to do this, with specific details on how to add various numerics related items to PROCESS.

Please remember to modify the relevant sections and table(s) in this User Guide if changes are made to the source code!

2.1 Source Code Modification

Described here are the general rules that apply when the Fortran source code is modified. See Section 3.4 for instructions on how to commit changes to the PROCESS Git repository and produce new releases. The variable descriptor file is generated from specially-formatted comment lines within the source code (see Section 3.3.2 for more details). Therefore, it is exceedingly important to keep these lines relevant and in sync with the variables they describe.

2.1.1 Changing the Fortran code

Please ensure that the following rules are adhered to when modifying the PROCESS source code:

1. New variables names should be as explicit as possible, even if they tends to get a bit long. More generaly follow the zen of python (you can find it in the following webpage `https://www.python.org/dev/peps/pep-0020/`).
2. New routines, sub-routines or functions should be defined with one or several unitarity tests, setup using the pFUnit test suite documented in `http://pfunit.sourceforge.net/`.
3. Keep the layout consistent in with the existing code, including indentation of clauses (`if`-statements, `do`-loops, etc.).
4. Use the standard routine header (see below).
5. Always use `implicit none` and declare all local variables explicitly.
6. Declare all 'real' (i.e. floating-point) variables as `real(kind(1.0D0))`.

7. Ensure all routine arguments have the appropriate attribute `intent(in)`, `intent(out)` or `intent(inout)`, as necessary.

8. Always write explicit real constants using the scientific `D` notation, e.g. `1.0D0`, `2.3D0`, `-1.23D6`. That is to say, use `1.0D0` and not `1.0` or `1.` or `1` when the expression should be using floating-point arithmetic.

A Fortran 90/95 manual, complete with guidelines for good Fortran 90/95 practice, may be found at the following webpage:

`http://fusweb1.fusion.ccfe.ac.uk/~pknight/f95notebook.html`

2.1.2 Source code documentation

It is critically important to keep the documentation in the source code itself up-to-date, relevant and tidy. Please keep to the following guidelines whenever the source code is modified.

1. Use comments copiously in the code, avoiding useless comments.

2. If a used equation is extracted from a published paper, refer explicitly to it, citing the paper and equation number when possible (precise the page if no equation number is available,). For example :
   `!! Ref eq : J. Johner, Fusion Science and Technology 56 (2011) 308-349, eq(18)`
   For equations taken from the PROCESS user manual, just indicate the page and the paragraph as the equation numbering might change with document updates.

3. Use the standard header layout, and do not omit any of the sections. Here is an example subprogram header:

```
! !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

subroutine culblm(bt,dnbeta,plascur,rminor,betalim)

  !+ad_name  culblm
  !+ad_summ  Beta scaling limit
  !+ad_type  Subroutine
  !+ad_auth  P J Knight, CCFE, Culham Science Centre
  !+ad_cont  N/A
  !+ad_args  bt      : input real :  toroidal B-field on plasma axis (T)
  !+ad_args  dnbeta  : input real :  Troyon-like g coefficient
  !+ad_args  plascur : input real :  plasma current (A)
  !+ad_args  rminor  : input real :  plasma minor axis (m)
  !+ad_args  betalim : output real : beta limit as defined below
  !+ad_desc  This subroutine calculates the beta limit, using
  !+ad_desc  the algorithm documented in AEA FUS 172.
  !+ad_desc  <P>The limit applies to beta defined with respect to the total B-field.
  !+ad_desc  Switch ICULBL determines which components of beta to include (see
  !+ad_desc  routine <A HREF="constraints.html">constraints</A> for coding):
  !+ad_desc  <UL>
  !+ad_desc  <P><LI>If ICULBL = 0, then the limit is applied to the total beta
  !+ad_desc  <P><LI>If ICULBL = 1, then the limit is applied to the thermal beta only
  !+ad_desc  <P><LI>If ICULBL = 2, then the limit is applied to the thermal +
  !+ad_desc                        neutral beam beta components
  !+ad_desc  </UL>
  !+ad_desc  The default value for the g coefficient is DNBETA = 3.5
```

```
!+ad_prob  None
!+ad_call  None
!+ad_stat  Okay
!+ad_docs  AEA FUS 172: Physics Assessment for the European Reactor Study
!+ad_docs  AEA FUS 251: A User's Guide to the PROCESS Systems Code
!
! !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

A description of all the available automatic documentation marker tags (these all start with `!+ad_`) may be found by examining the main program header of the (self-documenting!) automatic documentation program itself (in file `autodoc.f90`).

4. Ensure that all routines called are listed via `!+ad_call` lines. Update the description lines as necessary. You may use html tags and hyperlinks (some are shown in the example above) as required; to be sure that they have been added correctly, type `make html` to create the web documentation and examine the relevant html file (i.e. `culblm.html` for the example above) using your favourite web browser.

5. If you add a new routine to a module, remember to modify the header of the module as well as that of the new routine (add a `!+ad_cont` line to it).

6. Add suitable documentation to this User Guide whenever a model is added or modified. This should be done immediately to ensure that the Guide remains consistent with the source code. Change the code's revision number and the date in `process.tex`.

7. Add a new file to the folder `release_notes`. It is particularly important to describe here any changes to the required IN.DAT - especially any change that makes previous IN.DAT files unusable.

2.2 Input Parameters

Input parameters (see User-guide) are added to the code in the following way:

1. Choose the most relevant module (usually one of those in source file `global_variables.f90`). Keeping everything in alphabetical order (or possibly within a group of variables closely-related to a particular switch), add a declaration statement for the new variable, specifying a "sensible" default value, and a correctly formatted comment line to describe the variable. Copy the examples already present, such as

