Developers guide Documentation

Release 0.1

DFTB+ developers

Feb 06, 2018

Contents

CHAPTER 1

Git workflow

1.1 General workflow

For developing DFTB+ we use a feature-branching workflow described, for example, in Understanding the GitHub Flow. The main points for most developers are:

- Development happens based on the *master* branch, which always contains a clean release-ready code.
- Every feature is developed in a separate feature branch, which is derived from the *master* branch. If the feature is mature enough (it works correctly, its code is clean, it is well documented, thoroughly tested, etc.), the feature branch is merged into the *master* branch.

The main official public (upstream) repository only contains two branches: the branches *master* and *release*, latter containing tagged official releases. Some short living intermediate branches (e.g. *stage* and *hotfix*) may appear from time to time, but these are special purpose branches created by the administrators/release managers and are not being used for feature development.

In order to add a feature, you have to do the following steps:

1. Fork the official (upstream) repository and set up your own repository. (This step you have to do only once.)
2. Make sure, your *master* branch is synchronised to the upstream *master*.
3. Derive a feature branch from the *master* branch of your forked project.
4. Develop your feature in your feature branch.
5. When your feature implementation has finished, make sure, it integrates well into the most recent version of the code. (The code may have evolved while you were implementing your feature.)
6. Issue a *pull request* for your feature branch.
7. Wait for feedback from the core developers and then apply any suggestions or required changes to your feature branch.
8. When you obtain the notification that your feature branch has been merged to the upstream *master* branch, delete the feature branch in your personal repository.
9. In order to develop the next feature, execute the above steps again, *starting from step 2*.

Below you find a detailed description of each step, using the DFTB+ main repository as an example. If you work on an other DFTB+ related project, replace the repository name *dftbplus* with the actual repository name.

1.2 Fork the project

1.2.1 Fork the repository

1. Fork the desired repository (e.g. *dftbplus*) owned by the user *dftbplus* to *your personal* GitHub account. You will find the *Fork* button in the upper right corner on the project page.
2. Check out your personal fork to your local machine:

```
git clone git@github.com:YOUR_USER_NAME/dftbplus.git
```

3. Set up a mirror of the upstream reference repository:

```
git remote add upstream git@github.com:dftbplus/dftbplus.git
```

1.2.2 Set up your own repository

Set up your identity

When you contribute to our project it is important that the author information of your commits contain your full name and a valid (preferably your official) email address. Set up those for your repository (or globally by adding the `--global` option) by

```
git config user.name 'FULL_NAME'
git config user.email 'EMAIL_ADDRESS'
```

Add check on commit message formatting

We use the commonly adopted git commit message format containing a short imperative subject line and an optional detailed description which is separated by an empty line (see for example How to Write a Git Commit Message). Using a simple commit message hook, git can check that your commit messages follow this format. Please copy our special git commit hook as *.git/hooks/commit-msg* into your repository and make it executable (`chmod +x .git/hooks/commit-msg`).

You may wish to make this a global hook for all of your git repositories by adding it to an init.templatedir directory. This can be added for *all* repositories with

```
git config --global init.templatedir '~/.git-templates'
mkdir -p ~/.git-templates/hooks
```

The commit-msg file can then be placed in *~/.git-templates/hooks/commit-msg*. We would then also suggest setting the permission to be user writable only

```
chmod -R 700 ~/.git-templates
```

Any new local repositories will settings specified from this directory, unless overridden by a local *.git/* directory within the repository itself. Existing repositories need to be reinitialised in their top directory to use the init.templatedir

```
git init
```

Again, any local *.git/* directory overrides settings in *~/.git-templates*

1.3 Synchronising to the upstream master branch

Before you start developing a feature, you should make sure that you implement your feature in the most recent version of the code. This minimises the chances of conflicts (and additional work needed from you) when your feature is later merged into the upstream repository:

1. Pull the recent changes from the upstream *master* branch into your local *master* branch:

   ```
   git checkout master
   git pull --ff-only upstream master
   ```

   Upload the changes in your local *master* branch to GitHub by issuing:

   ```
   git push origin master
   ```

   Note: if the `git pull --ff-only upstream master` command fails, you have probably polluted your personal master branch, and it can no longer be made to exactly match the upstream one. In that case, you may revert it via a hard reset:

   ```
   git reset --hard upstream/master
   ```

   You will then have to derive a new feature branch from the reset *master* branch and then add your changes manually to this new feature branch. Therefore, to avoid this extra work, make sure you never change your personal *master*, apart from synchronising it with the upstream *master*.

