CENTRE TECNOLÒGIC DE TELECOMUNICACIONS DE CATALUNYA

GNSS-SDR: Contributing to the source code

Carles Fernández-Prades

April 26, 2015

CONTENTS

1 Using Git **4**
1.1 Working with the source code . . . 5
1.1.1 Note for GitHub users . . . 6
1.1.2 Note for Bitbucket users . . . 7
1.2 GNSS-SDR's Git branching model . . . 8
1.3 Workflow for the creation of a GNSS-SDR block . . . 10
1.4 Submitting contributions . . . 11
1.5 How to use someone else's branch . . . 11
1.6 Setting up tracking branches . . . 12
1.7 Good coding practices . . . 13

2 Coding style **15**
2.1 Naming conventions . . . 15
2.1.1 Naming Rules for Variables . . . 15
2.1.2 Naming Rules for Files . . . 16
2.1.3 Naming Rules for Functions . . . 16
2.1.4 Naming Rules for Classes and Structures . . . 16
2.1.5 Use sensible, descriptive names . . . 17
2.1.6 Only use English names . . . 17

2.1.7 Variables with a large scope should have long names, variables with a small scope can have short names . . . 17
2.1.8 Use namespaces for identifiers declared in different modules . . . 17
2.1.9 Use name prefixes for identifiers declared in different modules . . . 17
2.1.10 Do not use magic numbers . . . 17
2.2 Indentation and Spacing . . . 17
2.2.1 Braces should follow the GNU style . . . 17
2.2.2 Function parameters should be lined up with one parameter per line . . . 18
2.2.3 Braces without any contents may be placed on the same line . . . 18
2.2.4 Each statement should be placed on a line on its own . . . 18
2.2.5 Declare each variable in a separate declaration . . . 18
2.2.6 For declaring pointers and reference the "*" and "&" should be surrounded by spaces on both sides . . . 19
2.2.7 All binary arithmetic, bitwise and assignment operators and the ternary conditional operator (?:) should be surrounded by spaces . . . 19
2.2.8 Lines should not exceed 78 characters . . . 19
2.2.9 Do not use tabs . . . 19
2.3 Comments . . . 19
2.3.1 Comments should be written in English . . . 19
2.3.2 Comments should use the C++-style . . . 19
2.3.3 Use JavaDoc style comments . . . 19
2.3.4 You can even include formulae . . . 21
2.3.5 Multiple line comments should be split in one comment per line, each having the /* and */ markers on the same line . . . 21
2.3.6 All comments should be placed above the line the comment describes, indented identically . . . 21
2.4 Files . . . 21
2.4.1 There should only be one externally visible class defined in each header file 21
2.4.2 There should only be one externally visible function defined in each header file . . . 21
2.4.3 File name should be treated as case sensitive . . . 21
2.4.4 C++ source files should have extension ".cc" . . . 21
2.4.5 C++ header files should have extension ".h" . . . 21
2.4.6 Inline functions should be declared in header files and defined in inline definition files . . . 21
2.4.7 Header files must have include guards . . . 22
2.4.8 The name of the macro used in the include guard should have the same name as the file (excluding the extension) followed by the suffix "_H_" . 22
2.4.9 Header files should be self-contained . . . 22
2.4.10 When a header is included, there should not be a need to include any other headers first . . . 22
2.4.11 System header files should be included with <> and project headers with "" 22
2.4.12 Put #include directives at the top of files . . . 22
2.4.13 Do not use absolute directory names in #include directives . . . 22

2.4.14 Do not use relative directory names in `#include` directives 22
2.4.15 Use `const` instead of #define in header files 23
2.4.16 Each file must start with a copyright notice 23
2.5 Declarations 24
2.5.1 Provide names of parameters in function declarations 24
2.5.2 The parameter names should be the same in all declarations and definitions of the function 24
2.5.3 Do not use exception specifications 25
2.5.4 Declare inherited functions virtual 25
2.5.5 Do not use global variables 25
2.5.6 Do not use global variables or singleton objects 25
2.5.7 Do not use global using declarations and using directives in headers . . 25
2.5.8 The parts of a class definition must be `public`, `protected` and `private` 26
2.5.9 Declare class data private 26
2.5.10 Functions that can be implemented using public interface of a class should not be members 26
2.6 Statements 26
2.6.1 Never use gotos 26
2.6.2 Do not use `break` in loops 27
2.6.3 Do not use `continue` in loops 27
2.6.4 Only have one `return` in a function 27
2.6.5 All `switch` statements should have a `default` label 27
2.6.6 Do not use `do-while` loops 27
2.7 Other typographical issues 27
2.7.1 Avoid macros 27
2.7.2 Do not use literal numbers other than 0 and 1 27
2.7.3 Use plenty of assertions 28
2.7.4 Use prefix increment/decrement instead of postfix increment/decrement when the value of the variable is not used 28
2.7.5 Write conditional expressions like: if (6 == errorNum) 28
2.7.6 Use the new cast operators 29
2.8 Other recommendations 29
2.8.1 Use of Boost libraries is encouraged 29
2.8.2 Use common sense and BE CONSISTENT 29

1 Using Git

One of the greatest advantages of open source is the ability to view, modify and share the source code. This means diagnosing and fixing problems, or adding new features or documentation. In order to contribute these fixes/improvements to the project's developers, you need to send them back your changes in an adequate manner. This can be done by creating and sending a patch file, or by doing a "pull request". This document provides details about how to do that.

For the version control system we use Git[1], a free and open source application that automates the process of keeping an annotated history of the project, allowing reversion of code changes, easy branching and merging, sharing and change tracking. The GNSS-SDR's GitHub repository[2] hosts all of the data files needed for version control, being the central "truth" repository resource of GNSS-SDR's source code. A repository is simply a place where the history of the work is stored. The distributed nature of Git allows users to clone this repository, that is, to obtain an exact replica of the original repository at their own local hard drive. Then, users can build and use the software, modify the source code, share their work, and contribute back to GNSS-SDR. Git is a fantastic but complex source code management system - it has a steep learning curve, but it worth it. A good reference is the freely available Git Pro book[3], but there are plenty of good Git tutorials out there. If you have never used it, start from the Git Basics[4].

