Draft version 0.10

ATLAS NOTE

May 21, 2015

ATLAS LAr Calorimeter trigger electronics phase I upgrade: LDPB Firmware Development Environment

Nicolas Chevillot[a]

[a] *LAPP – CNRS – France*

Abstract

This document describes the working environment for the development of the LDPB firmware.

Contents

1 Introduction

In projects involving multiple contributors, it is important to have a solid way of sharing the work done. It is also important to track the progress and bugs in the software.

There exist many version control tools on the market now, GIT is one of them. It has been chosen as a base for the development of the LDPB firmware. It allows to share the code between different members and track the different versions of the software.

There are also numerous bug tracking system available, JIRA has been chosen.

Both tools are available from CERN. Figure 1 shows how all elements are connected. The main elements are:

- 'atlas-lar-ldpb-firmware' e-group: only members of the 'atlas-lar-ldpb-firmware' e-group are allowed to use other parts, see section 2.2.
- 'atlas-lar-ldpb-firmware' GIT repository: used to store and track history of the LDPB firmware, see section 2.3.
- 'gitolite-admin' GIT repository: used to configure 'atlas-lar-ldpb-firmware' GIT repository access rights, see section 2.4.
- 'ATLAS LAr LDPB Firmware' JIRA project: used to track issues, see section 2.5.

This document describes how to register to the necessary e-group and get access to both JIRA and GIT repositories needed for the ATLAS LDPB firmware. It also describes the working environment and how to simulate and compile a design.

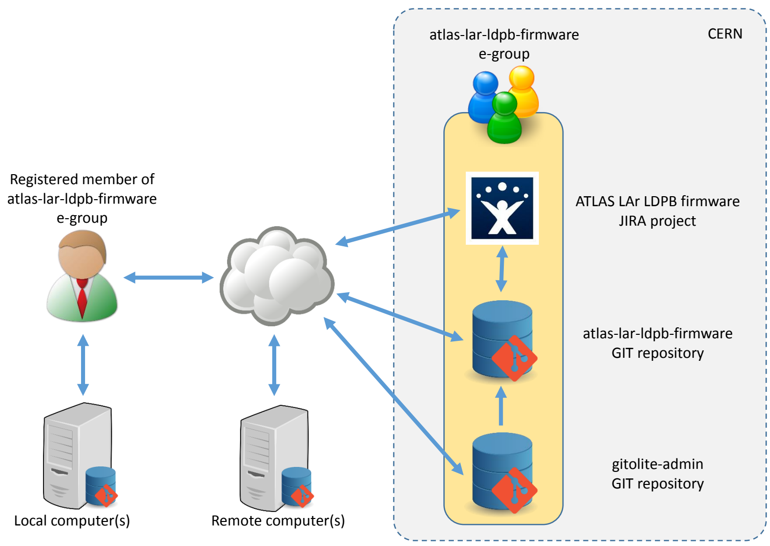

Figure 1: Environment overview

2 Working environment

2.1 Introduction

As described in previous figure (figure 1), the working environment is made out of e-Group, GIT repositories and JIRA project. This section describes how they are connected together.

All the tools and resources used in this project are CERN based. This is to allow everyone to access it from anywhere in the world. Of course you need to have a CERN login to use it.

2.2 e-Group

All accesses to GIT repositories as well as JIRA project is restricted through e-Group membership. In order to work with this project, you need to be a member of the following e-Group:

- $atlas - lar - ldpb - firmware$

In order to register to this e-group, go to https://e-groups.cern.ch and search for '$atlas - lar - ldpb - firmware$'. On the members tab, you can request your name to be added to the list. You can also contact the e-Group owner:

- Nicolas Chevillot: nicolas.chevillot@lapp.in2p3.fr

2.3 GIT repository

The source code for the ATLAS LAr LDPB firmware is stored on the GIT repository. The workflow is described in section 2.3.1.

The repository can be accessed directly through the CERN Web interface:

- https://git.cern.ch/web/atlas-lar-ldpb-firmware.git/tree

You will need to use CERN credentials to access the repository. This interface allows you to browse the different commits that were made, browse the code and download a snapshot. It is not intended really to develop any code with this interface and command line or Tortoise should be used instead.

2.3.1 Workflow

The workflow is shown on figure 2. It is the most simple way to work with GIT. The choice for this simple workflow is driven by the fact that there are a limited number of developers and we don't want to have more process than required.

There is only one branch used for development, it is called '*master*'. If we see that this is too simple, we can introduce the branch concept. However this would require a bit more handling. The following operations are used:

- clone: make a local copy of the CERN GIT repository, described in section 2.3.3.
- add: add changes to a commit list, described in section 2.3.4.
- commit: stores local changes, described in section 2.3.7.
- push: stores local commits to the CERN GIT repository, described in section 2.3.8.
- pull: retrieves changes made on the CERN GIT repository to the local repository, described in section 2.3.9.

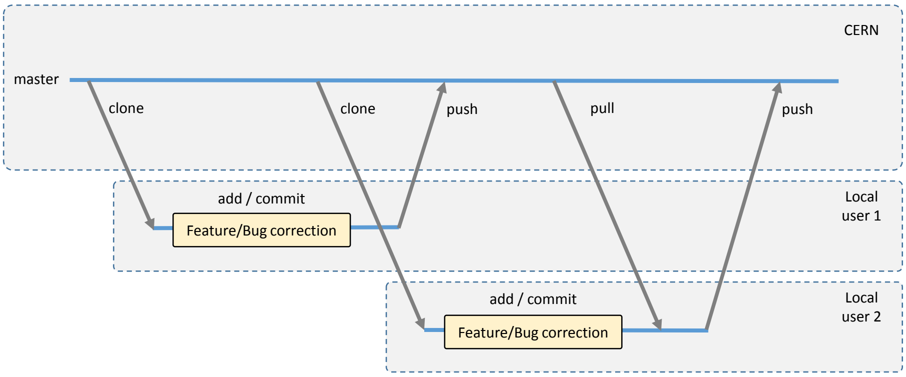

Figure 2: GIT workflow

Whether you use the command line GIT or Tortoise GIT GUI is up to you. To setup GIT command line, see section 2.3.2.

2.3.2 GIT command line setup

GIT needs some basic configuration to know who you are and to configure diff tools. You should read the official GIT documentation [1].

Configure your name and email:

```
$ git config --global user.name "Surname Name"
$ git config --global user.email email@address
```

Table 1: GIT Setup for user name and email

In order to commit changes, you will need to enter a description of the changes. By default '*vi*' tool is used. If you would like another editor by default, you should use the '*config*' command to change the '*core.editor*' global parameter.

- If you are working on Windows, you can use '*Notepad++*' [2] which is free and configure GIT with the command as shown in table 2. You also need to create a wrapper file in your editor directory as shown in table 3, this file should be named '*npp.sh*' or whatever name you have configured '*core.editor*' to. Make sure this file does not have any newline after the last command, i.e. it should have 2 lines only.

```
$ git config --global core.editor '"/cygdrive/c/Program Files (x86)/
   Notepad++/npp.sh"'
```

Table 2: GIT Setup for default editor, Notepad++

```
#!/bin/sh
"/cygdrive/c/Program Files (x86)/Notepad++/notepad++.exe" -multiInst -
   nosession -noPlugin "$(cygpath -w $1)"
```

Table 3: GIT Setup for default editor, wrapper file 'npp.sh' for Notepad++

- If you are working on Linux (or Unix environment), you can use '*nedit*' which is usually installed and configure GIT with the command as shown in table 4.

```
$ git config --global core.editor 'nedit'
```

Table 4: GIT Setup for default editor, nedit

In order to check what changes you made to the sources, you should setup a *diff* tool. There exists many *diff* tools with advantages and disadvantages. Also you should setup a tool to resolve conflicts when merging files.

- If you are working on Windows, you can use '*Meld*' [3] which free to use. In order to configure GIT to use '*Meld*' for both diff and merges, you should use the command as shown in table 5

```
$ git config --global diff.tool meld
$ git config --global difftool.prompt false
$ git config --global difftool.meld.cmd '"/cygdrive/c/Program Files (
    x86)/Meld/meld/meld.exe" $(cygpath -w $LOCAL) $REMOTE'
$ git config --global merge.tool meld
$ git config --global mergetool.meld.path '/cygdrive/c/Program Files (
    x86)/Meld/meld/meld.exe'
```

Table 5: GIT Setup for diff/merge tool, using Meld on Cygwin

- If you are working on Linux (or Unix environment), you can use '*Meld*' [3] which free to use. In order to configure GIT to use '*Meld*' for both diff and merges, you should use the command as shown in table 6 or 7 for '$lappc-f561$'.

```
$ git config --global diff.tool meld
$ git config --global difftool.prompt false
$ git config --global difftool.meld.cmd 'meld $LOCAL $REMOTE'
$ git config --global merge.tool meld
$ git config --global mergetool.meld.path 'meld'
```

Table 6: GIT Setup for diff/merge tool, using Meld on Linux

```
$ git config --global diff.tool meld
$ git config --global difftool.prompt false
$ git config --global difftool.meld.cmd '/usr/bin/python /usr/bin/meld
    $LOCAL $REMOTE'
$ git config --global merge.tool meld
$ git config --global mergetool.meld.path '/usr/bin/python /usr/bin/
    meld'
```

Table 7: GIT Setup for diff/merge tool, using Meld on Linux (special case for lappc-f561)

2.3.3 GIT Clone

As the GIT repository is stored at CERN, you will need to clone the repository to your local working machine. This machine can actually be a remote computer but you need to have a copy the GIT repository somewhere. You should not have to do this operation many times unless you want to clone the repository on another machine or directory. To retrieve changes made by other users on the CERN GIT repository, you should use the *pull* command described in section 2.3.8.

You should create a directory named *git* (or any other you like) to store the repository.

2.3.3.1 Using command line

From the Linux or Cygwin command line, change the path to the *git* directory you have created. On Cygwin, mounted disks are available as */cygdrive/letter*.

```
user@machine ~
$ mkdir /cygdrive/d/Users/Chevillot/Projects/ATLAS/git
$ cd /cygdrive/d/Users/Chevillot/Projects/ATLAS/git
user@machine@user@machine /cygdrive/d/Users/user/Projects/ATLAS/git
$ git clone https://git.cern.ch/reps/atlas-lar-ldpb-firmware
Cloning into 'atlas-lar-ldpb-firmware'...
Username for 'https://git.cern.ch':
Password for 'https://user@git.cern.ch':
remote: Counting objects: 7400, done.
remote: Compressing objects: 100% (4556/4556), done.
remote: Total 7400 (delta 4425), reused 4269 (delta 2423)
Receiving objects: 100% (7400/7400), 41.95 MiB | 9.16 MiB/s, done.
Resolving deltas: 100% (4425/4425), done.
Checking connectivity... done.
Checking out files: 100% (2851/2851), done.
$
```

Table 8: GIT Clone repository using command line

If you want to use the Tortoise GIT GUI on a repository that was retrieved using the command line, you will need to configure your repository to ignore changes in file modes. There is actually a problem in the GIT software for windows. This is shown in table 9.

```
user@machine@user@machine /cygdrive/d/Users/user/Projects/ATLAS/git
$ git config core.filemode false
$
```

Table 9: GIT repository ignore file mode changes using command line

2.3.3.2 Using Tortoise GIT interface

Open an explorer and right-click on the *git* directory you have created. A contextual-menu should open, select *Git Clone...*:

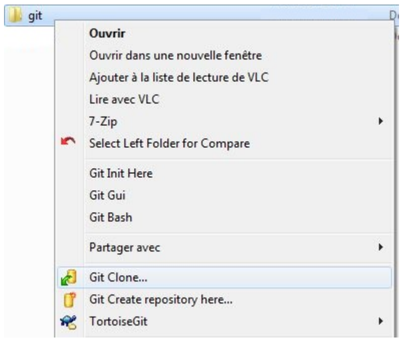

Figure 3: Tortoise GIT cloning repository step 1

Fill-in the URL for the repository and make sure the directory is the one you want.

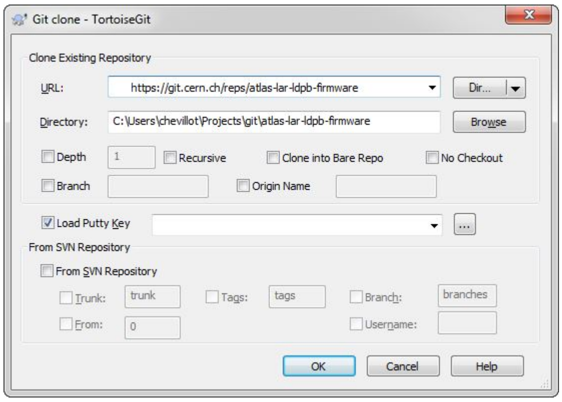

Figure 4: Tortoise GIT cloning repository step 2

Enter you CERN username and password:

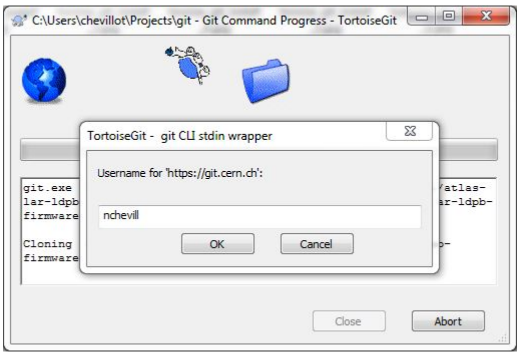

Figure 5: Tortoise GIT cloning repository step 3

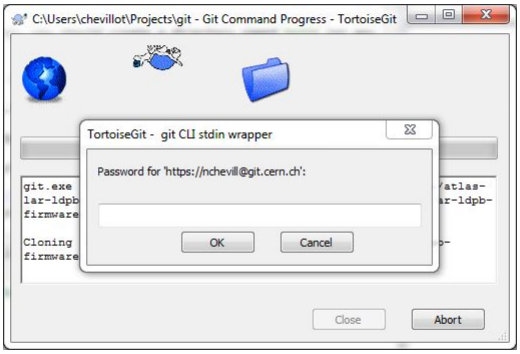

Figure 6: Tortoise GIT cloning repository step 4

Tortoise GIT should retrieve the repository now:

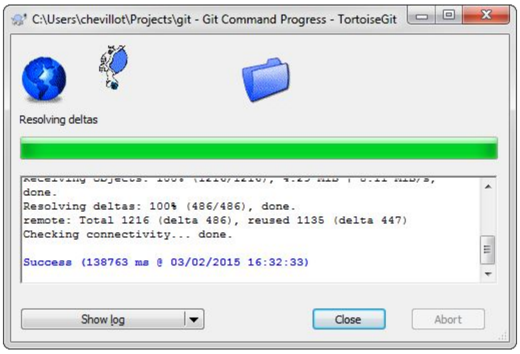

Figure 7: Tortoise GIT cloning repository step 5

2.3.4 GIT Add

In order to commit changes to the local GIT repository, you first need to add files into a list of files. This list of files will be the one used to commit your changes. The commit operation is described in section 2.3.7.