1.4 Developing your feature

1. Check out your *master* branch, which you should have synchronised to upstream *master* as described in the previous section:

   ```
   git checkout master
   ```

2. Create you own feature branch:

   ```
   git checkout -b some-new-feature
   ```

   To develop a new feature you should always create a new branch derived from *master*. You should never work on the *master* branch directly, or merge anything from your feature branches onto it. Its only purpose is to mirror the status of the upstream *master* branch. The feature branch name should be short and descriptive for the feature you are going to implement.

3. Develop your new feature in your local branch. Make check-ins whenever it seems to be logically useful:

   ```
   git commit -m "Some new thing added...."
   ```

4. Consider adding regression tests for your feature in the test directory and also adding to the documentation for the code.

5. If you want to share your development with others (or make a backup of your repository in the cloud), upload the current status of your local feature branch by pushing it to your personal repository:

```
git push --set-upstream origin some-new-feature
```

   This also automatically connects the appropriate branch of your personal repository on GitHub (*origin/some-new-feature*) with your local branch (*some-new-feature*), so from now on, if you are on your *some-new-feature* branch, a simple:

```
git push
```

   command without any additional options will be enough to transfer your recent changes on this branch to GitHub.

1.5 Merge the changes back into the upstream repository

When you have finished implementing your feature, it should be merged back into the upstream *master* as soon as possible, in order to minimise the number of possible conflicts. Generally, you should try to implement features in the smallest meaningful units, so that they can be quickly merged into the upstream repository.

First, make sure, that your feature integrates well into the most recent main code version. Be aware that the upstream code may have evolved while you were implementing your feature.

1. First synchronise your *master* branch to the upstream *master*, as written in the section *Synchronising to the upstream master branch*.
2. Integrate any changes that appeared on *master* during your feature development. Depending on how complex your feature branch is (especially how many commits it contains), you should follow one of two different strategies:
   - For simple feature branches with only one or two commits: Rebase your feature branch on *master*:
     (a) Check out your feature branch:

```
git checkout some-new-feature
```

     (b) Rebase it on *master*:

```
git rebase master
```

     Note, that the rebase method changes your git commits by reverting your changes and reapplying them on top of the current code. As long as your feature branch was not used (forked) by anybody else, it does not do any harm and helps to keep the history of your feature branch linear and simple. However, you should never rebase any branches, which you have already shared with others.
   - For more complex feature branches with multiple commits: Merge the *master* branch into your feature branch:
     (a) Check out your feature branch:

```
git checkout some-new-feature
```

     (b) Merge the *master* branch into it:

```
git merge master
```

     This will result in an extra merge commit.

3. Test whether your updated feature branch still works as expected (having regression tests for your feature can help here).
4. Push the latest status of your feature branch to your personal repository on GitHub:

   ```
   git push origin some-new-feature
   ```

   If you used the rebase method above and have pushed your branch to GitHub at least once already before the rebase, you may need the option `-f` to change the git-history (previous git-commits) also on GitHub.
5. Issue a pull request on GitHub for your *some-new-feature* branch (look for the upwards arrow in the left menu).
6. Wait for the comments of other developers, apply any fixes you are asked to make, and push the changes to your feature branch on GitHub.
7. Once the discussion on your pull request is finished, one of the developers with write permission to the upstream repository will merge your branch into the upstream *master* branch. Once this has happened, you should see your changes showing up there.

1.6 Delete your feature branch

Once your feature has been merged into the upstream code you should delete your feature branch, both locally and on GitHub as well:

1. In order to delete the feature branch locally, change to the *master* branch (or any branch other than your feature branch) and delete your feature branch:

   ```
   git checkout master
   git branch -d some-new-feature
   ```
2. In order to delete the feature branch on GitHub as well use the command:

   ```
   git push origin --delete some-new-feature
   ```

This closes the development cycle of your feature and opens a new one for the next one you are going to develop. You can then again create a new branch for the new feature and develop your next extension starting with the steps described in section *Synchronising to the upstream master branch*.

1.7 A note about Submodules

The DFTB+ program uses several libraries from elsewhere in the project. Both mpifx and scalapackfx are required to build the *main* branch code with MPI parallelism enabled. These libraries are included within the repository via the git *submodule* mechanism. However, since the code should be available for users without accounts on github.com, these are included as web links instead of ssh references.

You can globally configure git to substitute ssh links for the https references by issuing the command

```
git config --global url.ssh://git@github.com/.insteadOf https://github.com/
```

You can alternatively set up this substitution for only your local *dftbplus* repository. You should run this command in the directory containing your copy and leave out the `--global` option.