In order to get a full copy of GNSS-SDR's Git repository with the source code, just open a terminal and type:

```
$ git clone git://github.com/gnss-sdr/gnss-sdr.git
```

This will create a local folder named `gnss-sdr` that contains GNSS-SDR's source code. In fact, it does a lot more: it creates remote-tracking branches for each branch in the cloned repository, and creates and checks out an initial branch that is forked from the cloned repository's currently active branch.

Branching in Git is one of its many great features. A branch represents an independent line of development or, more accurately, a directed acyclic graph of development. If you have used other version control systems, it is probably helpful to forget most of what you think about branches - in fact, it may be more helpful to think of them practically as contexts since that is how you will most often be using them. When you checkout different branches, you change contexts that you are working in, and you can quickly context-switch back and forth between several different branches. Every time you switch to a new branch, you will actually see different files in your local folder.

In GNSS-SDR, the source code comes in two flavors: "`master`" and "`next`" (these are in fact the names of the Git branches). While "`master`" is the main, most stable branch that contains the latest release, "`next`" is where all the development is happening, the most bleeding edge code out there. Once you have cloned the main repository, you can easily switch between these two branches (or any other created by you or fetched from other users) by going to your

[1] See `http://git-scm.com/`
[2] See `https://github.com/gnss-sdr/gnss-sdr`
[3] See `http://git-scm.com/book/en/v2`
[4] See `http://git-scm.com/book/en/v2/Getting-Started-Git-Basics`

git-cloned repository and issuing the `git checkout` command with the name of the desired branch name, like this:

```
$ git checkout master   # now you are in the master branch
$ git checkout next     # now you are in the next branch
```

Listing 1: Switching branches.

If you do not know in which branch you are, pay attention to the first line of this command's output:

```
$ git status
```

Listing 2: Getting git status.

1.1 Working with the source code

If you are on GitHub, BitBucket or similar services, see below. Anyway, you can get the latest source by doing:

```
$ git clone git://github.com/gnss-sdr/gnss-sdr.git
```

Listing 3: Cloning the GNSS-SDR repository

This gets the latest upstream source and clones it into a `gnss-sdr` directory. If you have not done so yet, Git needs some configuring, such as setting up an identity:

```
$ git config user.name "Your␣Name"
$ git config user.email your@email.abc
```

Listing 4: Configuring Git

If you already cloned the repository before, download the latest changes and keep your repository in sync with the order of committed changes in the origin (this is required before submitting your changes back to origin):

```
$ git pull --rebase
```

Listing 5: Cloning the GNSS-SDR repository

Now that you are up to date, go to the `next` branch and create a new branch off from it:

```
$ git checkout next
$ git checkout -b my_feature
```

Listing 6: Branching off from next.

Whenever you want to work on something, create a branch for it. Then, do your changes, stage modified and new files and do commits:

```
... (change files, compile, test) ...
$ git add file1.cc file1.h ...    # This is called file staging
$ git commit -m "adding␣stuff"    # Records staged files to the repository
```

Listing 7: Adding files to the staging area and committing them.

Once your changes are finished, you can return back to next and merge your work:

```
$ git checkout next
$ git merge my_feature
```

Listing 8: Merging back to the `next` branch.

This process is depicted in Figure 1.1.

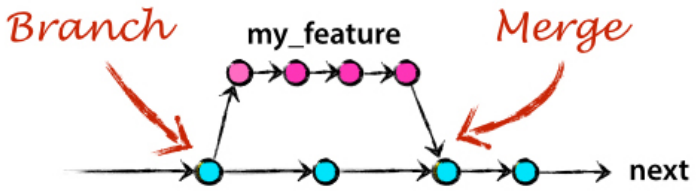

Figure 1.1: Always branch off from `next` when developing a new feature. Each node at the figure represents a commit.

As you edit files, Git sees them as modified, because you have changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats. The command git add puts the files into a "staging area", an index where you get to determine what changes get shipped away in the next commit, that is, what files are going to be recorded to the repository. If you need more details about this process, check this Git tutorial on how to record changes in your repository. Figure 1.2 summarizes this workflow.

1.1.1 Note for GitHub users

If you are a GitHub[5] user and want to take advantage of their collaborative code sharing and review tools, this is the easiest way:

1. Firstly (and just one time), you need to set up an SSH key for your local machine; see the documentation on how to generate a SSH key and tell GitHub about it at `https://help.github.com/articles/generating-ssh-keys/`
2. Go to `https://github.com/gnss-sdr/gnss-sdr`, fork your repository off ("Fork" button at the upper right corner), then create a remote branch (branching off from `next`) on GitHub and push there your changes.
3. Go onto the GitHub site, visit your repository.
4. There is a "Pull Request" button that will do all the work for you.

[5] see `https://github.com`

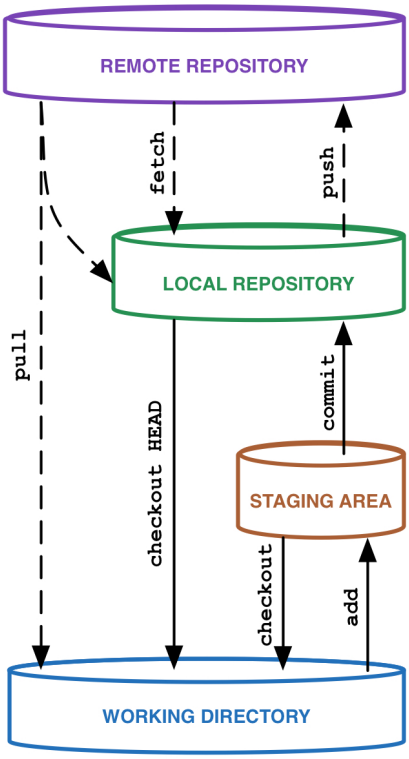

Figure 1.2: Git workflow diagram.