2.3.4.1 Using command line

The GIT *status* command should be used to check what files are currently in the commit list or untracked. For example in table 10, a file name *test_file.vhd* was created. This file is not yet versioned therefore is shown as untracked. In order to include it in the versioning, the GIT *add* command should be used as shown on table 11. As a result, the new file is shown in the list of changes to be committed as shown in table 12.

```
user@machine@user@machine /cygdrive/d/Users/user/Projects/ATLAS/git
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
(use "git add <file>..." to include in what will be committed)

test_file.vhd

nothing added to commit but untracked files present (use "git add" to
   track)
$
```

Table 10: GIT repository status before adding file using command line

```
user@machine@user@machine /cygdrive/d/Users/user/Projects/ATLAS/git
$ git add test_file.vhd
$
```

Table 11: GIT repository status using command line

```
user@machine@user@machine /cygdrive/d/Users/user/Projects/ATLAS/git
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

new file:   test_file.vhd
$
```

Table 12: GIT repository status before adding file using command line

If you have modified a file that is already versioned, it will be shown as a change not staged for commit as shown in table 13. You should use the GIT *add* command the same way to add it to the list of committed changes. After this is done, the file should be seen as 'to be committed' as shown in table 14.

```
user@machine@user@machine /cygdrive/d/Users/user/Projects/ATLAS/git
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working
    directory)

modified:   ttc_arch.vhd

no changes added to commit (use "git add" and/or "git commit -a")
```

Table 13: GIT repository status before adding versioned file using command line

```
user@machine@user@machine /cygdrive/d/Users/user/Projects/ATLAS/git
$ git add ttc_arch.vhd
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified:   ttc_arch.vhd
```

Table 14: GIT repository status after adding versioned file using command line

2.3.4.2 Using Tortoise GIT interface

You can check what files have been created/modified in the repository using the 'check for modifications' command. Right-click on the repository directory and select the command from the context menu as shown on figure 8. Tortoise GIT will show the list of modified files in the repository as shown on figure 9 where a new file has been created. Make sure the 'Show unversioned files' option is check to be able to see files you have created.

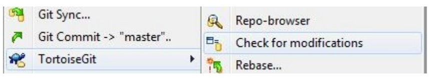

Figure 8: GIT repository check for modifications context menu using Tortoise GIT

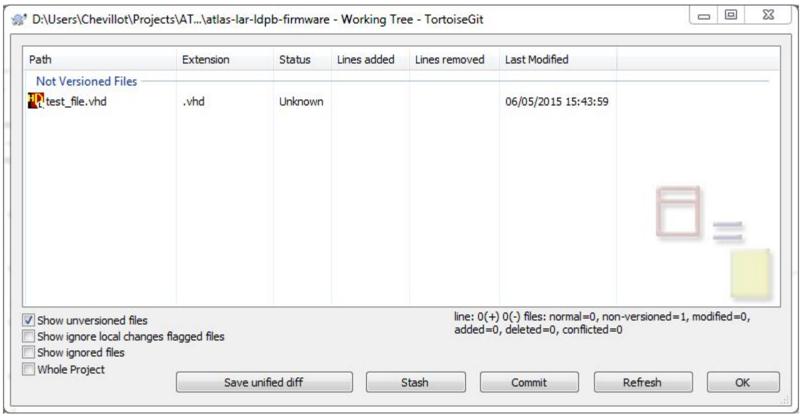

Figure 9: GIT repository status before adding file using Tortoise GIT

You can add the created file into the versioning system right-clicking on it and selecting ’Add’ from the context menu as shown on figure 10. Alternatively from the explorer you can also right-click on the file you want to add and choose ’Tortoise GIT’-’Add’.

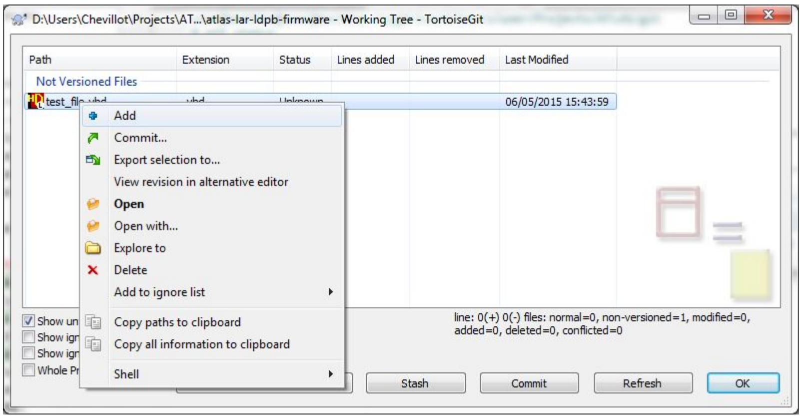

Figure 10: GIT repository status before adding file using Tortoise GIT

If you have modified a file that is already versioned, it will be shown as a modified file. There is no need to add the file to the commit list, Tortoise GIT automatically adds modified files into its list of files to be committed.

2.3.5 GIT Cancel changes

2.3.5.1 Using command line

If you made changes to a versioned file which you do not want to commit and want to go back to the non-modified content, you can use the ’*checkout*’ GIT command as shown in table 15. This example is valid only if you haven’t added this modified file to the list of files to be committed. In this case, see the next example.

```
user@machine /cygdrive/d/Users/user/Projects/ATLAS/git/atlas-lar-ldpb-
    firmware
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working
    directory)

modified:   LATOME/src/ttc/ttc_arch.vhd

no changes added to commit (use "git add" and/or "git commit -a")
$ git checkout LATOME/src/ttc/ttc_arch.vhd
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
$
```

Table 15: Cancel changes on modified file using command line

If you have already added the modified file to the list of files to be committed then you need first to remove it from the list of files to be committed using '*git reset HEAD file*' and then cancel the changes you have made as shown in table 16.

```
user@machine /cygdrive/d/Users/user/Projects/ATLAS/git/atlas-lar-ldpb-
    firmware
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified:   LATOME/src/ttc/ttc_arch.vhd
$ git reset HEAD LATOME/src/ttc/ttc_arch.vhd
Unstaged changes after reset:
M       LATOME/src/ttc/ttc_arch.vhd
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working
    directory)

modified:   LATOME/src/ttc/ttc_arch.vhd

no changes added to commit (use "git add" and/or "git commit -a")
$ git checkout LATOME/src/ttc/ttc_arch.vhd
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
$
```

Table 16: Cancel changes on modified file already in commit list using command line

If you have already committed your changes but not pushed it to the remote repository you can revert your commit using the '*reset*' command as shown in table 17.

```
$ git commit
[master 812ce27] test
1 file changed, 2 insertions(+)

$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
nothing to commit, working directory clean

$ git reset --soft HEAD^

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified:   ttc_arch.vhd
```

Table 17: Cancel commit on local repository using command line

2.3.5.2 Using Tortoise GIT interface

In order to revert changes you have made to a versioned file using the Tortoise GIT interface, you need to right-click on the file in the explorer and choose the '*Revert*' option from the contextual menu as shown on figure 11.

Figure 11: Cancel modifications on versioned file using Tortoise GIT

If you have already committed your changes but not pushed it to the remote repository you can revert your commit using the '*reset*' command. You should first open the Tortoise GIT log window, right-click on your repository directory in the windows explorer and select as shown on figure 12. In the list of commits done on your repository, select the commit to which you want to revert to as shown on figure 13, right click and select '*Reset"master"tothis..*'. You should choose the '*soft*' reset as shown on figure 14.

Figure 12: Tortoise GIT show log context menu

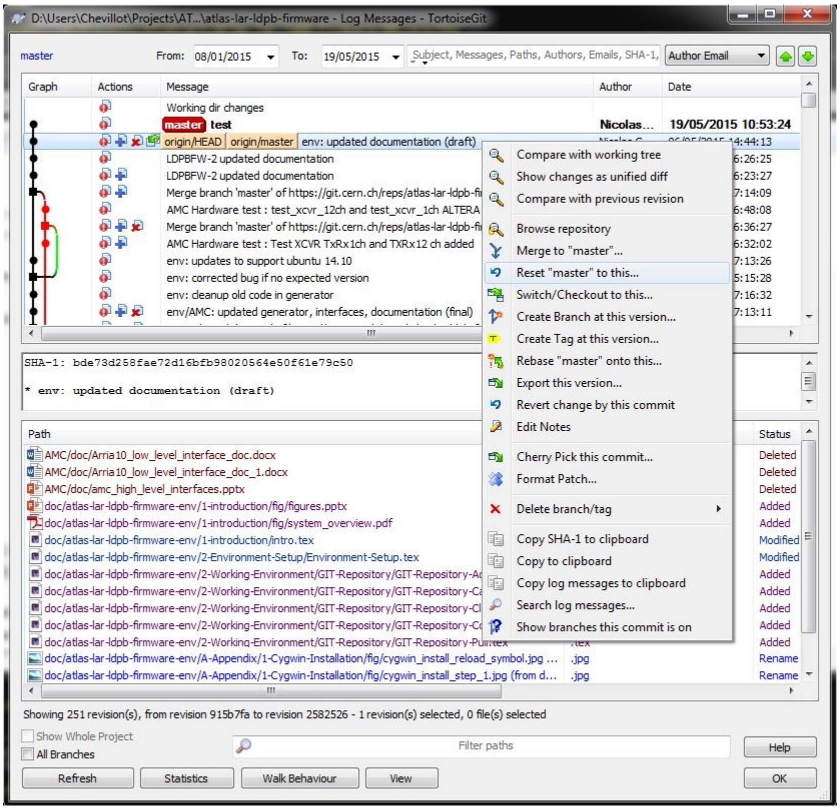

Figure 13: Tortoise GIT reset master to this

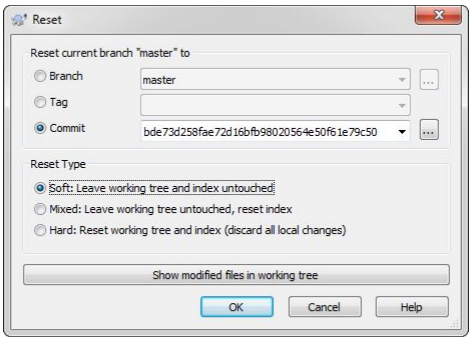

Figure 14: Tortoise GIT reset master to this, soft reset

2.3.6 GIT Reviewing changes

Reviewing file changes is one of the most important step before committing the changes. A good practice would be to have another peer review your changes before you commit. However there is no simple way to do it considering we use here GIT without branches.

2.3.6.1 Using command line

In order to review all changes made to the local repository, you can use the '*status*' and '*difftool*' GIT command. If you've configured GIT to use a graphical tool, it will automatically be opened, please check section 2.3.2.

Table 18 shows an example where one file named ’*LATOME/src/ttc/ttc_arch.vhd*’ has been modified. Figure 15 shows the *meld* tool that is opened on this file.

```
user@machine /cygdrive/d/Users/user/Projects/ATLAS/git/atlas-lar-ldpb-
    firmware
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working
    directory)

modified:   LATOME/src/ttc/ttc_arch.vhd

no changes added to commit (use "git add" and/or "git commit -a")
$ git difftool LATOME/src/ttc/ttc_arch.vhd
$
```

Table 18: GIT status and difftool to review changes on modified file using command line

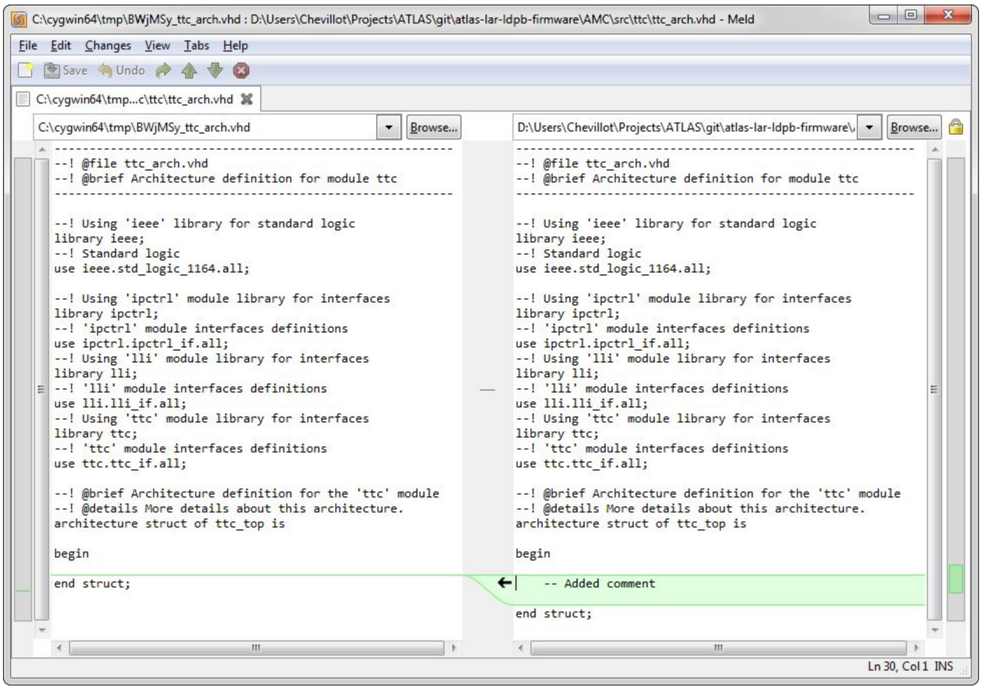

Figure 15: Meld review on modified file using command line

2.3.6.2 Using Tortoise GIT interface

To review the changes you have made to a file using the Tortoise GIT interface, right-click on the file and select from the Tortoise GIT menu, ’*Diff*’ option as shown on figure 16. This will open the diff tool on the file as shown on figure 17.