```
!+ad_vars  abktflnc /5.0/ : allowable first wall/blanket neutron
!+ad_varc                   fluence (MW-yr/m2) (blktmodel=0)
real(kind(1.0D0)) :: abktflnc = 5.0D0
```

Note that the automatic documentation marker tag `!+ad_vars` tells the `autodoc` utility (Section 3.3.2) that the line is (the first line of) a variable description, while `!+ad_varc` specifies any continuation lines. Also note that the colon (:) on the first line is necessary, as it is assumed to exist by the dictionary-building Python utility for the GUI.

2. Ensure that all the modules that use the new variable reference the relevant module via the Fortran `use` statement.

3. Add the parameter to routine `PARSE_INPUT_FILE` in source file `input.f90` in a suitable place — keep to alphabetical order. The existing examples provide guidance on how to do this. Note that real (i.e. double precision) and integer variables are treated differently, as are scalar quantities and arrays.

4. Input variables names should be as explicit as possible, even if they tends to get a bit long.

2.3 Iteration Variables

The format for defining iteration variables has CHANGED. These are added in the same way as input parameters, with the following additions:

1. The name of an iteration variable must not be more than 14 characters long. The variable name should be as be as explicit as possible within this constraint.

2. The parameter `ipnvars` in module `numerics` in `numerics.f90` will normally be greater than the actual number of iteration variables, and does not need to be changed.

3. Utilise the next available block of code in module `define_iteration_variables` in `numerics.f90`.

4. Assign values for the variable's lower and upper bounds to the relevant elements in arrays `boundl` and `boundu`.

5. Paste the variable name in the relevant places in the code block in place of the word 'DUMMY'.

6. Ensure that the relevant element of character array `lablxc` is exactly 14 characters long.

7. If the variable is declared in a module not listed at the the top of `module define_iteration_variables` then it is best to place the required use statement in the relevant function `itv_XX` and subroutine `set_itv_XX)`.

It should be noted that iteration variables must not be reset elsewhere in the code. That is, they may only be assigned new values when originally initialised (in the relevant module, or in the input file if required), and in the `subroutine set_itv_XX` where the iteration process itself is performed. Otherwise, the numerical procedure cannot adjust the value as it requires, and the program will fail.

2.4 Other Global Variables

This type of variable embraces all those present in the modules in `global_variables.f90` (and some others elsewhere) which do not need to be given initial values or to be input, as they are calculated within the code. These should be added to the code in the following way:

1. The variable name should be as be as explicit as possible.

2. Choose the most relevant module (usually one of those in source file `global_variables.f90`). Keeping everything in alphabetical order (or possibly within a group of variables closely-related to a particular switch), add a declaration statement for the new variable, specifying the initial value `0.0D0`, and a correctly formatted comment line to describe the variable (copying the examples already present — see also "Input Parameters" above).

3. Ensure that all the modules that use the new variable reference the relevant module via the Fortran `use` statement.

2.5 Constraint Equations

Constraint equations (see User-Guide and vardes.html) are added to `PROCESS` in the following way:

1. Increment the parameter `ipeqns` in module `numerics` in source file `numerics.f90` in order to accommodate the new constraint.
2. Add an additional line to the initialisation of the array `icc` in module `numerics` in source file `numerics.f90`.
3. Assign a description of the new constraint to the relevant element of array `lablcc`, in module `numerics` in source file `numerics.f90`.
4. Add a new Fortran `case` statement containing the new constraint equation to routine `CONSTRAINT_EQNS` in source file `constraint_equations.f90`, ensuring that all the variables used in the formula are contained in the modules specified via `use` statements present at the start of this file. Use a similar formulation to that used for the existing constraint equations, remembering that the code will try to force `cc(i)` to be zero.

Remember that if a limit equation is being added, a new f-value iteration variable may also need to be added to the code.

2.6 Figures of Merit

New figures of merit (see User-Guide and vardes.html) are added to `PROCESS` in the following way:

1. Increment the parameter `ipnfoms` in module `numerics` in source file `numerics.f90` to accommodate the new figure of merit.
2. Assign a description of the new figure of merit to the relevant element of array `lablmm` in module `numerics` in source file `numerics.f90`.
3. Add the new figure of merit equation to routine `FUNFOM` in source file `evaluators.f90`, following the method used in the existing examples. The value of `fc` should be of order unity, so select a reasonable scaling factor if necessary. Ensure that all the variables used in the new equation are contained in the modules specified via `use` statements present at the start of this file.

2.7 Scanning Variables

Scanning variables (see User-guide and vardes.html) are added to `PROCESS` in the following way:

1. Increment the parameter `ipnscnv` in module `scan_module` in source file `scan.f90` to accommodate the new scanning variable.
2. Add a short description of the new scanning variable to the `nsweep` entry in source file `scan.f90`.

3. Add a new assignment to the relevant part of routine `SCAN` in source file `scan.f90`, following the examples already present, including the inclusion of a short description of the new scanning variable in variable `xlabel`.

4. Ensure that the scanning variable used in the assignment is contained in one of the modules specified via `use` statements present at the start of this routine.

2.8 Submission of New Models

The `PROCESS` source code is maintained by CCFE, and resides in a *Git* [1] repository on the CCFE servers. We welcome contributions of alternative or improved models and algorithms.

We request that contributors provide the following information for any new models that they provide:

- The name of the fortran files should be as explicit as possible and in agreement with the GUI structure. The following convention should be used: `moduleName.f90`, with `moduleName` being the name used in the GUI. If several `.f90` files are associated to one GUI module, identify the main `.f90` file, and name the other files `moduleName_libName.f90` with `libName` being the chosen name for the library.
- A comprehensive description of the model; please provide a full list of references.
- A list of all inputs and outputs: descriptions, default (input) values, allowed ranges, units.
- If possible, please cross-reference any input/output variables to existing global variables listed in the variable descriptor file (see User-guide and vardes.html).
- Any new input parameters, iteration variables, constraint equations, figures of merit etc.
- A definition of any pre-requisites.
- As many unitary test should be defined as possible, using the pFUnit test suite documented in `http://pfunit.sourceforge.net/`.
- Any available test data, code examples or test programs in any language.

2.9 Code Structure

2.9.1 Directory structure

The folder structure for the PROCESS system prior to compilation is described below:

```
*-- CMakeLists.txt                    : Build and compile files
*-- GNUmakefile                       : Build and compile pFUnit files
+-- lib                              : Libraries used in PROCESS
|   +-- PLASMOD                       : PLASMOD lib files
+-- source                           : source files
|   +-- Fortran                       : Fortran source files
|   +-- cpp                           : C++ source files
+-- test_suite
|   *-- ci_test_suite.py              : Python file for running test suite in Continu
```

```
|   *-- ci_test_suite_functions.py      : Python functions  for running test suite in C
|   *-- test_suite.py                   : Python file for running test suite by user or
|   *-- test_suite_functions.py         : Python functions  for running test suite by u
|   +-- test_files                      : Input files for test suite
|   +-- test_area                       : Output files for test suite
+-- unit_tests
|   +-- pfunit_files                    : pFUnit test files
|   +-- gtest_files                     : GTest test files
+-- utilities/                          : Python utilities files
+-- fispact/                                : fispact Data file
+-- data                                : Data files
|   +-- fluids
|   +-- h_data
|   +-- lz_non_corona
|   +-- lz_non_corona_14_elements
+-- documentation                       : Contain documentation files
```

2.9.2 Numerics modules

These modules contain the equation solvers, their calling routines and other relevant procedures. Various mathematical routines from a number of standard libraries are also incorporated into these files. Table 2.1 summarises the numerics source files.

source file	description
caller.f90	calls physics, engineering, building and cost routines
constraint_equations.f90	defines the constraint equations
evaluators.f90	function evaluators for HYBRD and VMCON packages
iteration_variables.f90	adjusts values of iteration variables
maths_library.f90	miscellaneous 'black-box' maths routines, including HYBRD and VMCON
numerics.f90	numerics array definitions, and calling routines for HYBRD and VMCON packages
quanc8.f90	8 pannel newton cotes integration function
scan.f90	performs a parameter scan

Table 2.1: *Summary of the numerics modules in PROCESS.*

2.9.3 Physics modules

These modules contain the main physics routines that evaluate the plasma and fusion parameters. Also included here are the routines describing the current drive and divertor systems. Table 2.2 summarises the main physics source files.

2.9.4 Engineering modules

These modules contain the description of the machine geometry and its major systems, including the PF and TF coil sets, the first wall, blanket and shield, and other items such as the buildings, vacuum system, power conversion and the structural components. Table 2.3 summarises the main engineering source files.

source file	description
`current_drive.f90`	current drive efficiency calculations
`divertor.f90`	Kukushkin/Harrison divertor model
`divertor_ode.f90`	Kallenbach divertor model (1D)
`fispact.f90`	nuclide inventory/activation calculations
`hare.f90`	ECCD current drive using HARE
`ife.f90`	Inertial Fusion relevant physics/engineering
`impurity_radiation.f90`	radiation power calculations
`physics.f90`	tokamak plasma and fusion calculations
`physics_functions.f90`	plasma physics parameters calculation called by `physics_functions.f90`, `stellarator.f90` and `plasmod.f90`
`plasma_geometry.f90`	plasma geometry algorithms
`plasma_profiles.f90`	plasma density and temperature profile calculations
`plasmod.f90`	interface with the PLASMOD transport code
`reinke_module.f90`	Reinke minimum impurity fraction for divertor protection calculation
`startup.f90`	plasma start-up auxiliary power requirements
`stellarator.f90`	stellarator-relevant physics/engineering

Table 2.2: *Summary of the physics modules in* PROCESS.

source file	description
`availability.f90`	Plant component lifetimes and overall availability
`buildings.f90`	Buildings calculations
`fw.f90`	First wall calculations
`hcpb.f90`	HCPB blanket and shield calculations
`hcll.f90`	HCLL blanket and shield calculations
`machine_build.f90`	Machine build calculations
`pfcoil.f90`	PF coil module
`plant_power.f90`	Heat transport and power balance calculations
`pulse.f90`	Pulsed power plant calculations