1.1.2 Note for Bitbucket users

If you prefer BitBucket[6] and their offering of unlimited private repositories, log in there and click on the "create" button at the top. Fill in the form accordingly and make it a private repository. Then, in your local machine:

```
$ git remote add bb https://username@bitbucket.org/username/repo_name.git
```

Listing 9: Adding a Bitbucket remote.

and push your local branches there:

```
$ git push bb master
$ git push bb next
$ git push bb my_feature
```

Listing 10: Pushing branches to the Bitbucket remote.

[6] see `https://bitbucket.org`

1.2 GNSS-SDR'S GIT BRANCHING MODEL

The central "truth" repository lives at Github. It holds two branches with infinite lifetime, called "`master`" and "`next`". Note that this repository is only *considered* to be the central one (since Git is a decentralized version control system, there is no such thing as a central repository at a technical level). We will refer to this repository as `origin`, since this name is familiar to all Git users. In addition to those branches, our development model uses a variety of supporting branches to aid parallel development between team members, ease tracking of features, and to prepare for production releases. The model is summarized in Figure 1.3.

Thus, `origin` contains two branches:

- We consider `origin/master` to be the main branch where the source code of `HEAD` always reflects a production-ready state.
- We consider `origin/next` to be the main branch where the source code of `HEAD` always reflects a state with the latest delivered development changes for the next release. This is where any automatic nightly builds are built from.

When the source code in the `next` branch reaches a stable point and is ready to be released, all of the changes are merged back into `master` and then tagged with a release number.

Therefore, each time when changes are merged back into `master`, this is a new production release by definition. The only exception is the new commit after tagging that includes the Digital Object Identifier (DOI) for that particular release.

This model is based in the post "A successful Git branching model", by Vincent Driessen[7].

[7] See `http://nvie.com/posts/a-successful-git-branching-model/`

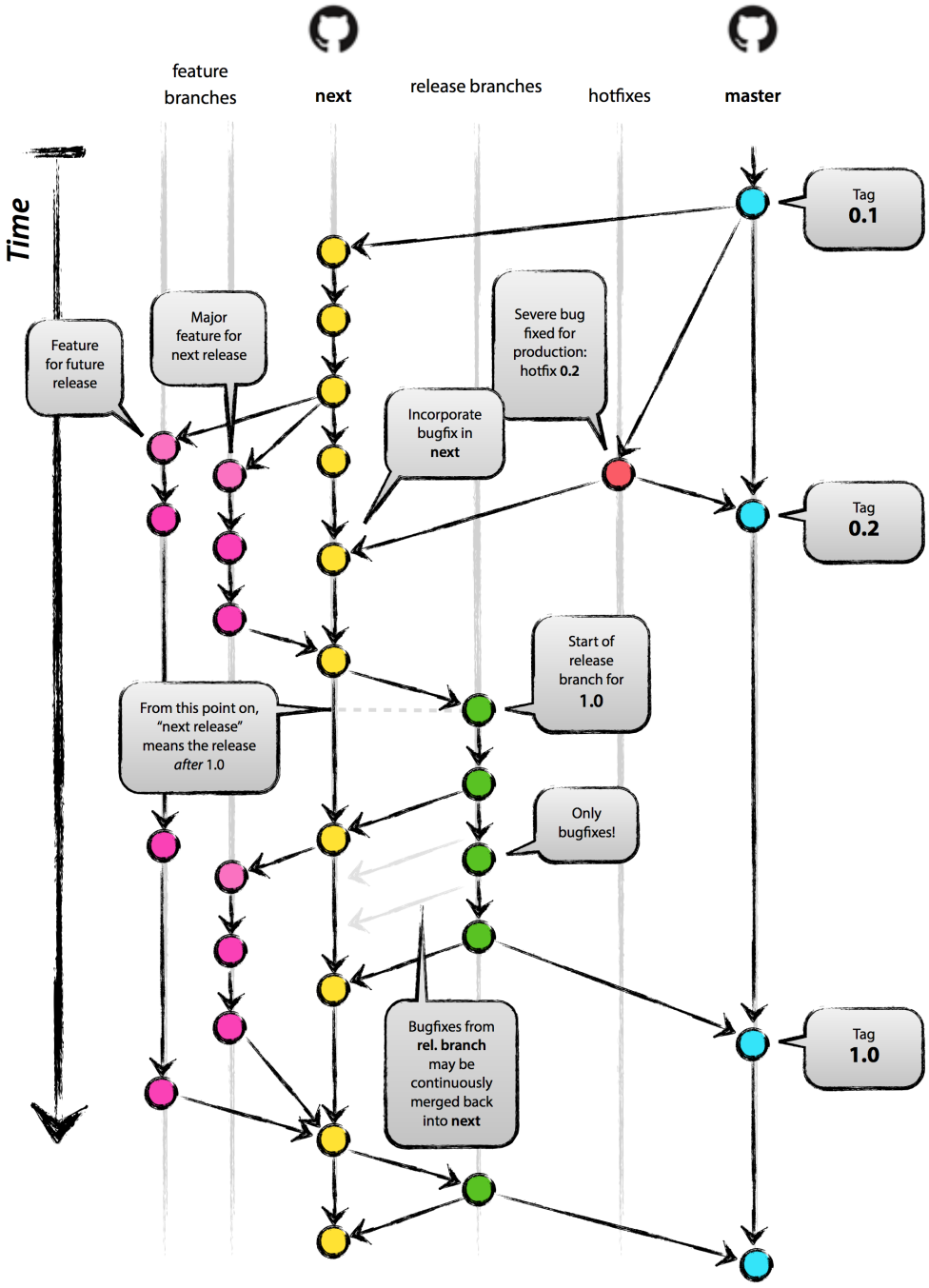

Figure 1.3: GNSS-SDR's Git branching model.