Figure 16: Review changes contextual menu using Tortoise GIT

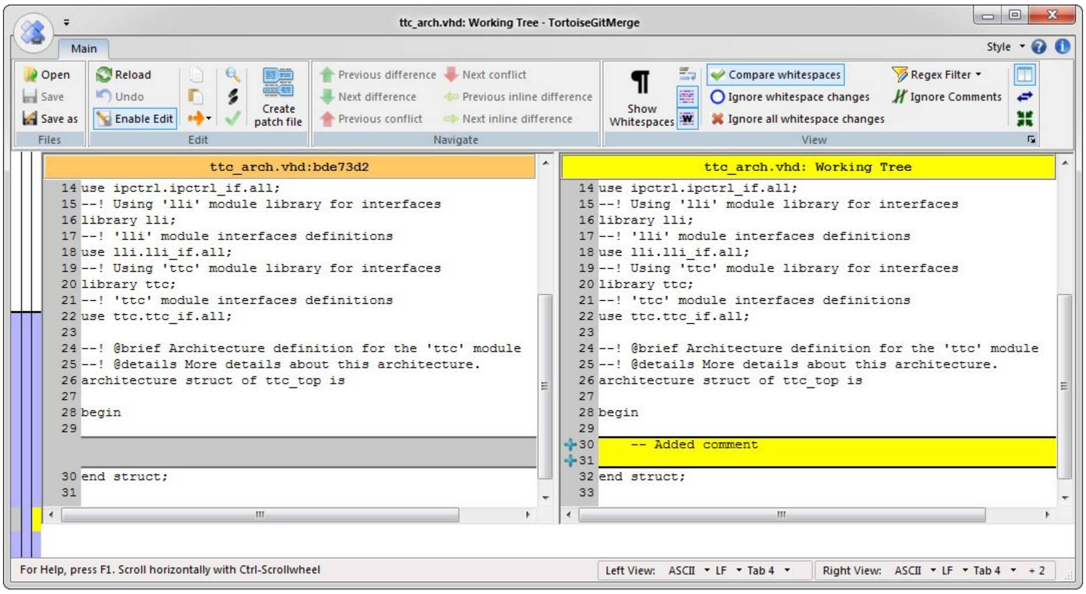

Figure 17: Review tool on modified file using Tortoise GIT

2.3.7 GIT Commit

Once all the files you want to commit have been added to the commit list (see section 2.3.4), you need to commit your changes. Those changes are first committed on your local repository and until you push your changes to the remote repository, they remain local, see section 2.3.8.

2.3.7.1 Using command line

The GIT command '*commit*' should be used to commit your changes to the local repository. You can check what will be committed with the GIT '*status*' command. Table 19 shows an example.

The GIT '*commit*' command will start an editor for you to enter a description of the commit. It should be as self-explanatory as possible while short. By default the editor started is '*vi*' which can be user-unfriendly. Please refer to section 2.3.2 in order to configure your own preferred editor.

If you want your commit to be linked to a JIRA issue, you should make sure your comments are following the description in section 2.5.4. In any case the following comment format should be kept: *(Component) Comments*

- Component: Identifies the JIRA project component it applies to, this is only for clarification so we can identify easier which part of the code it belongs to. You can find all JIRA components for the project on the JIRA webfront [5].

- Comments: Explains what is being done.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified:   ttc_arch.vhd
$
```

Table 19: Example use of GIT commit

2.3.7.2 Using Tortoise GIT interface

In order to commit your changes, you should open the '*checkformodifications*' windows as shown on figure 18. There are other ways to commit, this is just one method. Make sure all the files you want to commit are in the list and push the '*commit*'. This will open a '*commit*' dialog where you should enter the comments for your commit as shown on figure 19.

If you want your commit to be linked to a JIRA issue, you should make sure your comments are following the descritpion in section 2.5.4.

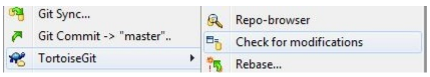

Figure 18: GIT repository check for modifications context menu using Tortoise GIT

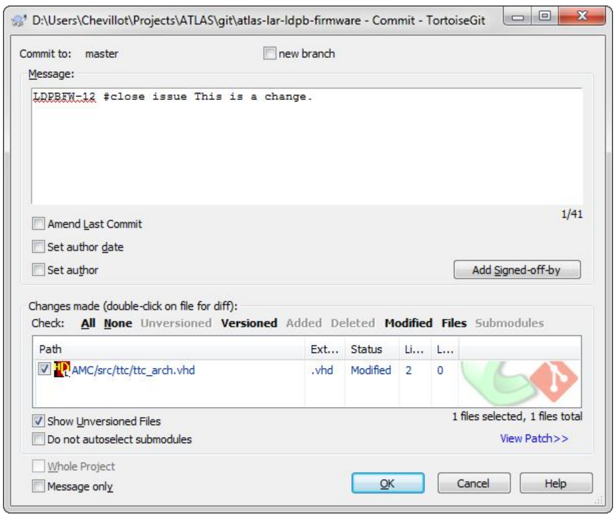

Figure 19: GIT commit using Tortoise GIT

2.3.8 GIT Push

All your committed changes on the local repository are not available to others until you push your commits to the remote repository. This is done with the '*push*' command.

2.3.8.1 Using command line

The GIT '*push*' command is used to push your local commits to the remote repository as shown on table 20. In case there are commits on the remote repository which you have not already retrieved, the push will fail. You first need to pull the remote changes and the push again. This is described in section 2.3.9.

```
$ git commit
[master aafee42] This is a test
1 file changed, 2 insertions(+)

$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
nothing to commit, working directory clean

$ git push
Username for 'https://git.cern.ch':
Password for 'https://nchevill@git.cern.ch':
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 510 bytes | 0 bytes/s, done.
Total 6 (delta 5), reused 0 (delta 0)
To https://git.cern.ch/reps/atlas-lar-ldpb-firmware
0792182..aafee42  master -> master
```

Table 20: GIT push using command line

2.3.8.2 Using Tortoise GIT interface

In order to push your commits to the remote repository, you need to right-click on your local repository directory in the explorer and choose '*Pull*' as shown on figure 20. You should keep the default settings on the '*Pull*' dialog shown on figure 21 and select '*OK*'. An example of a push is shown on figure 22.

Figure 20: Tortoise GIT push context menu

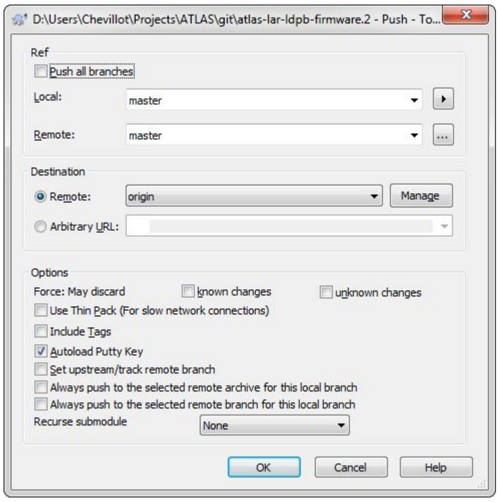

Figure 21: Tortoise GIT push dialog

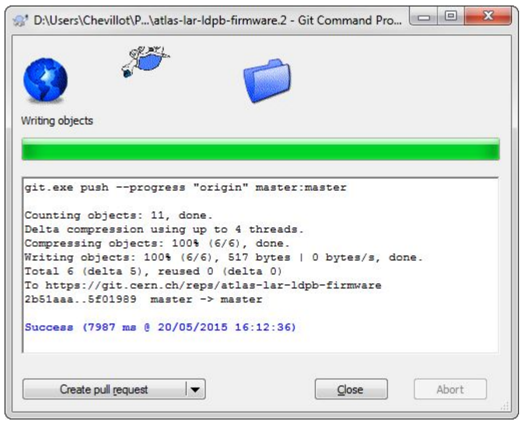

Figure 22: Tortoise GIT push result

2.3.9 GIT Pull

If you want to retrieve changes that were made by other team members, or just want to update your local repository, you should use the '*pull*' command. This will retrieve all changes on the remote repository and store it on your local repository. If you have made changes on files that were also modified by others, you will need to resolve the conflicts.

2.3.9.1 Using command line

- The GIT '*pull*' command is used to update the local repository to the latest changes on the remote repository as shown in table 21.

```
$ git pull
Username for 'https://git.cern.ch':
Password for 'https://nchevill@git.cern.ch':
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From https://git.cern.ch/reps/atlas-lar-ldpb-firmware
0792182..aafee42  master     -> origin/master
Updating 0792182..aafee42
Fast-forward
LATOME/src/ttc/ttc_arch.vhd | 2 ++
1 file changed, 2 insertions(+)
```

Table 21: GIT pull using command line

- If you want to pull a specific commit, you should first identify which commit you want. To do this, you can browse on the GIT Webfront and find out the commit SHA1 key [4]. If you click on the commit comments (and not the committer name), you should find the commit SHA1 key on the right of the 'commit', below 'committer'. Then from the command line, you need to use the '*reset*' command as shown on table 22.

```
$ git reset --hard 9e75595db2fc24e5df57ca05c5fe6cdff7cbde12
HEAD is now at 9e75595 LDPBFW-8 #start progress Updated documentation
    for JIRA and GIT
$
```

Table 22: GIT repository pull specific commit using command line

- If you have made changes to one or more files that were also modified on the remote repository, and you have committed your changes, then GIT will claim you need to resolve conflicts. You can use the '*mergetool*' command to do so as shown in table 23. The tool you have configured for the merge will be started and you should resolve the conflict there. An example using '*meld*' is shown in figure 23. In this example, a commit in the local repository added the line '–comment2' but this file was also modified on the remote repository with the line '–comment 1', in this case the merge is rather simple and both lines should be kept. There are no rules for merging, you need to understand the changes and what they are meant for, if you don't, you should ask whoever made the changes.

```
$ git pull
Username for 'https://git.cern.ch':
Password for 'https://nchevill@git.cern.ch':
Auto-merging LATOME/src/ttc/ttc_arch.vhd
CONFLICT (content): Merge conflict in LATOME/src/ttc/ttc_arch.vhd
Automatic merge failed; fix conflicts and then commit the result.