`safety.f90`	Steady-state temperatures after a LOCA event
`sctfcoil.f90`	Superconducting TF coil module
`stellarator_fwbs.f90`	Stellarator HCPB breeding blankets
`structure.f90`	Support structure calculations
`superconductors.f90`	Supraconductor properties
`tfcoil.f90`	Resistive TF coil module
`vacuum.f90`	Vacuum system calculations

Table 2.3: *Summary of the engineering modules in* PROCESS.

2.9.5 Costing module

Three cost models are available:

- the 1990 one contained in `costs.f90` performs all the cost calculations, including values in M$ for each machine system, and the cost of electricity in m$/kWh. Normally, the machine costs are written to the output file; if this is not required set switch `output_costs = 0`.
- The Kovari 2015 one, contained in `costs_2015.f90`, provides only the capital cost.
- A new cost model under development in `costs_step.f90`

2.9.6 Specific modules libraries

Several modules are only called by specific modules to load dedicated physical data or mathematical tools. Here a list of theses modules loaded by:

- Hare module : `currn.f90`, `TorGA_curgap.f90` and `green_func_ext.f90`
- Kallenbach module : `kallenbach_module.f90` (testing libraries), `read_and_get_atomic_data.f90`, `read_radiation.f90`
- Reinke module : `read_radiation.f90`
- Supraconductor TF module : `ode.f90`

2.9.7 Other modules

These modules perform miscellaneous tasks, such as initialisation of variables and file input / output. File `process.f90` contains the main program, and includes the overall controlling loop.

Table 2.4 summarises these modules.

source file	description
`autodoc.f90`	Automatic html `PROCESS` documentation generation
`error_handling.f90`	centralised error handling module
`fson_library.f90`	library used to read in data from JSON-format files
`global_variables.f90`	defines and initialises most shared variables
`initial.f90`	checks self-consistency of input variables and switches
`input.f90`	reads in user-defined settings from input file
`output.f90`	utility routines to format output to file
`process.f90`	main program and top-level calling routines

Table 2.4: *Summary of the remaining modules in `PROCESS`.*

Chapter 3

Code Management Tools

This chapter will be of interest to people involved in the continuing maintenance of the PROCESS source code. As stated elsewhere, the source code is maintained by CCFE, and resides in a Git repository on the CCFE servers.

3.1 Initial access to the source code

To gain access to the PROCESS source code Git repository, you need to be given permission to do so via the CCFE GitLab server.

1. Use a web browser to go to `http://git.ccfe.ac.uk`
2. Login using your normal CCFE computer login details.
3. Assuming this is successful, contact the GitLab PROCESS "Owner" (currently James Morris james.morris2@ukaea.uk), who will add you to GitLab as a PROCESS "Developer".
4. If this is your first access to GitLab, you may have to set up SSH keys. To do so, get back to `https://git.ccfe.ac.uk/` and click on your profile on the top right corner, it will show a menu where you will have to click on setting. On the left menu then click on SSH Keys and simply follow the instructions.

   click on on your profile picture
5. Login to a Fusion Unix Network machine.
6. `cd SomewhereAppropriate` (you choose!)
7. `git clone git@git.ccfe.ac.uk:process/process.git -b develop my_develop` This copies the `develop` branch of the repository into a local folder `my_develop`, which will be created if it does not already exist.
8. `cd my_develop`
9. `git checkout develop`

   The sequence of commands above will provide you with a full copy of the `develop` branch of the PROCESS source code, Python utilities and all the documentation files.

3.2 Environment set-up

Please note that this section is only relevant for people actually developing the PROCESS code, i.e. if you are interested in running a copy of the code from your local directory on the fusion Unix network. For normal users please refer to the User-guide.

To compile PROCESS you must simply have:

- gfortran to compile the main PROCESS souces. The compilation will not work with ifort
- gcc to compile the PROCESS GUI
- python to run the PROCESS utilities.

If you are developing from the Freia CCFE cluster, please use the following commands to setup the environment for compilation and documentation generation:

```
module unload ifort
module unload pgi
module load gfortran
module load texlive/2017
```

It can be convenient to add the following lines to your .bashrc file[1], to avoid retyping them for each compilations.

If you are working from your local machine, you will have potentially to install the gcc and gfortran compilers manually. You will also have to install cmake and the google profiling software. For Linux Ubuntu users, it can be simply done using:

```
sudo apt-get install gfortram
sudo apt-get install gcc
sudo apt-get install cmake
sudo apt-get install googletest
sudo apt-get install pdflatex
sudo apt-get install bibtex
```

Additional steps must be followed to install google test. Please gently ask J. Morris for this, he is the one!

3.3 The CMakefile

To improve the portability of the PROCESS code to different operating systems (Linux, Windows or Mac), its compilation is now done using cmake. The compilation, html code documentation, pdf manual generation, unitary test and profiling instructions are contained in the CMakeLists.txt file. This has proved to be of great benefit in keeping all of the data from a given run together for archival purposes. Different verbosity level options can be set.

[1] This is a file in the user's home directory, assuming the bash Unix shell is being used. Although hidden, it can be opened by issuing the relevant command, for example gedit .bashrc.

3.3.1 Compilation

The compilation on the Freia cluster can be done using CMake, with the following commands:

```
cmake3 -H. -Bbuild
cmake3 --build build
```

The first cmake command creates the folder and setup the compilation options, the second triggers the actual `PROCESS` compilation. At the end of the compilation, a binary files folder `bin/` is created containing two executables `process.exe`, `process_GTest.exe` allowing to run the `PROCESS` code and its unitarity test, respectively and a sheared object (`.so`, called DDL in the windows environment), containing most of the `PROCESS` code.

To show all compiler warnings (`-Wall` and `-Wextra`), add the `-Ddebug=ON` option flag to the first cmake command. To allow unitary tests on functions using local fortran variables, the `PROCESS` code is compiled as a DLL (Dynamic Linl Libraries, `.so` files in C++). It can nevertheless be compiled as a single executable, adding the `-Ddll=OFF` option to the first cmake command.

Several options on the second cmake command () can be used to:

- Generate the python dictionaries: `--target dicts`
- Generate the pdf documentation: `--target doc`.
- Generate the html documentation: `--target html`.

To clean build directory, simply remove it:

```
rm -fr build
```

If the build is done on your local machine, use the same instruction replacing `cmake` by `cmake`[2].

3.3.2 Automatic Documentation

The `PROCESS` source code is self-documenting to a degree, using an included parser program (`autodoc`) to generate html files for each subprogram from specially-formatted comment lines within the code. It is the responsibility of the programmer to keep the `autodoc` comments within the source code relevant, comprehensive and up-to-date! Use the examples in the code as a starting basis for new routines; the output section corresponding to the various `autodoc` tags should be self-explanatory. See also Section 2.1.

The following files are used:

```
autodoc.f90
adheader.src
adfooter.src
```

[2]cmake3 is an alias used on the Freia cluster to precise the cmake version (3)

To create the ($\sim$ 526) html files from the source code, type