1.3 Workflow for the creation of a GNSS-SDR block

The recommended workflow for creating a new signal processing block is the following:

1. **Write the test for your signal processing block.**
   Write the test `src/test/gnss_block/my_block_test.cc` You can find examples of from existing blocks' tests at `src/test/gnss_block/`.

   Add an `#include` to `src/tests/test_main.cc` pointing to your test:

   ```
   #include "gnss_block/my_block_test.cc"
   ```

   Listing 11: Adding your test to the list.

   Google C++ Testing Framework automatically keeps track of all tests defined, and does not require the user to enumerate them in order to run them.

   You can now compile the software. At the `install/` folder you will find an executable called `install/run_tests`; that will perform all the existing tests and report their results. Since your block is not written yet, the test `my_block_test` should successfully fail since you are testing a block that does not exist.

2. **Write the test for the block factory of your signal processing block.**
   Modify the block factory test file `src/test/gnss_block/gnss_block_factory_test.cc` and include your test. You can now compile again the software. Since your block is not yet written, the test should successfully fail.

3. **Write your actual block, consisting of an adapter and a GNU Radio block.**
   A processing block consists of two parts: the adapter, a class that accommodates your custom processing block to the common interface expected for a block of its kind (`AcquisitionInterface`, `TrackingInterface` and so on); and the GNU Radio block, that is the actual signal processing implementation.

   If you are writing a signal processing block, please put it in an adequate place in the directory tree, under `src/algorithms`. For instance, if you are working on an acquisition algorithm, you need to create the following files:

   - `src/algorithms/acquisition/adapters/my_adapter.h`
   - `src/algorithms/acquisition/adapters/my_adapter.cc`
   - `src/algorithms/acquisition/gnuradio_blocks/my_signal_processing_block.h`
   - `src/algorithms/acquisition/gnuradio_blocks/my_signal_processing_block.cc`

   Then you have to incorporate your recently created block into the list of blocks that the `gnss_factory_block` is able to produce. To that end, you have to modify accordingly the block factory at `src/core/receiver/gnss_block_factory.cc` in order to have an easy way of creating blocks with the required configuration and implementation. Finally, you have to inform the building tool that there are new files to be compiled. This is done in the corresponding `CMakeLists.txt`. Take a look at the existing ones and add your files where appropriate.

4. **Pass the tests.**
   Compile the code (by typing `cmake../ && make` in the build directory of GNSS-SDR and then execute `install/run_tests`.

5. **Test the performance of a whole GNSS receiver that is using your signal processing block.**
   You might want to include your block in the configuration of a GNSS receiver that makes use of it, maybe in order to compare its performance to other implementations. Modify an existing configuration at `conf/` in order to include your block and the corresponding parameters, creating a new file "`conf/my_GNSS_receiver.conf`". Then you can invoke your brand new GNSS receiver by typing:

```
$ ./gnss-sdr --config-file=../conf/my_GNSS_receiver.conf
```

Listing 12: Executing your receiver.

1.4 Submitting contributions

Once you are ready to submit your code:

```
$ git format-patch next Added-a-Cool-Feature.patch
```

Listing 13: Creating a patch.

This makes a patch for the latest commit and displays the file name. You can send this patch via email (best way is by the gnss-sdr-developers mailing list). Someone else can apply this patch in his repository using:

```
$ git am --signoff < FILENAME.patch
```

Listing 14: Applying a patch.

This will apply and commit the patch with original author's information, appending to the commit message that it was signed off by you.

If you are working on a remote Git server, push your changes there (here pushing means uploading your data to the repository) and use the `git request-pull` command, that creates a template email that you can send to someone else to request a pull from them:

```
$ git request-pull origin/next git://server.org/path/to/repo.git my_feature
```

Listing 15: Creating a pull request.

1.5 How to use someone else's branch

As a collaboration tool, Git can set up what is known as a remote to connect to other people's repositories. Those repositories, in the Git distributed system, do not need to be on a single server, but can be anywhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.

Now, someone might be doing something interesting you care about. Say this is Carles FernÃąndez and you want to track his work:

```
$ git remote add cf git://github.com/carlesfernandez/gnss-sdr.git
$ git fetch cf    # This downloads all the content from Carles' repo.
$ git branch -r   # Lists remote branches
```

Listing 16: Adding someone else's branch.

Then you can see all your available remote branches (including those of the remote repository you just added) and their name. You can then checkout a specific remote branch by typing:

```
$ git checkout cf/very_cool_feature
```

Listing 17: Checking a remote branch.

When checking out a remote branch, you can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout. If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the `checkout` command again:

```
$  git checkout -b new_branch_name
```

Listing 18: Branching off

1.6 Setting up tracking branches

When you create a new local branch off from next, it diverges from your local next at that point in time:

```
$ git checkout next
$ git checkout --track -b my_feature
```

Listing 19: Creating a new local branch

If you want to keep that new branch updated with the new changes of next, you either need to pull changes through next and then rebase your branch, or remember where your merge point was.

What we have here is `my_feature` tracking `next`, and not `origin/next`; in other words, it is a local branch tracking another local branch. There are times when this is useful, but what if you want to track the remote one directly instead of having to pull through a local copy? The solution is to have a tracking branch.

In Git terminology, a tracking branch is a local branch that is connected to a remote branch. When you `push` and `pull` on that branch, it automatically pushes and pulls to the remote branch that it is connected with.

Checking out a local branch from a remote branch automatically creates a tracking branch. If you are on a tracking branch and type `git push`, Git automatically knows which server and branch to push to. Also, running `git pull` while on one of these branches fetches all the remote references and then automatically merges in the corresponding remote branch.

When you clone a repository, it automatically creates a master branch that tracks origin/-master. That is why `git push` and `git pull` work out of the box with no other arguments: if you are on a tracking branch and type git push, Git automatically knows which server and branch to push to. However, you can set up other tracking branches if you wish - ones that do not track branches on `origin` and do not track the `master` branch.