$ git mergetool
Merging:
LATOME/src/ttc/ttc_arch.vhd

Normal merge conflict for 'LATOME/src/ttc/ttc_arch.vhd':
{local}: modified file
{remote}: modified file
```

Table 23: GIT pull with conflict using command line

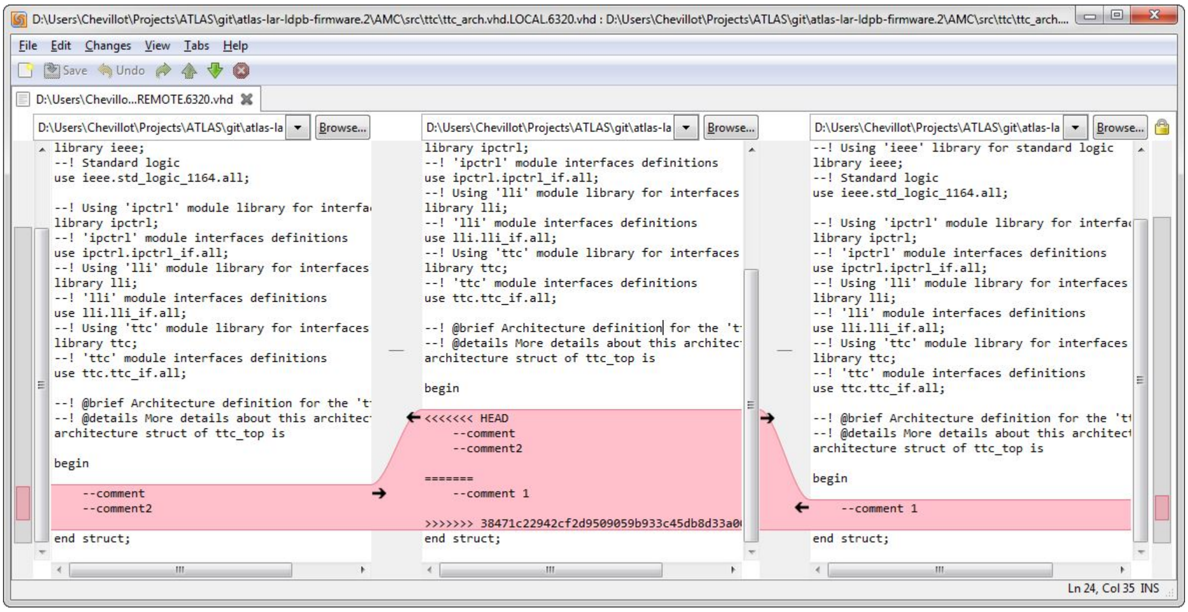

Figure 23: GIT resolve conflicts using meld using command line

2.3.9.2 Using Tortoise GIT interface

- In order to pull the latest changes from the remote repository, you should right-click on the local repository directory in the explorer and select the Tortoise GIT '*Pull*' from the context menu as shown on figure 24. Click '*OK*' on the next dialog to choose default settings as shown on figure 25. Tortoise GIT will retrieve the latest changes as shown on figure 26.

Figure 24: Tortoise GIT pull context menu

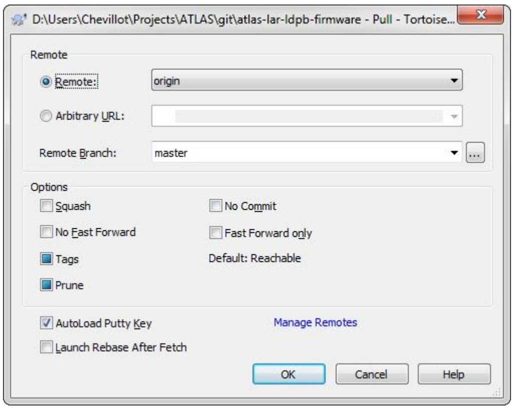

Figure 25: Tortoise GIT pull dialog

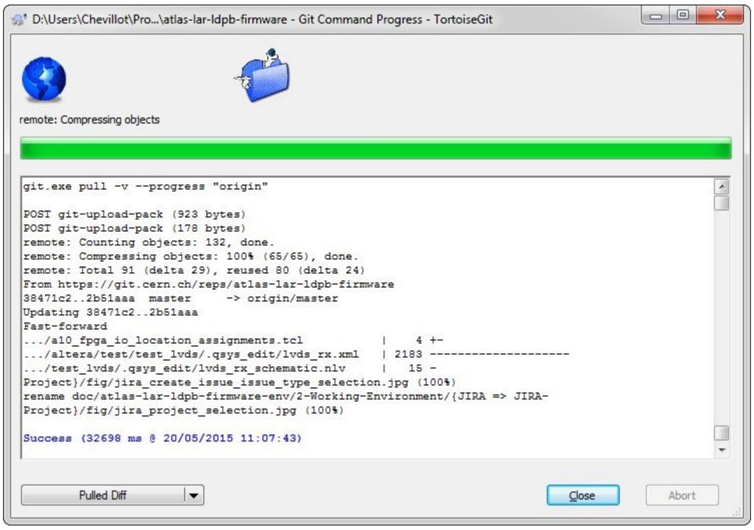

Figure 26: Tortoise GIT pull result

- If you want to retrieve a specific commit, you should right-click on the local repository directory in the explorer and choose the '*Switch/Checkout...*' command as shown on figure 27. You should identify which commit you want. To do this, you can browse on the GIT Webfront and find out the commit SHA1 key [4]. If you click on the commit comments (and not the committer name), you should find the commit SHA1 key on the right of the 'commit', below 'committer'. Then from the '*Switch/Checkout...*' dialog, you should enter the commit SHA1 key in the commit field and deselect 'Create branch' as shown on figure 28 and 29.

Figure 27: Tortoise GIT switch context menu

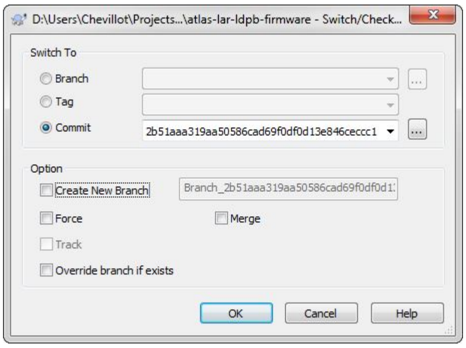

Figure 28: Tortoise GIT pull specific commit, switch dialog

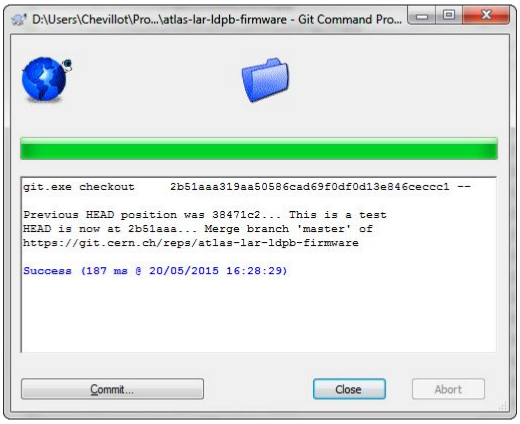

Figure 29: Tortoise GIT pull specific commit, switch result

- If you have made changes to one or more files that were also modified on the remote repository, and you have committed your changes, then Tortoise GIT will not succeed pushing the commits as shown on figure 30. You shoud click on the '*Pull*' button and proceed through the update to the top of the remote repository as described in the previous bullet. It will also fail showing conflicted files as shown on figure 31. The next step is to resolve the conflicts, right-click on your local repository in the explorer and select '*Resolve*' as shown on figure 32. Tortoise GIT will display a dialog box with all files that are in conflict and need to be resolved as shown on figure 33. If you right click on a file, you will have several choices:
  - Edit conflicts: starts the merge tool as shown on figure 34.
  - Resolved: You need to resolve the conflict using your own tool before using this option. Once you have resolved the conflict, you can tell GIT it is resolved.
  - Resolved conflict using 'theirs': does not start any merge tool, use the version from the remote repository.
  - Resolved conflict using 'yours': does not start any merge tool, use your version.

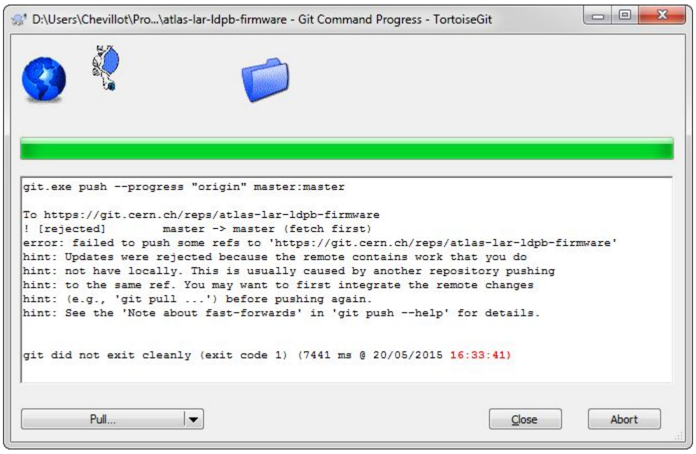

Figure 30: Tortoise GIT push failed

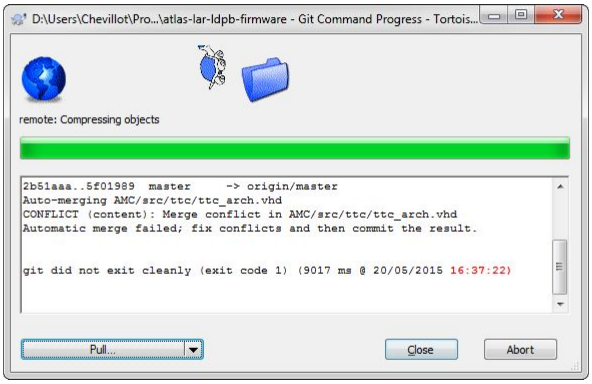

Figure 31: Tortoise GIT pull failed

Figure 32: Tortoise GIT reverse context menu

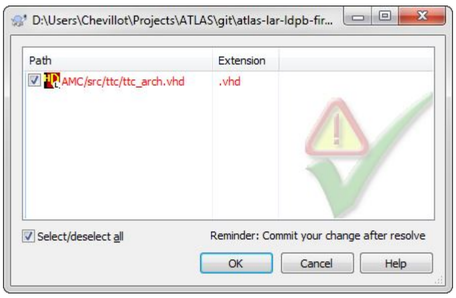

Figure 33: Tortoise GIT reverse

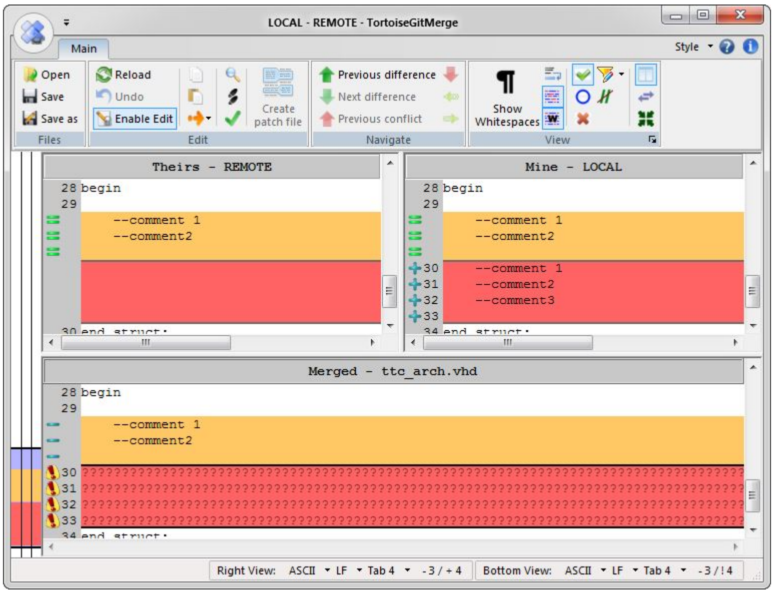

Figure 34: Tortoise GIT resolve

2.4 Access rights GIT repository

The '$gitolite-admin$' GIT repository is a special shared repository that takes care of access rights for other repositories at CERN. It allows to set read/write accesses to directories and files within our '$atlas-lar-ldpb-firmware$' repository.

By default you will have read-only access to the repository. Please contact one of the e-group owner if you require to have write access to one or more directories, see section 2.2. If you try to commit changes in a directory for which you are not allowed, the *push* operation will fail.

The read/write access rights are controlled in the following file:

- $gitolite-admin/conf/subs/atlas-lar-ldpb-firmware.conf$ [9]

2.5 JIRA project

One important aspect of working in a team is to be able to share information and keep track of progress. JIRA is a tool that can be used for multiple reasons, one of them is bug tracking. You can find more informations about JIRA on the website [6].

The JIRA project can be accessed from the following link: https://its.cern.ch/jira/browse/LDPBFW/?selectedTab=com.atlassian.jira.jira-projects-plugin:summary-panel [5].

2.5.1 Navigate through the project

The top banner of the JIRA interface allows to select which project you want to open. You should select the 'ATLAS LAr LDPB Firmware' project as show on figure 35.

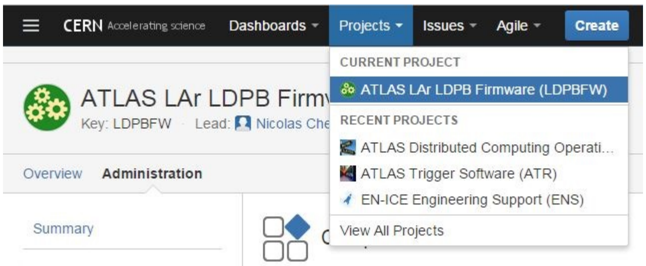

Figure 35: JIRA project selection

2.5.2 Creating an issue

In order to create an issue, you should click on the 'Create' button on the top banner as shown on figure 35. It will open a 'Create Issue' dialog as shown on figure 36. JIRA can handle many types of issues, not only a bug. We will use the following issue types:

- Task: describes a task that needs to be done. For example a workpackage, a new feature, something not yet existing or not fully implemented.
- Bug: describes a problem on the existing code, something that needs to be fixed.

Select the issue type you want to create and click '*Next*'. The next dialog has a lot of fields, some mandatory and others not, this is shown on table 24.

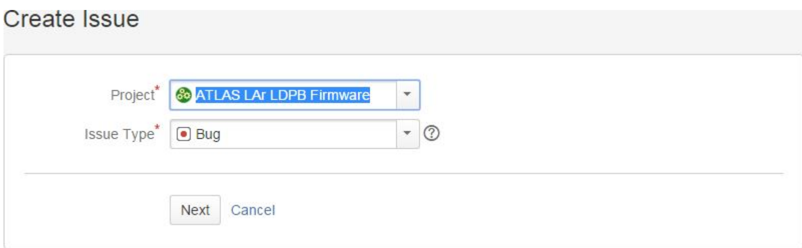

Figure 36: JIRA create issue, issue type selection

Field	Description
Summary	You need to enter a brief description of the issue you are creating. This should be self-explanatory.
Priority	Assess how important is this issue and choose from the following:
	Trivial: The issue is specific to non critical parts of the code. Minor: The issue is not blocking anyone to work and there exist a workaround. Major: The issue is rather important and needs to be fixed in a reasonable timeframe. The project is not working properly but there exist a workaround that allows other team members to continue to work. Critical: The issue is important and needs to be fixed as soon as possible. The project is not working at all but there exist a workaround that allows other team members to partially work. Blocker: The issue is very important and needs to be fixed as soon as possible. Other teams members cannot work due to this problem. Needs Decision: if you are not sure about how important is this issue, you should use this option. The issue will be discussed then within the team.
Due Date	If the issue needs to be resolved before a specific date, you should enter it here. If not leave it empty.
Component/s	Select in the list the component to which the issue applies. The assignee for this issue will be automatically selected depending on the component. You can select multiples components if you need to.
Approver	Not used, leave empty.
Affects Version/s	Not used yet, leave empty.
Fix Version/s	Not used, leave empty.
Assignee	The person responsible for handling the issue, this is automatically set to the component lead. You need to specify therefore the component it applies to. You can also assign the issue to someone specific, however this should be agreed with the component lead.
Reporter	This is you, leave it as it is.
View Access	Not used, leave it empty.
Description	You should describe the issue you want to create. This should be as detailed as possible explaining the use-case, software version, environment used, ...
Original Estimate	Not used, leave empty.
Remaining Estimate	Not used, leave empty.
Attachment	You can attach files to the issue, this could be waveforms or code
Labels	Not used, leave empty.
Units	Not used, leave empty.
Percent Done	Not used, leave empty.
Due Time	Not used, leave empty.

Table 24: JIRA create issue fields (part)

Field	Description
Epic Link	Not used, leave empty.
ServiceNow ID	If the issue is linked to a CERN Service Request/Incident, you should enter the ID number.
External Reporter	Not used, leave empty.
Story Points	Not used, leave empty.
Planned Release	Not used, leave empty.
External Issue ID	Not used, leave empty.
External Issue URL	Not used, leave empty.
Savannah Fields Container	Not used, leave empty.
Sprint	Not used, leave empty.
Group Watchers	Not used, leave empty.
Technical Stop	Not used, leave empty.
Reported by standby service	Not used, leave empty.
Original Reporter	Not used, leave empty.
Planned Start	Not used, leave empty.
Planned End	Not used, leave empty.

Table 24: JIRA create issue fields

2.5.3 Issue workflow

Each created issue has 4 different states, shown in figure 37.

- Open: Issue is not handled by anyone at the moment.
- In progress: The issue is taken care of and work is being done.
- Reopened: The issue has been reopened after it was closed due to incomplete work.
- Closed: The issue is closed and work was done.

You should take care of changing issue state according to what is being done. If you are not working on the issue, it should be in 'Open' state for example.

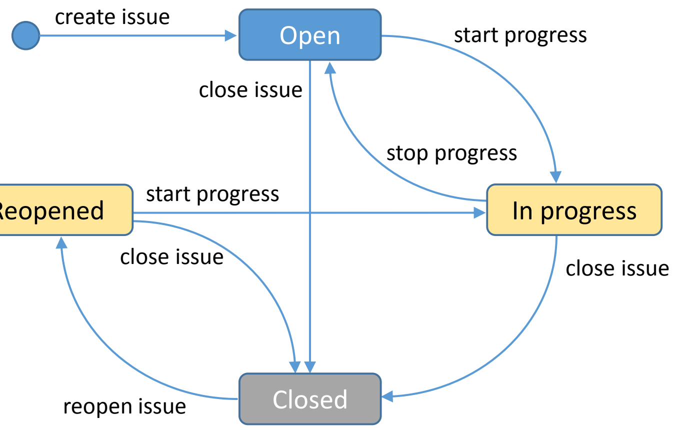

Figure 37: JIRA issue workflow

2.5.4 Linking GIT commits with JIRA issues

Each GIT commit can be linked to a JIRA issue. Unfortunately, it cannot trigger any JIRA issue state transition but at least you can see in the JIRA what files were modified to resolve the issue.

The link between GIT and JIRA is made with the commit comments starting with the JIRA issue number. You should use the following format when committing a change: *JIRA_project_key-Issue_number (Component) Comments*

- JIRA_project_key: Key that defines which project this commit applies to, in this case *LDPBFW*.
- Issue_number: Identifies which issue this commit applies to.
- Component: Identifies the JIRA project component it applies to, this is only for clarification so we can identify easier which part of the code it belongs to. You can find all JIRA components for the project on the JIRA webfront [5].
- Comments: Explains what is being done.

An example commit comment is shown in table 25, this links the commit to the JIRA issue '*LDPBFW–8*'. Ideally each commit should belong to a JIRA issue so it's easy to understand what is being done and why. For simple commits, you can skip the JIRA project key and issue number part.