```
cmake3 --build build --target html
```

This command will fill the following directory `documentation/html/` with one html file per fortran source code contained in `source/fortran` directory. Any html file can be opened by your favourite web browser. For example, to open the documentation of the main `PROCESS` function just use

```
cd documentation/html
firefox process.html
```

3.3.3 LaTeX documentation

In addition, a full LaTeX documentation is and have to be maintained by the `PROCESS` developpers:

- A user guide contained within `process.tex`
- A developper guide contained within `developerguide.tex`
- A description of the `PROCESS` solvers contained within `optsolverdoc.tex`
- A description of the `PROCESS` magnets modules contained within `optsolverdoc.tex`
- A description of the `PROCESS` python utilities contained within `utilitiesdoc.tex`

To generate the .pdf files, simply execute the following command:

```
cmake3 --build build --target doc
```

All these files must be maintained in strict agreement with the evolution of the `PROCESS` code. It is the developper responsability to update them when a change is commited in the different source codes.

3.3.4 Unitary test

A fortran unitarity test framework pFUnit, is implemented in the `PROCESS` code. A documentation of this tool is available in `http://pfunit.sourceforge.net/`. The unitary test are configured using `*.pf` files contained in the `test_files/pfunit_files/` folder. The unitarity test are executed using:

```
cmake3 --build build --target test-build
```

IS THE COMMAND RIGHT ??

3.4 Code Updates and Release Procedure

This section describes the procedures that should be followed whenever new commits to the `develop` or `master` branches of the `PROCESS` Git repository are to be made, and how new code releases are performed. It is assumed that readers have a working knowledge of Git commands.

3.4.1 Git workflow

The code management methodology is based on the so-called "gitflow" workflow. There are two main branches:

- `develop`, which is the basis for all code development work, and changes are committed to it frequently.
- `master`, which contains official "release" versions of `PROCESS`, and is updated on a roughly three-monthly timescale.

Development work on new models should use separate branches, named e.g. `dev_mynewmodel`, split from the `develop` branch. It is a very good idea to merge the `develop` branch into `dev_mynewmodel` frequently during the course of the work, so that changes to the main branch are transferred to the new model's files with minimal effort.

Once the new model has been finished and tested successfully (complete with full documentation — see Section 2.1), the branch should be merged back into the `develop` branch.

Note that typically development branches do not need version tagging as they should not be used in any production runs. However, e.g. for the DEMO baseline design a separate branch will be created to conserve the `PROCESS` output in the state when the baseline was fixed.

3.4.2 Tagging

Tagging in `PROCESS` is used to document the version of the code any individual run has been performed with. This is necessary for proper provenance capture. Any version produced after 2016 takes the form x.y.z for internal development versions and takes the form x.y for external master releases:

[major version].[minor version].[revision number]

with

- [major version] - release containing numerous major changes
- [minor version] - medium change, i.e. new model, major bug fix
- [revision number] - weekly or on demand build/change

Any development versions ending in .0 should be fully tested as they correspond to the public ones. For example the version 2.4 of the master version must corresponds to the 2.4.0 of the development version.

3.4.2.1 Tagging in a separate branch

This section covers tagging in e.g. the separate DEMO1 baseline branch and assures that while this branch can develop independently of the development branch, it still has unique version tagging and provenance capture. Tagging of a separate branch should be started when it is forked from develop. Hence, the initial tag should be

[BID]_[major version].[minor version].[revision number]

where [BID] is a short but informative branch identifier consisting of letters only. This allows for branching from the development version at any point. Ideally it does start from a fully tested version though!
Manoj, please check that both characters as well as the new dot format are possible options for the autodoc tool!

3.4.2.2 Tagging between manual tags

Between user tags Git will create tags in the following format:

`1.0.12-11-g3f1b433`

The parts are:

- `1.0.12` is the last manually entered tag by the user
- `11` is the number of commits since that tag
- `g3f1b433` is a unique identifier for this specific commit

This allows the user to checkout a specific commit between tagged versions. PROCESS now outputs this information into the 'OUT.DAT' and 'MFILE.DAT' and is updated upon compilation. This way each output file is trackable to a specific commit.

3.4.3 Git manipulations

The git manipulations can be made with command lines or through the VS code editor that proposes many convenient features for Git. The VS code editor can be downloaded at `https://code.visualstudio.com/` for any platforms. Many extensions are available, we strongly recommand to unstall the fortran, C++, python, cmake and latex one are these languages are used in `PROCESS`.

3.4.3.1 Commit logs