Say you want to work off the `next` branch. First, you need a copy of that in your local repository - a tracking branch:

```
$ git fetch --all     # This downloads all available content
$ git branch -r       # Lists remote branches
```

Listing 20: Downloading new code

Then, create a local tracking branch called `my_feature` from the remote branch called `origin/next`:

```
$ git checkout --track -b my_feature origin/next
```

Listing 21: Creating a branch

Now you have a branch `my_feature` which is tracking `origin/next`. When there is an update in the upstream repository, and do a `pull`, you will see it updating both `next`, and also `my_feature`.

Important: Never, ever commit (write) to a local tracking branch. Always use them as a base to branch off!

1.7 Good coding practices

- **Keep you changes referred to the latest commit of `origin/next` branch.** In Git terminology, this is called rebasing a branch. When rebasing, you need to specify which branch you want to rebase onto:

  ```
  $ git checkout my_feature
  $ git fetch origin
  $ git rebase origin/next
  ```

  This simply reshuffles your `my_feature` branch patches on top of the current `next` branch in the `origin` remote. In this context, `origin` is the default name of the remote Git repository you cloned from. Rebasing is a good idea while your feature branch is still in development. Check out Scott Chacon's Git Pro book section about rebasing to find out more details about this concept.

- **Use an integrated development environment (IDE) with Git support.** Most modern C++ IDEs have nice interfaces for using Git. We use Eclipse, and we love using EGit, an Eclipse Team provider for Git. If you know about another combination that works well for you, feel free to share it.

- **Before creating the patch file, please be sure that after your modifications everything compiles and runs without problems, and clean up your work.** Remove any junk lines or comments you may have left while fixing the code, and make sure your follow the

recommended coding style for GNSS-SDR described in Section 2 (indentation, white spaces, naming conventions and so on). This will make other developers' life easier.

- **Tell us about your branch!** If you have significant changes, you can simply email us (again, best way is by mailing list, so other users can get to know about you work) and tell us about your code. All we need is the link to your remote branch.

2 CODING STYLE

Since the seminal work by Kernighan et al. in 1974[8], there is a clear concern on the style in writing software and its impact in the final quality of the product. Following programming guidelines and code conventions not only helps to avoid introducing errors, but cuts maintenance costs and favors effective code reuse.

The following rules capture the most important aspects of coding style:

- All should be as understandable as possible.
- All should be as readable as possible, except when it would conflict with the previous rule.
- All should be as simple as possible, except when it would conflict with the previous rules.

The best way to look at these rules is to make everything as simple as possible, unless understandability or readability suffer. As a programmer you must always try to respect the above rules, even if you do not follow our suggested style of coding.

Any violation to the guide is allowed if it enhances readability. The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible.

The rules can be violated if there are strong personal objections against them. The attempt is to make a guideline, not to force a particular coding style onto individuals. Experienced programmers normally want to adopt a style like this anyway, but having one, and at least requiring everyone to get familiar with it, usually makes people start thinking about programming styling and evaluate their own habits in this area. On the other hand, new and inexperienced programmers normally use a style guide as a convenience of getting into the programming jargon more easily.

This coding style guide was written based on this Coding Style Generator. Some ideas were borrowed from the Google C++ Style Guide.

2.1 NAMING CONVENTIONS

2.1.1 NAMING RULES FOR VARIABLES

Variables are named using lower-case letters and words are separated using under-score. Abbreviations, when used in variable names, are also written in lower-case letters. Examples:

```
fft_size
my_variable_name
```

Listing 22: Variable names.

[8] B. W. Kernighan, P. J. Plauger, *The Elements of Programming Style*, New York: McGraw–Hill, 1974.

2.1.2 Naming Rules for Files

Files are named using lower-case letters and words are separated using under-score. Abbreviations, when used in file names, are also written in lower-case letters. Source files are named using .cc suffix, whereas header files end with .h extension. Examples:

```
my_file.h
my_file.cc
```

Listing 23: Source file names.

2.1.3 Naming Rules for Functions

Function names are named using lower-case letters and words are separated using under-score. Abbreviations, when used in function names, are also written in lower-case letters. This rule applies both to stand-alone functions as well as to member functions of classes. Example:

```
int my_function_name(int a, int b)
```

Listing 24: Function call.

Functions do things, and their name should make this clear. Therefore, always include a verb in it, no exceptions! Here follows a simple function call. Use a space right after the opening brace and right before the closing brace, just as with if structures. Also leave a space right after the comma, as it is done in the English language.

```
do_something( with, these, parameters );
```

Listing 25: Long function call.

When function calls get too long, you'll have to split them up in several lines. Align the following lines with the previous ones so the structure becomes obvious, and go to the next line after the comma.

```
GNSSChannel(ConfigurationInterface *configuration,
            unsigned int channel, unsigned int satellite,
            GNSSBlockInterface *pass_through,
            AcquisitionInterface *acq,
            CorrelatorInterface *trk,
            GNSSBlockInterface *nav, std::string role,
            std::string implementation);
```

Listing 26: Variable names.

2.1.4 Naming Rules for Classes and Structures

Each new word in a class or structure name should always start with a capital letter and the words should be separated with an under-score. Abbreviations are written in capital letters. Examples:

```
My_Class_Name
My_Struct_Name
BPSK
```

Listing 27: Class names.

2.1.5 Use sensible, descriptive names

Do not use short cryptic names or names based on internal jokes. It should be easy to type a name without looking up how it is spelt. Exception: Loop variables and variables with a small scope (less than 20 lines) may have short names to save space if the purpose of that variable is obvious.

2.1.6 Only use English names

It is confusing when mixing languages for names. English is the preferred language because of its spread in research and software development and because most libraries already used are in English.

2.1.7 Variables with a large scope should have long names, variables with a small scope can have short names

Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are i, j, k, m, n and for characters c and d.

2.1.8 Use namespaces for identifiers declared in different modules

This avoids name clashes.

2.1.9 Use name prefixes for identifiers declared in different modules

This avoids name clashes.