```
LDPBFW-8 (doc) Updated documentation for JIRA and GIT
```

Table 25: GIT commit comment example to link to JIRA issue

3 Simulation and Compilation environment

3.1 Introduction

The repository is intended to be used for multiple projects related to the ATLAS LAr LDPB hardware. For example ABBA demonstrator and LATOME firmware are stored in this repository as a first step.

How the different projects are organised is explained in the sections 6. The environment itself is stored in the *env* directory, this is described in this chapter. All documents related to the environment are stored in the *doc* directory.

The environment is *Makefile* and *Python* based. Therefore you need to have a setup where this can be run, either *Linux* or Linux like (i.e. *Cygwin*) running on Windows machine. It is highly recommended to use a Linux machine to work with the environment, it will be much much faster than running from Windows.

3.2 Tools versions

You need to make sure you are using at least the following versions:

- *Make* $\geq v4.0$
- *Git* $\geq v2.1$
- *Python* $\geq v2.7$

Once the repository has been retrieved (see chapter 2.3.3), you should check that you have the correct versions of the tools installed running the *env/check_tools_versions* bash script:

```
user@machine ~
$ cd <repository_path>/atlas-lar-ldpb-firmware/env
user@machine /cygdrive/d/Users/user/Projects/ATLAS/git/atlas-lar-ldpb-
    firmware/env
$ ./check_tools_versions
PASSED: make v4.1 found (>= v4.0)
PASSED: git v2.2 found (>= v2.1)
PASSED: python v2.7 found (>= v2.7)
```

Table 26: Running *env/check_tools_versions*

3.2.1 Simulation/compilation flow

The simulation flow is shown on figure 38. The sources files are first pre-processed to identify the dependencies with other files, this result in a specific *Makefile* which includes all the dependencies. This *Makefile* is used to compile the source files using *Modelsim* or *Questa*. The sim_libraries and simulation targets are used in this process.

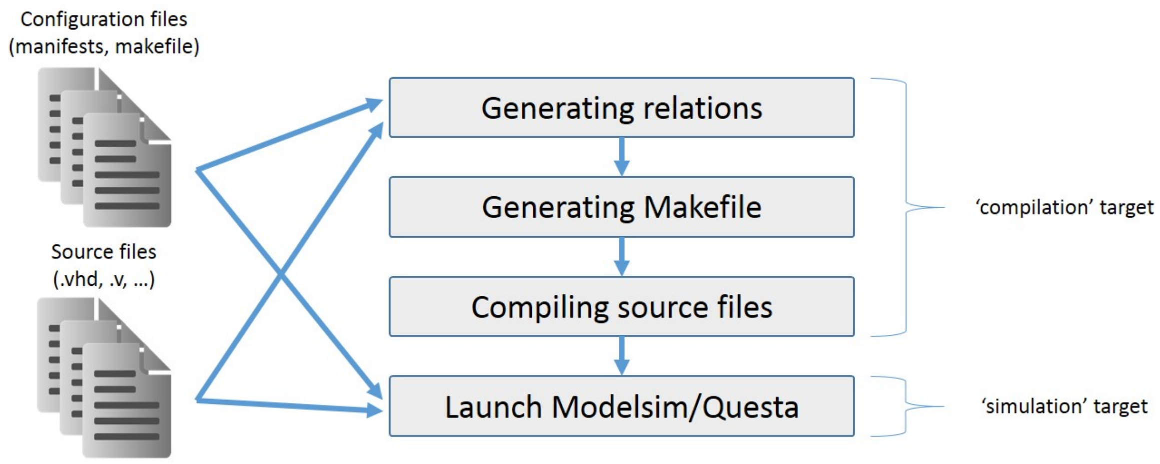

Figure 38: Simulation flow

The compilation flow is shown on figure 39. The *Manifest* files are used to create a *QuartusII* project. This project is used either to start the compilation process using the quartus_map, quartus_fit and quartus_asm targets or to launch the *QuartusII* GUI to update some of the IPs parameters or instanciates new ones using the quartus_gui target.

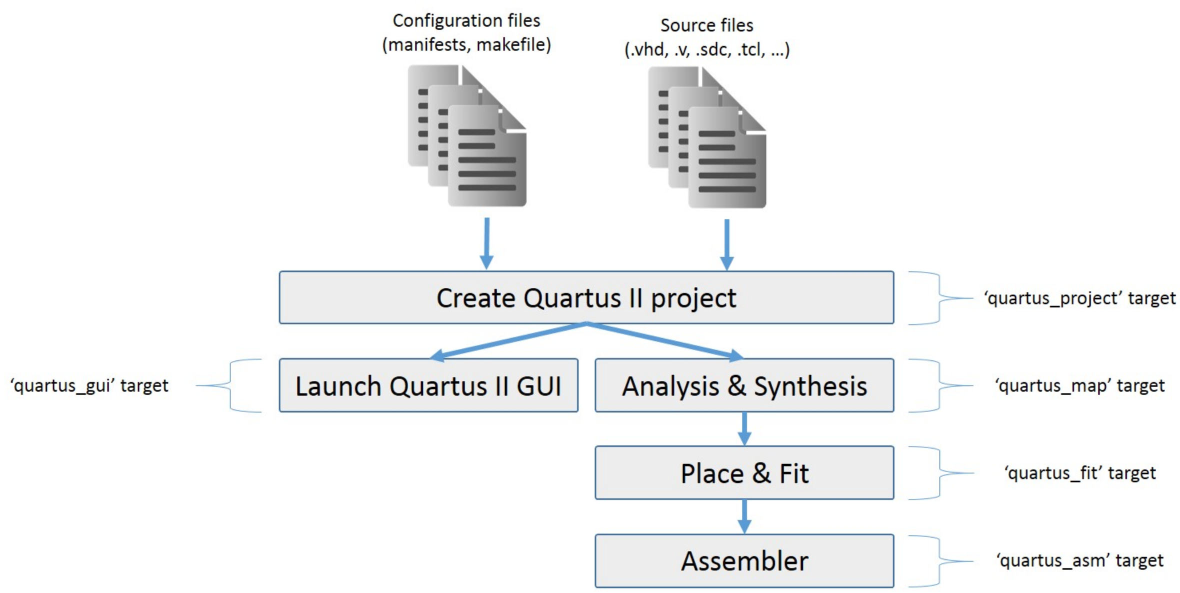

Figure 39: Compilation flow

3.2.2 Environment targets

The environment is Makefile based. You need to have at least *Make* $\geq v4.0$.

The main Makefile is located in *env/Makefile.env.mk*. Each top-level project *Makefile* should include this file. This Makefile defines a number of targets used for simulation and compilation as shown in table 27. It will include (if it exists) the file *Makefile.env.user.mk* as described in 3.2.3.

Target	Description
check_env	Check if the environment is correctly set for *Modelsim*/*Questa*/*Quartus* tools.
clean	Removes all temporary files.
doc	Generates doxygen documentation.
generate_altera_manifest	Generates Manifest.py for all Altera IPs in the current directory.
generate_modules_code_documentation	Generates code and documentation for all modules based on top_level.py.
sim_libraries	Compiles the design in order to simulate it.
help	Displays information about all available targets.
manifests_list	Displays the list of all manifests.
quartus_asm	Runs the Quartus II Assembler process on the current project.
quartus_fit	Runs the Quartus II Place&Fit process on the current project.
quartus_gui	Opens the current project in the Quartus II GUI.
quartus_map	Runs the Quartus II Analysis&Synthesis process on the current project.
quartus_project	Creates a Quartus II project.
quartus_sta	Runs the Quartus II TimeQuest Analyser process on the current project.
simulation	Simulates the design.
sources_list	Displays the list of all sources.
compilation	Compile the design. Uses the quartus_project, quartus_map, quartus_fit, quartus_asm targets.

Table 27: Environment targets

3.2.2.1 'check_env' target

Check if the environment is correctly set for *Modelsim*/*Questa*/*Quartus* tools.
See the following command :

```
$ make check_env
### Modelsim tool environment information ###
Variable HDLMAKE_MODELSIM_PATH set:
    "/cygdrive/d/MentorGraphics/questasim64_10.2c/win64"
Modelsim version 10.2c found under HDLMAKE_MODELSIM_PATH
Variable HDLMAKE_MODELSIM_10.2C_PATH set:
    "/cygdrive/d/MentorGraphics/questasim64_10.2c/win64"
Modelsim version 10.2c found under HDLMAKE_MODELSIM_10.2C_PATH
Detected Modelsim version 10.2c expecting version 10.2c

### Quartus tool environment information ###
Variable HDLMAKE_QUARTUS_PATH set:
    "/cygdrive/d/altera/13.1/quartus/bin64"
Quartus version 13.1 found under HDLMAKE_QUARTUS_PATH
Variable HDLMAKE_QUARTUS_13.1_PATH set:
    "/cygdrive/d/altera/13.1/quartus/bin64"
Quartus version 13.1 found under HDLMAKE_QUARTUS_13.1_PATH
Detected Quartus version 13.1 expecting version 13.1
```

Table 28: Example use of check_env target

3.2.2.2 'clean' target

Removes all temporary files.

```
$ make clean
rm -rf relations Makefile.hdlmake.relations.mk
rm -rf libraries preprocess Makefile.hdlmake.simulation.mk modelsim.
   ini
rm -f transcript.txt tcl_stacktrace.txt vsim.wlf
rm -f Makefile.hdlmake.compilation.mk
rm -f Makefile.hdlmake.manifests_files.mk
```

Table 29: Example use of clean target

3.2.2.3 'doc' target

Generates doxygen documentation. The VHDL/Verilog code is parsed and refman.pdf document is generated. In order to properly document your code, you should check the chapter 3.2.2.5.

```
$ make doc
[fpga_arch.vhd:211:Info: Elaborating entity ipctrl_top for hierarchy
    ipctrl_top::ipctrl]
[fpga_arch.vhd:218:Info: Elaborating entity istage_top for hierarchy
    istage_top::istage]
..
rogram Files (x86)/MiKTeX 2.9/fonts/type1/urw/helvetic/uhvro8a.pfb>
Output written on refman.pdf (102 pages, 1115448 bytes).
Transcript written on refman.log.
```

Table 30: Example use of doc target

3.2.2.4 'generate_altera_manifest' target

This target will search in the current directory for any Quartus II generated IPs and generate a *Manifest.py* including all necessary files for simulation and compilation. It will also generate an *altera_comp.vhd* file that you should use to instanciate the IPs in your code.

```
user@machine <path>/atlas-lar-ldpb-firmware/LATOME/src/altera/test/
    test_io_clocks
$ make generate_altera_manifest
Parsing file: ./fpll_125mhz_core/sim/mentor/msim_setup.tcl
Parsing file: ./fpll_156_25mhz_core/sim/mentor/msim_setup.tcl
Parsing file: ./fpll_160mhz_core/sim/mentor/msim_setup.tcl
Parsing file: ./in_syst_source_probe_32b/sim/mentor/msim_setup.tcl
Parsing file: ./iopll_100mhz_core/sim/mentor/msim_setup.tcl
Parsing file: ./iopll_160MHz_core/sim/mentor/msim_setup.tcl
```

Table 31: Example use of generate_altera_manifest target

3.2.2.5 'generate_modules_code_documentation' target

Generates code and documentation for all modules based on top_level.py. For each block is generated the following:

- *Makefile* file with correct relative path to the project.
- *Manifest.py* file with list of all files to simulate/compile the module.
- *module_if.vhd* file defining all module interfaces.
- *module_comp.vhd* file defining the module component to be used to instantiate it.
- *module_ent.vhd* file defining the module entity.
- *module_arch.vhd* file defining the module architecture. This is the only file that should be modified according to your module's design.
- *test/testbench* directory holding a testbench to simulate the module. Should be used as a base to develop testbenches.

```
user@machine <path>/atlas-lar-ldpb-firmware/LATOME/src
$ make generate_modules_code_documentation
[fpga]      Creating directory: generated/fpga
[fpga]      Generating manifest: generated/fpga/Manifest.py
[fpga]      Generating interface VHDL: generated/fpga/fpga_if.vhd
[ipctrl]        Creating directory: generated/ipctrl
[ipctrl]        Generating manifest: generated/ipctrl/Manifest.py
[ipctrl]        Generating interface VHDL: generated/ipctrl/ipctrl_if.
   vhd
```

Table 32: Example use of generate_modules_code_documentation target

3.2.2.6 'sim_libraries' target

Compiles the design in order to simulate it. All source files will be first preprocessed to determine what they provide or use (i.e. entity, package) and create relations files. Those relation files are used to create a Makefile with all dependencies. Then the *Modelsim*/*Questa* compiler is used to compile all the source files in the right order.

```
$ make sim_libraries
Generating manifests Makefile Makefile.hdlmake.manifests_files.mk
Generating relations Makefile Makefile.hdlmake.relations.mk
Generating relations manifest for ../../../../
   common_abba_vhdl_libraries/common_test_bench_lib/hdl/
   clock_generator_clk_generator_flow.vhd
Generating relations manifest for ../../../../
   common_abba_vhdl_libraries/common_mgwz_generated/hdl/fifo_sc_64bx8.
   vhd
...
Generating simulation Makefile Makefile.hdlmake.simulation.mk
Creating library: libraries/common_general_components
Creating library: libraries/common_ipbus_lib
...
Compiling Verilog file: ../../../../common_abba_vhdl_libraries/
   common_nios_lib/hdl/nios_abba_ipmc_full.v
Compiling Verilog file: ../../../../../../../../../../../../../../d/
   altera/13.1/quartus/eda/sim_lib/mentor/altera_lnsim_for_vhdl.sv
Compiling VHDL file: ../../../../common_abba_vhdl_libraries/
   common_test_bench_lib/hdl/clock_generator_clk_generator_flow.vhd
Compiling VHDL file: ../../../../../../../../../../../../../../d/altera
   /13.1/quartus/eda/sim_lib/altera_mf_components.vhd
...
```

Table 33: Example use of sim_libraries target

3.2.2.7 'help' target

Displays information about all available targets.