To see the commit messages you can use the `git log` command. There are various options described in table 3.1:

3.4.3.2 Branching from `develop`

The following commands create a new branch in which to work on a new model. We assume that you are already in the `process` directory created above (Section 3.1).

1. `git checkout develop` : to ensure that this is the branch *from which* you will be branching.
2. `git branch dev_mynewmodel` : Creates an empty local branch called `dev_mynewmodel`.
3. `git checkout dev_mynewmodel` : load the branch that has been checked out in the `dev_mynewmodel` branch. This is a *local* manipulation.
4. `git push origin HEAD` : updates the central `GitLab` repository. The created branch will appear in the `PROCESS` Git webpage

Command	Description	Example
`-(n)`	show the last `n` commits	`git log -5`
`--since` or `--after`	limits the logs to be from date given can use `number.scale` where scale=year/month/week/day/minute	`git log --since "21-01-15"` `git log --since 2.weeks`
`--until` or `--before`	limits the logs to be up to date given	`git log --until "22-01-15"`
`--author`	only shows commits from given author	`git log --author "morrisj"`
`--grep`	only show commits with a commit message containing the string given	`git log --grep "magnet"`
`--stat`	if you want to see some abbreviated stats for each commit	`git log --stat`
`--oneline`	Outputs commit number, date and message to a single line	`git log --oneline`
`--graph`	display commits in a ASCI graph	`git log --graph`
`-S`	only show commits adding or removing code matching the string	`git log -S "find_me"`

Table 3.1: `git log` option description.

3.4.3.3 Working on a new model

Edit your local copies of the files as necessary. A convenient free editor is Visual Studio (`https://visualstudio.microsoft.com`) as it has handful integrated Git property (easy commit, easy diff, integrated grep command etc..). Whenever you want to save the changes back to the repository. It is prudent to do this at the end of each working day, as well as when the changes are complete, the code compiles and all test runs are successful. To perform a user commit, use the following commands:

1. `git status` (this will show a list of modified files)
2. `git add changedfile1 changedfile2 ...`
   (This 'stages' the modified files marking them as ready for committing).
   `git commit`
   Alternatively, just use
   `git commit -a`
   This commits all modified tracked files.
3. An editor window will open; add a line summarising the changes you have made, save and close the window. (Do not use quotation markes in your message.) This will initiate the commit to your *local* copy of the repository.
4. An alternative is to type
   `git commit -a -m 'Type message here'`
   This commits all modified tracked files, and adds the message entered between single quotes as shown. No editor will open. Do not use any quotation marks inside the message.
5. `git push`
   Copies the changes you have made locally to the version in the central `GitLab` repository. This uses a merging process, but if no-one else has changed your branch then the central version will simply become a copy of your local version.

If VS is used as text editor it much more convenient to do:

1. Go to Source Control clicking the corresponding icon on the left panel (a Y with three empty circles on the ends) or using the following shortcut `Ctrl-Shift-G`. On the left panel will appear the list of all the files that :
   - has been modified since the last committed version with a M sign on the right of the filename,
   - has been added but untracked by Git with a U sign
   - has been added and taken into account by the Git repository with a A sign.
2. You can have a look at the differences of the modified files simply clicking on it on the list, it will show a dual text editor with on the left the last committed version and on the right the current file version, highlighting all the difference between the two!
3. Commit the file version to your branch. For this simply do
   - Verify if all the changes you made are compiling and executing flawlessly with meaningful physical result
   - Write a detailed and explicit description of the chances you are about to commit
   - Point your mouse on the file you want to commit and click on the + sign that appears to select the file you want to commit.
   - Use the `Ctrl+Enter` shortcut to commit.
4. Update your branch is the common `PROCESS` repository (`PROCESS` Git webpage) using `git push --set-upstream origin dev_mynewmodel`, with `dev_mynewmodel` being the name of your developing branch.

3.4.3.4 Merging `develop` into working branch

It is a good idea to periodically merge the `develop` branch into the branch in which you are working, to ensure that any changes made in `develop` are included in your working branch.

1. Firstly, make sure you have committed your latest changes into the central `GitLab` repository as described in the previous section. Set the directory containing the working branch as your current directory.
2. `git pull`.
   Merges the version in the central repository into the local branch by copying over any changes that have been made in the version stored centrally.
3. `git checkout develop`
   This switches the "current branch" to `develop`.
4. `git pull`
   Updates the current branch of the local repository to the same branch on the central repository.
5. `git checkout dev_mynewmodel`
   This switches the "current branch" to `dev_mynewmodel`.
6. `git merge develop`