2.1.10 Do not use magic numbers

Unnamed or ill–documented numerical constant values make the code difficult to follow.

2.2 Indentation and Spacing

2.2.1 Braces should follow the GNU style

The GNU Bracing Style means that the curly brace pairs are indented from the surrounding statement. Statements and declarations between the braces are indented relative to the braces. Braces should be indented 4 columns to the right of the starting position of the enclosing statement or declaration. Example:

```
void f(int a)
{
    int i;
    if (a > 0)
        {
            i = a;
        }
    else
        {
            i = a;
        }
}

class A
{
};
```

Listing 28: Indentation example.

2.2.2 Function parameters should be lined up with one parameter per line

This allows enough space for Short comments after each parameter. Loop and conditional statements should always have brace enclosed sub-statements. The code looks more consistent if all conditional and loop statements have braces.

Even if there is only a single statement after the condition or loop statement today, there might be a need for more code in the future.

2.2.3 Braces without any contents may be placed on the same line

The only time when two braces can appear on the same line is when they do not contain any code. Example:

```
while (...) {}
```

Listing 29: Braces appearing at the same line.

2.2.4 Each statement should be placed on a line on its own

There is no need to make code compact. Putting several statements on the same line only makes the code cryptic to read.

2.2.5 Declare each variable in a separate declaration

This makes it easier to see all variables. It also avoids the problem of knowing which variables are pointers. (Bad) example:

```
int* p, i;
```

Listing 30: Bad example.

It is easy to forget that the star belongs to the declared name, not the type, and look at it and assume that the type is "pointer to int" and both p and i are declared to this type.

2.2.6 For declaring pointers and reference the "*" and "&" should be surrounded by spaces on both sides

2.2.7 All binary arithmetic, bitwise and assignment operators and the ternary conditional operator (?:) should be surrounded by spaces

The comma operator should be followed by a space but not preceded by one; all other operators should not be used with spaces.

2.2.8 Lines should not exceed 78 characters

Even if your editor handles long lines, other people may have set up their editors differently. Long lines in the code may also cause problems for other programs and printers.

2.2.9 Do not use tabs

Tabs make the source code difficult to read because different programs treat the tabs differently. The same code can look very differently in different views. Avoid using tabs in your source code to avoid this problem. Use spaces instead.

2.3 Comments

2.3.1 Comments should be written in English

2.3.2 Comments should use the C++-style

Be consistent and use the `// ...` style comments.

2.3.3 Use JavaDoc style comments

The comment styles `///` and `/** ... */` are used by JavaDoc, Doxygen and some other code documenting tools.

For a complete description on how to document the code with Doxygen, click here

All classes in GNSS-SDR should be properly documented with Doxygen comments in include (.h) files. Source (.cc) files should be documented according to a normal standard for well documented C++ code.

An example of how the interface of a class should be documented in GNSS-SDR is shown here:

```
/*!
 * \brief Brief description of My_Class here
 *
 * Detailed description of My_Class here. With example code if needed.
 */
class My_Class
```

```
{
public:
    //! Default constructor
    My_Class(void)
    {
        setup_done = false;
    }

    /*!
     * \brief Constructor that initializes the class with parameters
     *
     * Detailed description of the constructor here if needed
     *
     * \param[in] param1 Description of \a param1 here
     * \param[in] param2 Description of \a param2 here
     */
    My_Class(TYPE1 param1, TYPE2 param2)
    {
        setup(param1, param2);
    }

    /*!
     * \brief Setup function for My_Class
     *
     * Detailed description of the setup function here if needed
     *
     * \param[in] param1 Description of \a param1 here
     * \param[in] param2 Description of \a param2 here
     */
    void setup(TYPE1 param1, TYPE2 param2);

    /*!
     * \brief Brief description of member_function1
     *
     * Detailed description of member_function1 here if needed
     *
     * \param[in]     param1 Description of \a param1 here
     * \param[in]     param2 Description of \a param2 here
     * \param[in,out] param3 Description of \a param3 here
     * \return Description of the return value here
     */
    TYPE4 member_function1(TYPE1 param1, TYPE2 param2, TYPE3 &param3);

private:

    bool setup_done;  /*!< Variable that checks if the class is properly
                           initialized with parameters */
    TYPE1 private_variable1; //!< Short description of private_variable1 here
    TYPE2 private_variable2; //!< Short description of private_variable2 here
};
```

Listing 31: Indentation example.

2.3.4 You can even include formulae

2.3.5 Multiple line comments should be split in one comment per line, each having the /* and */ markers on the same line

Long comments which span several lines are difficult to follow. Having each line in the comment begin with /* makes the comment much clearer and easier to identify. This also avoids problems where comments contain code and possibly nested comments.

2.3.6 All comments should be placed above the line the comment describes, indented identically

Being consistent on placement of comments removes any question on what the comment refers to.

Use `#ifdef` instead of `/* ... */` to comment out blocks of code. The code that is commented out may already contain comments which then terminate the comment block and causes lots of compile errors or other harder to find errors.

2.4 Files

2.4.1 There should only be one externally visible class defined in each header file

Having as few declarations as possible in a header file reduces header dependencies.

The header file should have the same name as the class plus extension .h.

External non-member functions that belong to the class interface may also be declared in the same header file.

2.4.2 There should only be one externally visible function defined in each header file

Having as few declarations as possible in a header file reduces header dependencies.

The header file should have the same name as the function plus extension .h.

Overloaded functions count as a single function here.

2.4.3 File name should be treated as case sensitive

2.4.4 C++ source files should have extension ".cc"

2.4.5 C++ header files should have extension ".h"

2.4.6 Inline functions should be declared in header files and defined in inline definition files

The keyword `inline` should be used in both places.

Using a separate inline file is useful to keep the header files clean and small. The separation is also useful where the inlining is disabled in debug builds. The inline file is then included from the source file instead of the header file to reduce compile time.