```
$ make help
Available targets:

check_env:        Check if the environment is correctly set for
   Modelsim/Questa/Quartus tools.
clean:            Removes all temporary files.
sim_libraries:    Compiles the design in order to simulate it.
help:             Displays information about all available targets.
manifests_list:   Displays the list of all manifests.
quartus_asm:      Runs the Quartus II Assembler process on the current
   project.
quartus_fit:      Runs the Quartus II Place&Fit process on the current
   project.
quartus_gui:      Opens the current project in the Quartus II GUI.
quartus_map:      Runs the Quartus II Analysis&Synthesis process on the
    current project.
quartus_project: Creates a Quartus II project.
quartus_sta:      Runs the Quartus II TimeQuest Analyzer process on the
    current project.
simulation:       Simulates the design.
sources_list:     Displays the list of all sources.
compilation:      Compile the design. Uses the quartus_project/
   quartus_map/quartus_fit/quartus_asm/quartus_sta targets.
```

Table 34: Example use of help target

3.2.2.8 'manifests_list' target

Displays the list of all *Manifest.py* files defined in the current sub-project. All the files are printed on the same line.

```
$ make manifests_list | tr ' ' '\n'
Manifest.py
../../../../altera/Manifest.py
../../Manifest.py
../../../user_emf_front_fpga_block_adc_readout_lib/Manifest.py
...
```

Table 35: Example use of manifests_list target

3.2.2.9 'projects' target

Simulate/compile all the projects defined in the projects manifest variable. Each project will be called recursively and a log file generated with PASS/FAIL results.

```
TBD
```

Table 36: Example use of projects target

439 #### 3.2.2.10 'quartus_asm' target

440 Runs the Quartus II Assembler process on the current project.

```
$ make quartus_asm
Generating manifests Makefile Makefile.hdlmake.manifests_files.mk
Quartus II Assemble
```

Table 37: Example use of quartus_asm target

441 #### 3.2.2.11 'quartus_fit' target

442 Runs the Quartus II Place&Fit process on the current project.

```
$ make quartus_fit
Generating manifests Makefile Makefile.hdlmake.manifests_files.mk
Quartus II Place and Route
Critical Warning (169085): No exact pin location assignment(s) for 11
    pins of 285 total pins
Please check the report file: user_emf_front_fpga1.fit.rpt
```

Table 38: Example use of quartus_fit target

443 #### 3.2.2.12 'quartus_gui' target

444 Opens the current project in the Quartus II GUI.

```
$ make quartus_gui
Generating manifests Makefile Makefile.hdlmake.manifests_files.mk
/cygdrive/d/altera/13.1/quartus/bin64/quartus user_emf_front_fpga1.qpf
```

Table 39: Example use of quartus_gui target

445 #### 3.2.2.13 'quartus_map' target

446 Runs the Quartus II Analysis&Synthesis process on the current project.

```
$ make quartus_map
Generating manifests Makefile Makefile.hdlmake.manifests_files.mk
Quartus II Analysis and Synthesis
```

Table 40: Example use of quartus_map target

3.2.2.14 'quartus_project' target

Creates a Quartus II project. All *Manifest* files are parsed to identify the source files to use to create the project. The *Quartus* STP (SignalTap) tool is also started in order to compile each .stp files before starting the compilation process.

```
$ make quartus_project
Generating manifests Makefile Makefile.hdlmake.manifests_files.mk
Generating compilation Makefile Makefile.hdlmake.compilation.mk
Creating Quartus II project
    preparing project with device ep4sgx290nf45c2
    making assignments
    running SignalTap
```

Table 41: Example use of quartus_project target

3.2.2.15 'quartus_sta' target

Runs the Quartus II Timequest Analyzer process on the current project.

```
$ make quartus_sta
Generating manifests Makefile Makefile.hdlmake.manifests_files.mk
Quartus II Timequest Analyzer
Critical Warning (332148): Timing requirements not met
Critical Warning (332148): Timing requirements not met
Critical Warning (332148): Timing requirements not met
Please check the report file: user_emf_front_fpga1.sta.rpt
```

Table 42: Example use of quartus_sta target

3.2.2.16 'simulation' target

Simulates the design. If the sources files are not compiled, the sim_libraries target will be issued first. *Modelsim/Questa* is then started.

```
$ make simulation
Generating manifests Makefile Makefile.hdlmake.manifests_files.mk
make[1]: 'relations' is up to date.
make[2]: Nothing to be done for 'sim_libraries'.
make[2]: Nothing to be done for 'sim_libraries'.
make[1]: Entering directory '/cygdrive/c/Users/chevillot/Projects/
    ATLAS/git/atlas-lar-ldpb-firmware/ABBA/ABBA_PH0/
    user_abba_daq_v00_vhdl_libraries/user_emf_front_fpga_lib/simulation
    /user_emf_front_fpga1'
/cygdrive/d/MentorGraphics/questasim64_10.2c/win64/vsim -L
    common_general_components -L common_ipbus_lib -L common_abba_lib -L
     user_emf_front_fpga_block_adc_readout_lib -L altera_lnsim -L sgate
     -L user_emf_front_fpga_block_optics_interface_lib -L altera_mf -L
    lpm -L altera -L user_emf_front_fpga_block_back_fpga_interface_lib
    -L common_mgwz_generated -L common_test_bench_lib -L
    user_emf_front_fpga_lib -L stratixiv -L common_ttc_lib -L
    common_nios_lib -L stratixiv_hssi -L common_10gbe_lib -l transcript
    .txt -i -multisource_delay latest -t ps +typdelays "
    user_emf_front_fpga_lib.user_emf_front_fpga1_tb(struct)"
Reading D:/MentorGraphics/questasim64_10.2c/tcl/vsim/pref.tcl
```

Table 43: Example use of simulation target

3.2.2.17 'sources_list' target

Displays the list of all source files defined in the current sub-project. All the files are printed on the same line.

```
$ make sources_list | tr ' ' '\n'
../../hdl/back_fpga_interface_tester_struct.vhd
../../hdl/user_emf_front_fpga1_tb_struct.vhd
../../hdl/user_emf_front_fpga1_struct.vhd
...
```

Table 44: Example use of sources_list target

3.2.2.18 'compilation' target

Compile the design. Uses the quartus_project, quartus_map, quartus_fit, quartus_asm targets.

3.2.3 Environment variables

In order for the environment to work, you need to setup the path to the appropriate tools to be used, i.e. *Modelsim/Questa* and *QuartusII*.

Table 45 shows the used environment variables.

Environment variable	Description
HDLMAKE_MODELSIM_PATH	Defines the path to the *Modelsim/Questa* binaries. Used when the project does not define a specific version of the tool to be used, refer to 3.3.2.7
HDLMAKE_MODELSIM_x.y_PATH	Defines the path to a specific version of the *Modelsim/Questa* binaries. Used when the project defines a specific version of the tool to be used, refer to 3.3.2.7
HDLMAKE_QUARTUS_PATH	Defines the path to the *QuartusII* binaries. Used when the project does not define a specific version of the tool to be used, refer to 3.3.2.10
HDLMAKE_QUARTUS_x.y_PATH	Defines the path to a specific version of the *QuartusII* binaries. Used when the project defines a specific version of the tool to be used, refer to 3.3.2.10
PROJECT_ROOT_PATH	Defines the path to the top directory of your current project. This should be defined in the *Makefile* for the project.

Table 45: Environment variables description

The environment variables are defined either:

- directly in your working environment
- in the *env/Makefile.env.user.mk* file

The *env/Makefile.env.user.mk* file defines where the tools are located, you can specify a generic version of the tool you want to use or a dedicated version of it. The example file shows how to define different paths whether you are running *Cygwin* or *Linux* based machine and how to distinguish a specific computer.

You can use the example file *env/Makefile.env.user.mk.example*, rename it to *env/Makefile.env.user.mk* and modify it to fit your local settings.

3.3 Project description

Each project can contain a number of sub-projects.
Sub-projects are either a test-bench used for simulation or meant to compile the design.

In order to be able to simulate or compile the design, the environment *Makefile.env.mk* should be included.
This is done using a local *Makefile*, this is described in chapter 3.3.1.

The description of what is intended is made through a *Manifest.py* file, this is described in chapter 3.3.2.

3.3.1 Project/Module Makefile

Each sub-project should have a *Makefile* including the environment *env/Makefile.env.mk*.

The *Makefile* should define the *PROJECT_ROOT_PATH* environment variable. Please refer to chapter 3.2.3 for details.

An example *Makefile* is given in table 46. You can copy/paste this example and update the *PROJECT_ROOT_PATH* and *ENVIRONMENT_PATH* to match your module location.

```
# Path to the project directory
export PROJECT_ROOT_PATH = $(realpath ../../../..)

# Path to environment directory at the top of the repository
export ENVIRONMENT_PATH = ${PROJECT_ROOT_PATH}/../../env

# include the main makefile
include ${ENVIRONMENT_PATH}/Makefile.env.mk
```

Table 46: Example *Makefile*

3.3.2 Project/Module manifest

Each project or module is described using a *Python* file named *Manifest.py*. This file contains several variables (this should be compatible with *Python*).

Each possible variable is described below in the next chapters. Table 47 shows all available variables that can be used in a *Manifest*.

Tables 48, 49 and 50 show examples of *Manifest.py* files.

Variable	Description
action	Defines which action is to be undertaken for the project.
files	Defines which sources files to use for the current module.
library	Defines which library by default is used for the module.
modules	Defines a list of paths where local modules (i.e. *Manifest.py*) are to be used in the project/sub-module.
projects	Defines a list of projects to be simulated/compiled along the current project.
sim_tool	Defines which tool to be used for simulation.
sim_tool_version	Defines which *Modelsim/Questa* tool version should be used to simulation the design.
comp_device	Defines which device is used for compilation.
comp_tool	Defines which tool to be used for compilation.
comp_tool_version	Defines which *Quartus* tool version should be used to compile the design.
sim_do_cmd	Define a file to be executed at simulation startup.
top_module	Defines the top level entity for simulation and compilation.
vsim_opt	Additional options for modelsim simulator.

Table 47: Manifest variables

```
# Simulating the design
action = "simulation"

# Simulation tool is Modelsim v10.2c
sim_tool = "modelsim"
sim_tool_version = "10.2c"
# Compilation tool is Quartus v13.1, used in simulation for Altera
    primitives
comp_tool = "quartus"
comp_tool_version = "13.1"

# Top module used for simulation
top_module = "my_working_lib.my_testbench_entity(struct)"

# List of modules
modules = {
        "local" : [
                "$PROJECT_ROOT_PATH/Module_1",
                "$PROJECT_ROOT_PATH/Module_2",
        ],
}

# Default library
library = "my_working_lib"

# List of source files for the testbench
files = [
        "Testbench\_1.vhd",
]
```

Table 48: Example *Manifest.py* used for simulation

```
# Default library
library = "my_module_1_lib"

# List of source files for the module
files = [
        "Module_1_block_1.vhd",
        "Module_1_block_2.vhd",
]
```

Table 49: Example *Manifest.py* used for a module

```
# Synthesizing the design
action = "compilation"

# Compilation tool is Quartus v13.1
comp_tool = "quartus"
comp_tool_version = "13.1"

# Targeted device
comp_device = "ep4sgx290nf45c2"

# Top module used for compilation
top_module = "my_top_entity"

# List of modules
modules = {
        "local" : [
                "$PROJECT_ROOT_PATH/Module_1",
                "$PROJECT_ROOT_PATH/Module_2",
        ],
}

# List of source files for compilation
files = [
        "tcl_script.tcl",
        "synopsis_design_constraint_file.sdc",
        "signal_tap_file.stp",
]
```

Table 50: Example *Manifest.py* used for compilation

3.3.2.1 Manifest variable 'action'

Defines which action is to be undertaken for the project. This option is valid only for the top-module.

Possible value	Description
simulation	The project describes all necessary elements to simulate the design, i.e. a test-bench
compilation	The project describes all necessary elements to compile the design

Table 51: Manifest 'action' variable description

```
action = "simulation"
```

Table 52: Example use of 'action' variable

3.3.2.2 Manifest variable 'files'

Defines which sources files to use for the current module.

The supported file types are:

File extension	Description
.vhd .vhdl .vho	VHDL source file
.v .vh .vo .vm	Verilog source file
.sv .svh	System-Verilog source file
.tcl	TCL script file
.stp	Quartus II SignalTap file
.sdc	Quartus II TimeQuest Synopsis Design Constraint file
.qip	Quartus II IP file

Table 53: Manifest 'files' variable supported file types

```
files = [
        "my\_file\_1.vhd",
        "my\_file\_2.v",
],
```

Table 54: Example use of 'files' variable

Each file can be associated some attributes. Valid attributes are:

Attribute	Description
action	Use this file only for the specified action. For example can be used to include one file only in case of simulation, i.e. test-bench
dependencies	Specify a list of files on which this file depends.
library	Used to override the default library defined for the module.
preprocess	Used to pre-process the file before using it for simulation or compilation. If set to *envsubst*, all environment variables contained in the file will be replaced by their value. This is useful for memory initialization where the path should be absolute.

Table 55: Manifest 'files' variable description

```
files = [
        "my_file_1.vhd",
        {"my_file_tb_1.v": {'action': "simulation"}},
],
```

Table 56: Example use of 'files' variable with attributes

3.3.2.3 Manifest variable 'library'

Defines which library by default is used for the module. All files defined in the module will use this library unless otherwise specified.

```
library = "my_working_lib"
```

Table 57: Example use of 'library' variable

3.3.2.4 Manifest variable 'modules'

Defines a list of paths where local modules (i.e. *Manifest.py*) are to be used in the project/sub-module.

For the moment only local paths can be used as source (the tool could allow for remote *GIT/SVN* repositories).

```
modules = {
        "local" : [
                "$PROJECT_ROOT_PATH/module1",
                "$PROJECT_ROOT_PATH/modules/module2",
        ],
}
```

Table 58: Example use of 'modules' variable

3.3.2.5 Manifest variable 'projects'

Defines a list of projects to be simulated/compiled along the current project.