   Merges `develop` into the `dev_mynewmodel` branch.

7. Look for messages on the screen containing the word "conflict" indicating that some files cannot be merged directly. This typically happens if the same (or very closely-spaced) lines have been edited in both the `develop` and `dev_mynewmodel` branches.

   If any files are affected, they will be listed. Edit them and look for any lines containing `=======`. Such lines separate the changes made in the two branches, as in:

   ```
   <<<<<<< HEAD
   This line was edited in dev_mynewmodel branch
   =======
   This line was edited in develop branch
   >>>>>>> develop
   ```

   Resolve the conflict(s) as necessary. Then type `git add file1 file2 ...`, where `file1` etc. are the names of the files you removed conflicts from. Finally type `git commit` and edit the change log file.

8. `git push`: Update the central repository.

3.4.3.5 Committing changes to `develop`

Whenever a commit to the `develop` branch is to be made, the following procedure should be followed. Ensure all documentation is up to date (see Section 2.1) and the code is fully tested.

1. In routine `inform` of file `process.f90`, change the definition of `progver` by incrementing the last digit of the revision number by one e.g. from `1.3.26` to `1.3.27` for each minor commit or the second git for each fully tested major model that is included while setting the last digit to 0 e.g. from `1.3.26` to `1.4.0`. Furthermore, update the Release Date. It is important to keep exactly the same format.

2. Add a brief comment to the bottom of source file `process.f90` describing the changes made since the last commit in the same branch. Start the line with `! GIT XYZ:` , following the existing examples.

3. If any of the User Guide `.tex` files have been modified, edit the definition of `\version` in `process.tex` by changing the Revision (to e.g. `1.3.27`) and the date.

4. If you have changed any "use" statements in the code, or any compilation dependencies in the Makefile, run
   `make clean`

5. Check the code compilation and the html code documentation in a verbose mode

   ```
   cmake3 -H. -Bbuild -Ddebug=ON
   cmake3 --build buid --target html
   cmake --build buid
   ```

6. Run the input file(s) in the `tests` folder to ensure PROCESS runs correctly.

7. Close all opened editor windows. The commit will not work otherwise.

8. `git commit -a -m 'Type your message here'`
   This commits all modified tracked files. (Do not use quotation marks inside your message.)

   Alternatively, you can do this in two steps:
   `git add process.f90 process.tex changedfile1 changedfile2 ...`
   (This 'stages' the modified files marking them as ready for committing).
   `git commit`

9. An editor window will open; add a line summarising the changes you have made. Use a format like this:

   ```
   1.3.27   A summary of the changes made
   Further details. Changes due to Git issues can be described like this:
   #270    Description
   #273    Description
   ```

   Save and close the window. This will initiate the commit to your *local* copy of the repository.

10. `git tag -a 1.3.27 -m 'Revision 1.3.27'`

11. `git push`

12. `git push origin 1.3.27`

The instructions given in Section 3.4.4 should now be followed to make the new `develop` release available to all users.

3.4.3.6 Merging develop into master

When merging develop into master, please note that the tagging changes as described in section 3.4.2.

3.4.4 Full code rebuild

The standard `PROCESS` executables and the corresponding documentation available to all users are stored in the functional account called `PROCESS` on the CCFE Fusion Unix Network. Whenever the `master` or `develop` branches are updated a full rebuild of the standard executables and documentation should be performed. This is done as follows:

1. Ensure that all key users have been informed (using the commit message described above or directly).

2. `alter PROCESS` (this changes your current login to that of the `PROCESS` user; only registered individuals are able to do this)

3. `cd develop` (or `cd master`, as appropriate to the branch to be rebuilt)

4. `git pull`
   Updates the version stored in this folder to the version stored centrally.

5. Re-compile the code. This is essential because the Git repository does not include any of the files generated in the compilation process.

```
cmake3 -H. -Bbuild
cmake3 --build buid
```

6. `exit` (to return to your own username again)

The `cmake` steps performs the rebuild of the `process.exe` executable file, updates the User Guide and all the html files, and recreates the Python dictionaries as required by the Python utilities.

Chapter 4

Graphical User Interface

To currently use the PROCESSGUI, go through the following steps. This only works if you have at least reporter access on the project.

1. Clone `Home` branch from the Gitlab. `git clone --branch home git@git.ccfe.ac.uk:mkumar/PROCESS_GUI.git folder_name` . If `folder_name` is not typed then it will create a folder `PROCESS_GUI` and clone source code there.
2. Go to folder with source code where you will find a number of directory. Issue command `qmake-qt5 PROCESS_GUI.pro` , which will generate Makefile for compilation.
3. After successful step 2, issue command `make`. This will initiate compilation and save executable to a folder called `bin`.
4. After successful step 3, go to folder called `bin` and type `./PROCESS_GUI.exe` to launch the application.

Bibliography

[1] Git version control system `http://git-scm.com/`