2.4.7 Header files must have include guards

The include guard protects against the header file being included multiple times. The format of the symbol name should be `<PROJECT>_<PATH>_<FILE>_H_`. To guarantee uniqueness, they should be based on the full path in a project's source tree. For example, the file `gnss-sdr/src/bar/baz.h` should have the following guard:

```
#ifndef GNSS_SDR_BAR_BAZ_H_
#define GNSS_SDR_BAR_BAZ_H_
...
#endif // GNSS_SDR_BAR_BAZ_H_
```

Listing 32: Header guard.

2.4.8 The name of the macro used in the include guard should have the same name as the file (excluding the extension) followed by the suffix "_H_"

This avoids clashing with other names.

2.4.9 Header files should be self-contained

No missing `#includes`.

2.4.10 When a header is included, there should not be a need to include any other headers first

A simple way to make sure that a header file does not have any dependencies is to include it first in the corresponding source file. Example:

```
/* foobar.cc */
#include "foobar.h"
#include <cmath>
...
```

Listing 33: Include the header file first in the source file.

2.4.11 System header files should be included with <> and project headers with ""

2.4.12 Put #include directives at the top of files

Having all `#include` directives in one place makes it easy to find them.

2.4.13 Do not use absolute directory names in `#include` directives

The directory structure may be different on other systems.

2.4.14 Do not use relative directory names in `#include` directives

The directory structure of the project may change in the future. It is then difficult to correct all the directory names.

2.4.15 USE CONST INSTEAD OF #DEFINE IN HEADER FILES

`#define` is a preprocessor directive. Before compiling, the middle symbol is replaced by the right hand symbol(s). The preprocessor does nothing but text replacement, so `#defines` have no respect for the usual C++ scoping rules. In other words, `#define` is not type safe.

When replacing `#defines` with constants, two special cases are worth mentioning. The first is defining constant pointers. Because constant definitions are typically put in header files (where many different source files will include them), it is important that the pointer be declared `const`, usually in addition to what the pointer points to. To define a constant char*-based string in a header file, for example, you have to write `const` twice:

```
const char * const authorName = "Carlos␣Aviles";
```

Listing 34: Defining constant pointers.

However, it is worth reminding you here that string objects are generally preferable to their `char*`-based progenitors, so `authorName` is often better defined this way:

```
const std::string authorName("Carlos␣Aviles");
```

Listing 35: Better definition.

The second special case concerns class-specific constants. To limit the scope of a constant to a class, you must make it a member, and to ensure there is at most one copy of the constant, you must make it a static member:

```
class My_Acquisition_Algorithm
{
private:
    static const int num_dwells = 5; // constant declaration
    int scores[num_dwells];          // use of constant
    ...
};
```

Listing 36: Class-specific constants.

In general, use `const` whenever possible. The wonderful thing about `const` is that it allows you to specify a semantic constraint âĂŤ a particular object should not be modified, and compilers will enforce that constraint. It allows you to communicate to both compilers and other programmers that a value should remain invariant. Whenever that is true, you should be sure to say so, because that way you enlist your compilers' aid in making sure the constraint is not violated.

2.4.16 EACH FILE MUST START WITH A COPYRIGHT NOTICE

Please use the following template at the header of all files:

```
/*!
 * \file filename
 * \brief Brief description of the file here
 * \author Names of the authors who contributed to this code
 *
```

```
 * Detailed description of the file here if needed.
 *
 * -------------------------------------------------------------------------
 *
 * Copyright (C) 2010-2015  (see AUTHORS file for a list of contributors)
 *
 * GNSS-SDR is a software defined Global Navigation
 *          Satellite Systems receiver
 *
 * This file is part of GNSS-SDR.
 *
 * GNSS-SDR is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * GNSS-SDR is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNSS-SDR. If not, see <http://www.gnu.org/licenses/>.
 *
 * -------------------------------------------------------------------------
 */
```

Listing 37: copyright notice.

2.5 Declarations

2.5.1 Provide names of parameters in function declarations

Parameter names are useful to document what the parameter is used for.

2.5.2 The parameter names should be the same in all declarations and definitions of the function

Use a `typedef` to define a pointer to a function. Pointers to functions have a strange syntax. The code becomes much clearer if you use a `typedef` for the pointer to function type. This `typedef` name can then be used to declare variables etc.

```
double sin(double arg);
typedef double (*Trigfunc)(double arg);

/* Usage examples */
Trigfunc myfunc = sin;
void callfunc(Trigfunc callback);
Trigfunc functable[10];
```

Listing 38: Parameter names.

2.5.3 Do not use exception specifications

Exception specifications in C++ are not as useful as they look. The compiler does not make the code more efficient. On the contrary, the compiler has to insert code to check that called functions do not violate the specified exception specification at runtime.

2.5.4 Declare inherited functions virtual

An inherited function is implicitly virtual if it is declared virtual in the base class. Repeat the virtual keyword when declaring an inherited function in a derived class to make it clear that this function is virtual.

2.5.5 Do not use global variables

Use singleton objects instead.

Global variables are initialized when the program starts whether it will be used or not. A singleton object is only initialized when the object is used the first time.

If global variables are using other global variables for their initialization there may be a problem if the dependent variables are not initialized yet. The initialization order of global variables in different object files is not defined. Singleton objects do not have this problem as the dependent object will be initialized when it is used. However, watch out for cyclic dependencies in singleton object initializations.

2.5.6 Do not use global variables or singleton objects

Global variables and singleton objects break visibility of what functions do as these can only be used as invisible side effects of functions. To make it clear what inputs and outputs a function has, pass these objects as parameters to the functions.

2.5.7 Do not use global using declarations and using directives in headers

Bringing in names from a `namespace` to the global `namespace` may cause conflicts with other headers. The author of a header does not know in which context the header is used and should avoid polluting the global namespace. Instead, only use using declarations in the source files.