```
# List of projects
projects = [
    "$PROJECT_ROOT_PATH/src/fpga",
    ...
    "$PROJECT_ROOT_PATH/src/user",
]
```

Table 59: Example use of 'projects' variable

3.3.2.6 Manifest variable 'sim_tool'

Defines which tool to be used for simulation. For the moment only *Modelsim/Questa* is supported.

Possible value	Simulation tool used
modelsim	Modelsim/Questa

Table 60: Manifest 'sim_tool' variable description

```
sim_tool = "modelsim"
```

Table 61: Example use of 'sim_tool' variable

3.3.2.7 Manifest variable 'sim_tool_version'

Defines which *Modelsim/Questa* tool version should be used to simulation the design.

If not specified, the tool will be searched in $HDLMAKE\_MODELSIM\_PATH$ environment variable.

If specified, the tool will be searched in the $HDLMAKE\_MODELSIM_{\_version\_number\_}PATH$ environment variable. If not found then $HDLMAKE\_MODELSIM\_PATH$ will be used.

Possible value	Environment variable used
not defined	HDLMAKE_MODELSIM_PATH
x.y	HDLMAKE_MODELSIM_x.y_PATH
13.1	HDLMAKE_MODELSIM_13.1_PATH
14.1	HDLMAKE_MODELSIM_14.1_PATH

Table 62: Manifest 'sim_tool_version' variable description

```
sim_tool_version = "10.2c"
```

Table 63: Example use of 'sim_tool_version' variable

3.3.2.8 Manifest variable 'comp_device'

Defines which device is used for compilation.

```
comp_device = "ep4sgx290nf45c2"
```

Table 64: Example use of 'comp_device' variable

3.3.2.9 Manifest variable 'comp_tool'

Defines which tool to be used for compilation. For the moment only *QuartusII* is supported.

Possible value	Compilation tool used
quartus	Quartus II

Table 65: Manifest 'comp_tool' variable description

```
comp_tool = "quartus"
```

Table 66: Example use of 'comp_tool' variable

3.3.2.10 Manifest variable 'comp_tool_version'

Defines which *Quartus* tool version should be used to compile the design.

If not specified, the tool will be searched in $HDLMAKE\_QUARTUS\_PATH$ environment variable.

If specified, the tool will be searched in the $HDLMAKE\_QUARTUS_{\_version\_number}\_PATH$ environment variable. If not found then $HDLMAKE\_QUARTUS\_PATH$ will be used.

Possible value	Environment variable used
not defined	HDLMAKE_QUARTUS_PATH
x.y	HDLMAKE_QUARTUS_x.y_PATH
13.1	HDLMAKE_QUARTUS_13.1_PATH
14.1	HDLMAKE_QUARTUS_14.1_PATH

Table 67: Manifest 'comp_tool_version' variable description

```
comp_tool_version = "14.1"
```

Table 68: Example use of 'comp_tool_version' variable

3.3.2.11 Manifest variable 'sim_do_cmd'

Define a file to be executed at simulation startup.

```
# Waveforms for simulation
sim_do_cmd = "wave.do"
```

Table 69: Example use of 'sim_do_cmd' variable

3.3.2.12 Manifest variable 'top_module'

Defines the top level entity for simulation and compilation.

```
# Top module used for simulation/compilation
top_module = "my_library.my_test_bench"
```

Table 70: Example use of 'top_module' variable

3.3.2.13 Manifest variable 'vsim_opt'

Additional options for modelsim simulator.

```
# Additional options for modelsim
vsim_opt = "-l transcript.txt -i -multisource_delay latest -t ps +
   typdelays"
```

Table 71: Example use of 'vsim_opt' variable

4 Code documentation

4.1 Introduction

The doc target is used to generate code documentation based on Doxygen. Doxygen is a commonly used tool that allows with simple comments in the code to automatically generate readable PDF documents. You can refer to the Doxygen manual: http://www.stack.nl/ dimitri/doxygen/manual/index.html.

Doxygen comments in VHDL start with the following: **- - !**

Examples of how to comment the code are given below.

4.2 Commenting VHDL file header

Each file should have a simple header describing what the file is about. The following keywords should be used:

- @brief Brief description about the file.
- @details More detailed description about the file.

```
---------------------------------------------------------
--! @file my_module.vhd
--! @brief Short description about what this file is about
--! @details More description about what this file is about
---------------------------------------------------------
```

Table 72: VHDL doxygen comment for a file

4.3 Commenting VHDL library use

The doxygen comment should be placed before the 'library'/'use' keywords.

```
--! Using IEEE library for standard logic
library IEEE;
--! Standard logic
use IEEE.STD_LOGIC_1164.all;
```

Table 73: VHDL doxygen comment for a library

4.4 Commenting VHDL entity

The doxygen comment should be placed before the entity. The following keywords should be used:

- @brief Brief description about the entity, i.e. a title.
- @details More detailed description about the entity and what it is for.

```
--! Entity definition for whatever module
entity my_entity_top is
    port(
        ...
    );
end my_entity_top;
```

Table 74: VHDL doxygen comment for an entity

4.5 Commenting VHDL entity port

The doxygen comment should be placed on the same line as the port, at the end of the line.

```
entity my_entity_top is
    port(
        my_port: in std_logic; --! description of what my port is
            about
        ...
    );
end my_entity_top;
```

Table 75: VHDL doxygen comment for an entity port

4.6 Commenting VHDL architecture

The doxygen comment should be placed before the architecture. The following keywords should be used:

- @brief Brief description of the architecture, i.e. a title.
- @details More detailed description of the architecture and what it is for.

```
--! @brief Architecture definition for whatever module
--! @details More details about this architecture.
architecture struct of my_entity_top is
    ...
begin
    ...
end struct;
```

Table 76: VHDL doxygen comment for an architecture

4.7 Commenting VHDL package

The doxygen comment should be placed before the package. The following keywords should be used:

- @brief Brief description about the package, i.e. a title.
- @details More detailed description about the package and what it is for.

```
--! @brief Short description about the package
--! @details More details about the package.
package lli_if is
   ...
end;
```

Table 77: VHDL doxygen comment for a package

4.8 Commenting VHDL constant

The doxygen comment should be placed on the same line as the constant, at the end of the line.

```
constant MY_BUS_DATA_WIDTH: integer := 32; --! Bus width for my bus
```

Table 78: VHDL doxygen comment for a constant

4.9 Commenting VHDL array

The doxygen comment should be placed on the same line as the constant, at the end of the line.

```
type my_array_t is array(MY_ARRAY_NB-1 downto 0) of whatever_type_t;
   --! Description of what the array type is about
```

Table 79: VHDL doxygen comment for an array

4.10 Commenting VHDL record

The doxygen comment should be placed between the ’type ¡type_name¿ is’ and ’record’. The following keywords should be used:

- @brief Brief description about the record, i.e. a title.
- @details More detailed description about the record and what it is for.
- @param Description of record entry. Should have one ’param’ line per record entry.

```
type my_record_t is
--! @brief Short description of what the record is about
--! @details More detailed description of what the record is about
--! @param entry_1 Description of the first entry in the record
...
--! @param entry_n Description of the nth entry in the record
record
    entry_1: std_logic_vector(ENTRY_1_DATA_WIDTH-1 downto 0);
    ...
    xcvr_rx_320_clk: std_logic;
end record;
```

Table 80: VHDL doxygen comment for a record

5 Use cases

5.1 Introduction

This section describe a number of common use cases, they should help you to get acquainted to the environment.

5.2 Create a project or module

When you want to create a new project or module, you need to create a directory to hold all necessary files. In this directory you should create a *Makefile* and *Manifest.py* to describe your project/module. The content of those files is described in 3.3.1 and 3.3.2.

The next step is to add your source files into the directory you just created and add them in the *Manifest.py*.

5.3 Add or remove a source file from the module's list

If you want to add or remove a source file from the module's list, you need to edit the module's *Manifest.py* file. This file is described in the section 3.3.2. You should edit the *files* section of this file. Once this file is updated you can simulate/compile your design using the standard commands. The dependencies are automatically updated.

5.4 Create a new testbench

If you want to add a new testbench in your module, you should create a new directory. This directory should hold at least a *Makefile* and *Manifest.py* file. The best is to copy from an existing testbench.

The first thing to do is to make sure the *Makefile*'s *PROJECT_ROOT_PATH* variable points to the correct project's location. This is described in the section 3.3.1.

Then you need to update/create the *Manifest.py* file to describe your testbench, this is described in section 3.3.2.

How you organise your testbench is not described in this document.

5.5 Use a Linux remote machine

If you are using Cygwin on a Windows machine, it might be useful to work remotely on a linux machine instead. Indeed simulation/compilation under Windows is not as efficient as on Linux.

If you still want to edit your source code from the Cygwin environment, you will need to have the GIT repository stored in a location that can be accessed from both environments. You can also work completely remotely and do not need the Cygwin environment.

If you want to work from Cygwin remotely on a Linux machine, you will need to start the X Window system. This is shown in table 81. If you don't want to have graphical interface, you can skip the X Window part.

```
$ export DISPLAY=:1.0
$ startxwin &
$ ssh -Y lappc-f561
Last login: Fri Apr 24 14:56:20 2015 from lappc-f460.in2p3.fr
-bash-4.1$
```

Table 81: Example use of X Window to work remotely on a Linux machine

5.6 Altera IPs

Each block using Altera IPs should create locally to the module an *altera* directory. Quartus II should be used to create or parametrize each IP used. All the files generated by Quartus II should be stored in the *altera* directory.

Once all the files are stored in the *altera* directory, the generate_altera_manifest target should be used to generate the associated *Manifest.py* file containing all necessary files for simulation and compilation.

Following sections describe how to create Altera IPs from scratch (section 5.6.1), and update existing ones (section 5.6.2).

5.6.1 Generating Altera IPs from scratch

- Create a directory named *altera*.
- Copy *LATOME/Makefile* to the *altera* directory.
- Edit the *Makefile* to reflect where this file is located relative to the LATOME directory, i.e. update the *PROJECT_ROOT_PATH* variable.
- Use the generate_altera_manifest to generate an empty *Manifest.py*.
- Use the quartus_gui target to start Quartus II. This will open the Quartus II software.
- Open the IP Catalog: *tools* → *IP Catalog*.
- Select which IP you want to add and click on the *Add* button. The *IP Parameter Editor* is started.
- Enter the *Entity name* you want to give your new IP and make sure the path where the IP will be stored is correct as shown on figure 40. Click on *OK*.

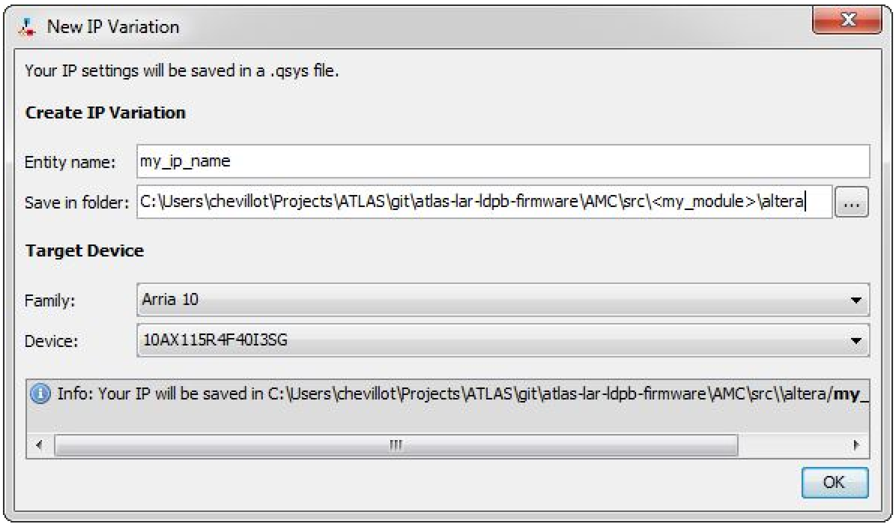

Figure 40: Quartus II new IP variation dialog

- Configure the IP as you wish.
- Click on *Generate* once done in order to generate the HDL code used for simulation and compilation. This will open the *Generation dialog*, make sure the correct options are used as shown on figure 41. Click on *Generate* and *Close* once generated.
- Click on *Finish* when done.

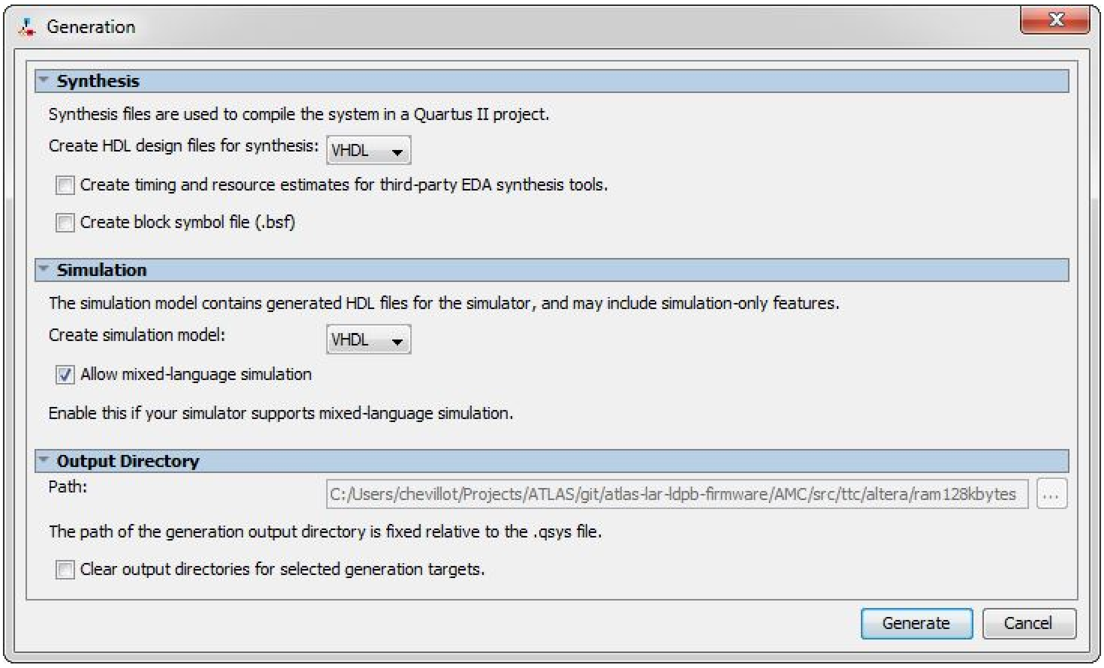

Figure 41: Quartus II generation dialog

- Agree to add the newly created IP to the current project as show on figure 42.

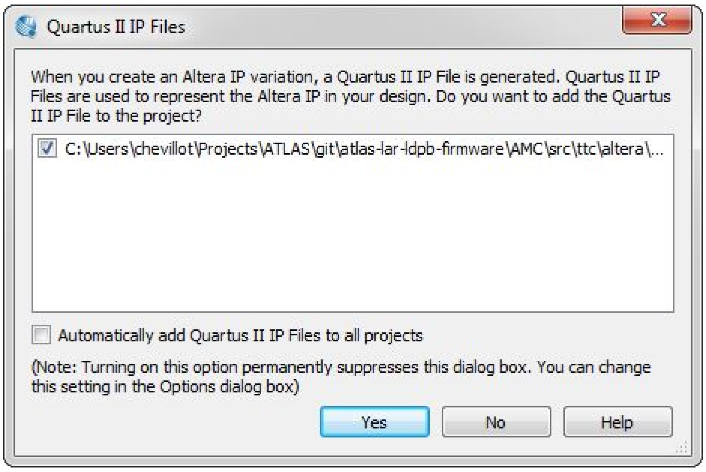

Figure 42: Quartus II add IP to current project

- Add more IPs as you wish.
- Once all IPs are created, you can quit Quartus II.
- Use the generate_altera_manifest target to update the *Manifest.py* file according to the IPs you have created. An example is shown in table 82. The *Manifest.py* file is updated including all necessary files for simulation and compilation of your IPs.