When checking out the code, you can pull the submodules with

```
git submodule update --init --recursive
```

But if you need to modify these submodules, you should fork their respective projects. Then in your DFTB+ repository, change the locations that both the *.gitmodules* file and the submodule's entry in the *.git/config* point to so they match your fork. Finally re-initialise and update the submodules.

CHAPTER 2

Fortran style guide

The main principle when contributing code to Fortran projects should be readability. If your code can not be easily read and understood by others it will be hard to maintain and extend. It should also fit well with the existing parts of the code (in style as well as in its programming paradigms) maintaining the principle of least surprise.

Below you will find some explicit coding rules we try to follow. The list can not cover all aspects, so also look at the existing source code and try to follow the conventions being used.

If you use Emacs as editor, consider adding appropriate customisation settings to your config file in order to automatically enforce some of the conventions below.

2.1 Line length and indentation

- Maximal **line length** is **100** characters. For lines longer than that, use continuation lines.
- **Nested blocks** are indented by **2** white spaces:

```
write(*, *) "Nested block follows"
do ii = 1, 100
  write(*, *) "This is the nested block"
  if (ii == 50) then
    write(*, *) "Next nested block"
  end if
end do
```

- **Continuation lines** are indented by **4** white spaces. Make sure to place continuation characters (&) both at the end of the line as well as at the beginning of the continuation line:

```
call someRoutineWithManyParameters(param1, param2, param3, param4,&
    & param5)
```

  Try to break lines at natural places (e.g. at white space characters) and include one white space character after the opening ampersand in the continuation line.

- **Single line preprocessor directives** are indented as normal code:

```
@:ASSERT(someCondition)
call someRoutine(...)
```

- **Preprocessor block directives** (directives with starting and ending constructs) are outdented by **2** characters with respect of the code they enclose. The enclosed code must be aligned as if the preprocessor directives were not present:

```
  call doSomething()
#:if WITH_SCALAPACK
  call someRoutineScalapackVersion(...)
#:else
  call someRoutineSerialVersion(...)
#:endif

do iKS = 1, nKS
#:if WITH_SCALAPACK
  call someRoutineScalapackVersion(iKS, ...)
#:else
  call someRoutineSerialVersion(iKS, ...)
#:endif
end do
```

2.2 Naming

The naming conventions basically follow those in the Google Style Guide for Java naming convention, with minor modifications.

- **Variable** names follow the **lowerCamelCase** convention:

```
logical :: hasComponent
```

- **Constants** (parameters) use the **lowerCamelCase** convention similar to variables

```
integer, parameter :: maxArraySize = 100
```

  with the exception of the constants used to define the kind parameter for intrinsic types, which should be all lowercase (and short):

```
integer, parameter :: dp = kind(1.0d0)
real(dp) :: val
```

- **Subroutine** and **function** names follow also the **lowerCamelCase** notation:

```
subroutine testSomeFunctionality()
myValue = getSomeValue(...)
```

- **Type** (object) names are written **UpperCamelCase**:

```
type :: TRealList
type(TRealList) :: myList
```

  All type names should be prefixed with a capital 'T', in order to clarify the distinction between type names and variable names:

```
type :: TBroydenMixer
:
end type TBroydenMixer
:
type(TBroydenMixer) :: broydenMixer
```

- **Module** names follow **lower_case_with_underscore** convention:

```
use dftb_common_accuracy
```

  Underscores are used for name-spacing only, so the module above would be typically found at the path *dftb/common/accuracy.f90*. The individual component names (`dftb`, `common`, `accuracy`) may not contain any underscores and must be shorter than 15 characters.

- **Preprocessor** variables and macros follow **UPPER_CASE_WITH_UNDERSCORE** convention:

```
#:if WITH_MPI
  withMpi = ${FORTRAN_LOGICAL(WITH_MPI)}$
#:endif
```

2.3 White spaces

Please use white spaces to make the code readable. In general, you **must use** white spaces in following situations:

- Around arithmetic operators:

```
2 + 2
```

- Around assignment and pointer assignment operators:

```
aa = 3 + 2
pWindow => array(1:3)
```

- Around the `::` separator in declarations:

```
integer :: ind
```

- After commas (`,`) in general and especially in declarations, calls and lists:

```
real(wp), allocatable :: array(:)
type, extends(TBaseType) :: TDerivedType
subroutine myRoutine(par1, par2)
call myRoutine(val1, val2)
print *, 'My value:', val
do ii = 1, 3
array(1:3) = [1, 2, 3]
```

- When separating array indices, when the actual index value for an index contains an expression:

```
myArray(ii + 2, jj) = 12
```