Do not use `using` directives. Bringing in names from a `namespace` can cause all sorts of problems as the `namespace` might contain more names than you would expect. Use them carefully.

```
#include <iostream>

// Bring in names from the std namespace.
using namespace std;

// Declaring an object with the same name as
// a function in the std namespace.
int dec(int);

void f()
```

```
{
    // Now we can use iostream names lazily.
    cout << "Hello world." << endl;

    // Error: Ambiguous reference to dec.
    cout << "Decimal base is " << dec << endl;
}
```

Listing 39: Problem using `namespace`.

2.5.8 The parts of a class definition must be `public`, `protected` and `private`

This makes it easy to read the class definition as the public interface is of interest to most readers.

2.5.9 Declare class data private

Classes should encapsulate their data and only provide access to this data by member functions to ensure that data in class objects are consistent.

The exception to the rule is C type `struct` that only contains data members.

2.5.10 Functions that can be implemented using public interface of a class should not be members

A class definition can be kept small and less prone to change if it only defines the core functionality. Any other functions that can be implemented with this minimal class definition should be implemented as non-member functions. They are still seen as part of the interface of the class.

Example:

```
class T
{
    T operator+=(const T & right);
};

T operator+(const T & left, const T & right)
{
    T temp(left);
    temp += right;
    return temp;
}
```

Listing 40: Functions that can be implemented using public interface of a class should not be members.

2.6 Statements

2.6.1 Never use gotos

`gotos` break structured coding.

2.6.2 DO NOT USE `BREAK` IN LOOPS

A `break` statement is a `goto` statement in disguise and makes code less readable. A `break` statement is still acceptable in switch statements.

2.6.3 DO NOT USE `CONTINUE` IN LOOPS

A `continue` statement is a goto statement in disguise and makes code less readable.

2.6.4 ONLY HAVE ONE `RETURN` IN A FUNCTION

It is confusing when there are more than one `return` statement in a function. Having only one exit point of a function makes it easy to have a single place for post conditions and invariant check. When debugging it is useful to have a single exit point of a function where you can put a single breakpoint or trace output. It is sometimes necessary to introduce a result variable to carry the function return value to the end of the function. This is an acceptable compromise for structured code.

2.6.5 ALL `SWITCH` STATEMENTS SHOULD HAVE A `DEFAULT` LABEL

Even if there is no action for the `default` label, it should be included to show that the programmer has considered values not covered by `case` labels. If the case labels cover all possibilities, it is useful to put an assertion there to document the fact that it is impossible to get here. An assertion also protects from a future situation where a new possibility is introduced by mistake.

2.6.6 DO NOT USE `DO-WHILE` LOOPS

`do-while` loops are less readable than ordinary while loops and for loops since the conditional is at the bottom of the loop. The reader must scan the entire loop in order to understand the scope of the loop. In addition, `do-while` loops are not needed. Any `do-while` loop can easily be rewritten into a `while` loop or a `for` loop. Reducing the number of constructs used enhance readability.

2.7 OTHER TYPOGRAPHICAL ISSUES

2.7.1 AVOID MACROS

Most macros can be replaced by constants, enumerations or inline functions. As macros are not part of the C++ language, they do not provide type safety and debugger support.

2.7.2 DO NOT USE LITERAL NUMBERS OTHER THAN 0 AND 1

Use constants instead of literal numbers to make the code consistent and easy to maintain. The name of the constant is also used to document the purpose of the number.

2.7.3 Use plenty of assertions

Assertions are useful to verify pre-conditions, post-conditions and any other conditions that should never happen. Pre-conditions are useful to verify that functions are called with valid arguments. They are also useful as documentation of what argument value ranges a function is designed to work with.

Assertions are macros that print error messages when the condition is not met. The macros are disabled in release mode and do not cost anything in performance or used memory in the end product.

Example: This square root function is only designed to work with positive numbers.

```
#include <assert.h>

double sqrt(double x)
{
    // precondition: x is positive
    assert(x > 0);
    double result;
    ...
    // postcondition: result^2 ~= x
    assert(abs(result*result-x)/x < 1E-8) ;
}
```

Listing 41: Assertion example.

2.7.4 Use prefix increment/decrement instead of postfix increment/decrement when the value of the variable is not used

For class objects there may be two different member functions for the postfix and prefix operations. The postfix operation has to keep a temporary return value of the object before changing the object. For built-in objects this does not matter as the compiler will be able to optimise away the temporary value when it is not used.

Even if this only matters for class objects, it is a good habit to use prefix increment/decrement at all times.

2.7.5 Write conditional expressions like: if (6 == errorNum) ...

This style avoids accidental assignments of the variable when the comparison operator is written with only one equal sign (=). Do not rely on implicit conversion to bool in conditions.

```
if (ptr) // wrong
if (ptr != NULL) // ok
if (ptr != nullptr) // even better (C++11)
```

Listing 42: Conditional expressions.

2.7.6 USE THE NEW CAST OPERATORS

Use `dynamic_cast`, `const_cast`, `reinterpret_cast` and `static_cast` instead of the traditional C cast notation. These document better what is being performed.

- Use `static_cast` as the equivalent of a C-style cast that does value conversion, or when you need to explicitly up-cast a pointer from a class to its superclass.
- Use `const_cast` to remove the `const` qualifier.
- Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.
- Do not use `dynamic_cast` except in test code. If you need to know type information at runtime in this way outside of a unit test, you probably have a design flaw.

2.8 OTHER RECOMMENDATIONS

2.8.1 USE OF BOOST LIBRARIES IS ENCOURAGED

Boost[9] is a set of free, expertly designed, peer–reviewed portable C++ source libraries. Boost provides reference implementations that are suitable for eventual standardization. Actually, some of the Boost libraries are already included in the current C++ standard and several more are expected to be included in the new standard now being developed.

2.8.2 USE COMMON SENSE AND BE CONSISTENT

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you are saying, rather than on how you are saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a file looks drastically different from the existing code around it, the discontinuity throws readers out of their rhythm when they go to read it. Try to avoid this.

[9] See `http://www.boost.org`