You **may omit** white space in following cases:

- When separating array indices and the actual index values are simple and short (typically two letters) variable names, one or two digit integers or the range operator `::`

```
myArray(:,1) = vector
latVecs(1,1) = 1.0_wp
myArray(ii,jj) = myArray(jj,ii)
```

You **must omit** white spaces in following cases:

- Around opening and closing braces of any kind:

```
call mySubroutine(aa, bb)  ! and NOT call mySubroutine( aa, bb )
myVector(:) = [1, 2, 3]    ! instead of myVector(:) = [ 1, 2, 3 ]
tmp = 2 * (aa + bb)        ! instead of 2 * ( aa + bb )
```

- Around the equal (=) sign, when passing named arguments to a function or subroutine:

```
call mySubroutine(aa, optionalArgument=.true.)
```

- Around the power operator:

```
val = base**power   (instead of val = base ** power)
```

Avoid white spaces for **visual aligning** of code, use:

```
integer, intent(in) :: nNeighbors
real(wp), intent(out) :: interaction
```

instead of:

```
integer, intent(in)   :: nNeighbors
real(wp), intent(out) :: energy
```

Although latter may look more readable, it makes rather difficult to track real changes in the code with the revision control system. For example when a new line is added to the block making the realignment of previous (but otherwise unchanged) lines necessary

```
integer, intent(in)             :: nNeighbors
real(wp), intent(out)           :: energy
real(wp), intent(out), optional :: forces(:)
```

the version control system will indicate all of those lines having been modified, although only the alignment (but not the actual instructions) were changed.

2.4 Comments

- **Module**, **Subroutine** and **function** comments should be consistent with doxygen / FORD literate comments for publicly visible interfaces and variables.
- Comments are indented to the same position as the code they document:

```
! Take spin degeneracy into account
energy = 2.0_wp * energy
```

- Generally, write the comment *before* the code snippet it documents:

```
! Loop over all neighbours
do iNeigh = 1, nNeighbours
```

```
  :
end do
```

- Try to avoid mixing code and comments within one line as this is often hard to read:

```
bb = 2 * aa   ! this comment should be before the line.
```

- Never use multi-line suffix comments, as an indenting editor would mess up the indentation of subsequent lines:

```
bb = 2 * aa  ! This comment goes over multiple lines, therefore, it
             ! should stay ALWAYS before the code snippet and NOT HERE.
```

- Specifically comment any workarounds, include the compiler name and the version number for which the workaround had to be made. Always use the following pattern, so that searching for workarounds which can be possibly removed is easy:

```
! Workaround: gfortran 4.8
! Finalisation not working, we have to deallocate explicitly
deallocate(myPointer)
```

- Comments should always start with one bang only. Comments with two bangs are reserved for source code documentation systems:

```
! This block needs a documentation
do ii = 1, 2
  :
end do
```

- If you need a comment for a longer block of code, consider instead packaging that block of code into a properly named function (if the additional function call would be performance critical, write it as an internal procedure):

```
somePreviousStatement
ind = getFirstNonZero(array)
someStatementAfter
```

instead of

```
somePreviousStatement

! Look for the first nonzero element
found = .false.
do ind = 1, size(array)
  if (array(ind) > 0) then
    found = .true.
    exit
  end if
end do
if (.not. found) then
  ind = 0
end if

someStatementAfter
```

2.5 Allocation status

At several places, the allocation status of a variable is used to signal choices about logical flow in the code:

```
!> SCC module internal variables
type(TScc), allocatable :: sccCalc
.
.
.
if (allocated(sccCalc)) then

end if
```

This is to be preferred to the use of additional logical variables if possible.

Part of the reason for this choice is that from Fortran 2008 onwards, optional arguments to subroutines and functions are treated as not-present if not allocated.

CHAPTER 3

Python components of DFTB+

The PEP 8 guidelines are the basis of the python style used in the code.

Pylint files are suplied (*utils/srccheck/pylint/*) for code checking. In many cases, python unit tests are also implemented (see *test/tools/dptools/* for examples).

For example to test the straingen script for Python3 compliance, from the top of the repository type:

```
env PYTHONPATH=$PWD/tools/dptools/src pylint3 --rcfile \
utils/srccheck/pylint/pylintrc-3.ini tools/dptools/bin/straingen
```

while to test for correct performance:

```
make test_dptools
```

CHAPTER 4

Licence

4.1 Human readable summary of the licence

4.1.1 You are free to

Share copy and redistribute the material in any medium or format

Adapt remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

4.1.2 Under the following terms

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

4.1.3 Notices

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.