```
$ make generate_altera_manifest
Parsing file: ./ram128kbytes/sim/mentor/msim_setup.tcl
Generating Manifest.py
Generating altera_comp.vhd
```

Table 82: Example use of generate_altera_manifest target after adding new IP in Quartus II

- You can clean all temporary files using the clean target.
- Add the following files into the GIT repository (do not add the Quartus II project files!):
  - *Makefile*
  - *Manifest.py*
  - *altera_comp.vhd*
  - *your_ip_name.qsys*
  - *your_ip_name* (directory)

5.6.2 Updating existing Altera IPs

The procedure is very similar to the one for creating a new IP.

- Go to the *altera* directory of your block.
- Use the quartus_gui target to start Quartus II. This will open the Quartus II software.

- In the *Project Navigator*, click on the *IP components* tab.
- Double-click on the IP you want to reconfigure.
- Click on *Generate* once done in order to generate the HDL code used for simulation and compilation. This will open the *Generation dialog*, make sure the correct options are used as shown on figure 41. Click on *Generate* and *Close* once generated.
- Click on *Finish* when done.
- Reconfigure/add more IPs as you wish.
- Once done, you can quit Quartus II.
- Use the generate_altera_manifest target to update the *Manifest.py* file according to the IPs you have reconfigured/created. The *Manifest.py* file is updated including all necessary files for simulation and compilation of your IPs.
- You can clean all temporary files using the clean target.
- Make sure to add additional files for the IPs you have reconfigured:
  - *your_ip_name.qsys*
  - *your_ip_name* (directory)
- If you have added new IPs, do not forget to add the necessary files into the GIT as described in the previous section.

5.6.3 Use Altera IPs in your code

As described in section 3.2.2.4, the generate_altera_manifest target has created an *altera_comp.vhd* file that should be used to instantiate your IPs in your code. Each IPs is compiled in the *altera_comp* library which means you need to use this library in your code and also compile it. Table 83 shows an example of VHDL code. You will need to add the altera module in your module *Manifest.py*, you can check the following section 3.3.2.4.

```
--! Using 'altera_comp' library for Altera IP
library altera_comp;
--! Using 'altera_comp' module library for component
use altera_comp.altera_comp.all;

--! @brief Architecture definition for the 'my_module' module
--! @details This is a test module
architecture my_module_top_struct of my_module_top is

begin

    --! Instantiation of my_ip module
    my_ram_1: component ram128kbytes
    port map(
        signal1 => signal1,
        ...
        signaln => signaln
    );

end architecture my_module_top_struct;
```

Table 83: Example VHDL code using an Altera IP

6 LATOME Project

6.1 Introduction

The firmware for the LATOME project is using the environment defined in this document. The source code for the firmware is stored in the LATOME directory.

The LATOME project is based on the Altera Arria-10 '$10AX115R4F40I3SG$' FPGA device. This device should be used in order to create the proper IPs in Quartus II. Quartus II version 14.1.1.190 is used.

6.2 High-Level modules

The firmware is organized in high-level modules which are defined in the *LATOME/top_level.py* file, they are shown in table 84. This file is used by the generate_modules_code_documentation target to generate the code skeleton and interfaces documentation used in the LAr-LATOME-FW firmware documentation [8].

Module name	Description
ipctrl	IP Bus controller
istage	Input stage
lli	Low-level interface
mon	TDAQ Monitoring
osum	Output summing
remap	Configurable remapping
ttc	TTC module
user	User code

Table 84: LATOME High-level modules description

6.3 High-Level interfaces

All interfaces between each high-level modules are defined in the *LATOME/top_level.py* file, they are shown on table 85.

<table>
<tr><th>Interface</th><th>#</th><th colspan="2">Description</th></tr>
<tr><td colspan="4">Clocks</td></tr>
<tr><td>ipctrl_100_clk</td><td>1</td><td colspan="2">IP Bus controller 100MHz clock</td></tr>
<tr><td>ttc_240_clk</td><td>1</td><td colspan="2">TTC 240MHz recovered clock</td></tr>
<tr><td>ttc_320_clk</td><td>1</td><td colspan="2">TTC 320MHz recovered clock</td></tr>
<tr><td colspan="4">Data path</td></tr>
<tr><td rowspan="6">istage_remap_sc_data_aligned_st</td><td>48</td><td colspan="2">Supercell ADC data aligned on the TTC 320MHz clock</td></tr>
<tr><td>Signal/Bus</td><td>Width</td><td>Description</td></tr>
<tr><td>bcid_0</td><td>1</td><td>Identifies BCID 0 occurring every 3564 cycles.</td></tr>
<tr><td>data</td><td>12</td><td>Supercell ADC data</td></tr>
<tr><td>error</td><td>2</td><td>Report CRC errors or BCID errors on the corresponding channels</td></tr>
<tr><td>startofpacket</td><td>1</td><td>Indicates the first word in the series of 8 words of each BCID packet</td></tr>
</table>

Table 85: LATOME High-level interfaces description (part)

Interface	#	Description	
	valid	1	Indicates that the data going out of the input stage can be used by the configurable remapping. This signal is activated when all the selected channels are synchronized and re-timed without errors.
lli_istage_ltdb_data_st	48	Incoming LTDB data	
	Signal/Bus	**Width**	**Description**
	data	16	Incoming supercell data from the deserializer
	rx_bitslip	1	Signal sent to the Rx part of the input transceiver to shift the data by one bit. Used until the frame is properly aligned.
	valid	1	Transceiver locked
	xcvr_rx_320_clk	1	320 MHz recovered clock from the transceiver
remap_user_remap_data_st	32	Reordered ADC data aligned on the TTC 240MHz clock	
	Signal/Bus	**Width**	**Description**
	data	24	ADC data of two supercells
	error	4	CRC error, BCID error for two supercells
	startofpacket	1	First word of the packet
	valid	1	Valid data to user code
user_osum_out_data_st	32	Processed data block from user code	
	Signal/Bus	**Width**	**Description**
	data	28	Two 14 bits data words from user code packed in a 28 bits word
	error	4	CRC error, BCID error for two supercells
	quality	8	Quality information
	startofpacket	1	First word of the packet
	valid	1	Valid data from user code
FEX data			
osum_lli_fex_data_st	48	Packed FEX data	
	Signal/Bus	**Width**	**Description**
	data	32	FEX data
	valid	1	FEX data is valid
	xcvr_tx_280_clk	1	Transceiver clock
GBT Link			
lli_mon_gbt_link_c	3	GBT Monitoring link	
gbt_120_clk	1	Transceiver clock	
rx_data_st	1	RX data bus	
	Signal/Bus	**Width**	**Description**
	data	16	TBD
	ready	1	TBD
	valid	1	TBD
tx_data_st	1	TX data bus	
	Signal/Bus	**Width**	**Description**
	data	16	TBD
	ready	1	TBD

Table 85: LATOME High-level interfaces description (part)

Interface	#	Description	
	valid	1	TBD
Gigabit Ethernet Link			
lli_ipctrl_gbe_link_c	1	Gigabit Ethernet link	
gbe_100_clk	1	Transceiver clock	
rx_data_st	1	RX data bus	
	Signal/Bus	**Width**	**Description**
	data	32	TBD
	empty	2	TBD
	endofpacket	1	TBD
	error	6	TBD
	ready	1	TBD
	startofpacket	1	TBD
	valid	1	TBD
tx_data_st	1	TX data bus	
	Signal/Bus	**Width**	**Description**
	data	32	TBD
	empty	2	TBD
	endofpacket	1	TBD
	error	1	TBD
	ready	1	TBD
	startofpacket	1	TBD
	valid	1	TBD
LVDS Link			
lli_ttc_lvds_link_c	4	LVDS link	
lvds_160_clk	1	Transceiver clock	
rx_data_st	1	RX data bus	
	Signal/Bus	**Width**	**Description**
	data	16	TBD
	valid	1	TBD
tx_data_st	1	TX data bus	
	Signal/Bus	**Width**	**Description**
	data	16	TBD
	valid	1	TBD
Memory			
mon_lli_ddr_c	1	DDR3 memory bus	
bank_mm	2	DDR3 memory bank	
ddr_200_clk	1	DDR3 200MHz clock	
Monitoring			
user_mon_monitoring_data_c	32	Monitoring data	
adc_ped_st	1	ADC data after substracting pedestal. 12 bits per SC	
	Signal/Bus	**Width**	**Description**
	data	24	ADC data without pedestal data
	startofpacket	1	First cell
	valid	1	Valid data from user code
quality_st	1	Quality from combine block. 4 bits per SC	
	Signal/Bus	**Width**	**Description**

Table 85: LATOME High-level interfaces description (part)

Interface	#	Description	
	data	8	Quality data
	startofpacket	1	First cell
	valid	1	Valid data from user code
raw_adc_st	1	ADC data before substracting pedestal. 12 bits per SC	
	Signal/Bus	**Width**	**Description**
	data	24	ADC data
	startofpacket	1	First cell
	valid	1	Valid data from user code
sat_detect_st	1	Output of the saturation detection. 2 bits per SC	
	Signal/Bus	**Width**	**Description**
	data	4	Saturation detection data
	startofpacket	1	First cell
	valid	1	Valid data from user code
transverse_e_id_st	1	Transverse energy E_T from combine block. 14 bits per SC	
	Signal/Bus	**Width**	**Description**
	data	28	E_T data
	startofpacket	1	First cell
	valid	1	Valid data from user code
transverse_e_st	1	Transverse energy E_T from filtering block. 14 bits per SC	
	Signal/Bus	**Width**	**Description**
	data	28	E_T data
	startofpacket	1	First cell
	valid	1	Valid data from user code
Monitoring data			
osum_mon_monitoring_data_st	1	Monitoring data	
	Signal/Bus	**Width**	**Description**
	data	32	Monitoring data
	valid	1	Monitoring is valid
Registers			
ipctrl_istage_reg_mm	1	Input stage registers	
ipctrl_lli_reg_mm	1	Low-level interface registers	
ipctrl_mon_reg_mm	1	Monitoring registers	
ipctrl_osum_reg_mm	1	Output summing registers	
ipctrl_remap_reg_mm	1	Configurable remapping registers	
ipctrl_ttc_reg_mm	1	TTC registers	
ipctrl_user_reg_mm	1	User code registers	
TTC data			
ttc_istage_bcid_st	1	Current BCID value provided by the TTC receiver	
	Signal/Bus	**Width**	**Description**
	data	12	Received BCID
	valid	1	BCID is valid
XAUI Link			
lli_mon_xaui_link_c	1	XAUI Monitoring link	

Table 85: LATOME High-level interfaces description (part)

Interface	#	Description	
rx_data_st	1	RX data bus	
	Signal/Bus	**Width**	**Description**
	data	32	TBD
	empty	2	TBD
	endofpacket	1	TBD
	error	6	TBD
	ready	1	TBD
	startofpacket	1	TBD
rx_status_st	1	RX status bus	
	Signal/Bus	**Width**	**Description**
	data	40	TBD
	error	7	TBD
	valid	1	TBD
tx_data_st	1	TX data bus	
	Signal/Bus	**Width**	**Description**
	data	32	TBD
	empty	2	TBD
	endofpacket	1	TBD
	error	2	TBD
	startofpacket	1	TBD
	valid	1	TBD
tx_pause_st	1	TX pause bus	
	Signal/Bus	**Width**	**Description**
	data	32	TBD
tx_status_st	1	TX status bus	
	Signal/Bus	**Width**	**Description**
	data	40	TBD
	error	7	TBD
	valid	1	TBD
xaui_312_5_clk	1	Transceiver clock	

Table 85: LATOME High-level interfaces description

6.4 Code structure

The code generated from the *top_level.py* file is located in the *LATOME/src* directory. Each module's code located in the *LATOME/src/module_name* directory is organised according to section 3.2.2.5.

Of all generated code, the *module_arch.vhd* file should be the only one to be modified by the module's owner.

Module's specific source code used for both simulation and compilation should be added in the module's directory. There is no restriction about creating a directory structure within the module's directory.

Each file needed only for simulation should be located within the *test* directory structure, there is also no restriction how this directory is organized. The provided *test/testbench* directory should be used as a based to develop further testbenches.

Each module's directory is protected by access rights. Only the module's owner is allowed to modify the code. All other members of the team are allowed however to browse the code. The access rights are handled in the gitolite-admin CERN repository [9]. Access rights are handled by the e-group owner, please refer to section 2.3 if you have access rights issues.

A Cygwin installation

If you are running on *Linux* environment, you do not need to follow what is described here. This section is only useful for running on *Windows* environment.

There exist multiple solution for this purpose, you can use anything you like however it has only been tested on *Cygwin* which i recommend.

Cygwin can be downloaded from the following address: https://cygwin.com/install.html

You should download either '$setup - x86.exe$' or '$setup - x86\_64.exe$' depending on your machine.

A.1 Choose source

Start '$setup - x86.exe$' or '$setup - x86\_64.exe$' depending on which one you downloaded. Click on next to go to the next step.

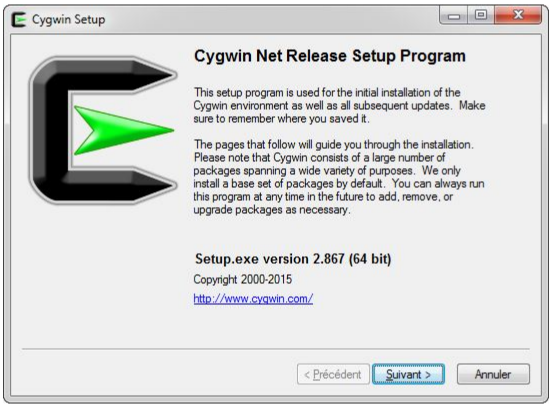

Figure 43: Cygwin setup step 1

Select '$Install\ from\ internet$' and click on next to go to the next step.

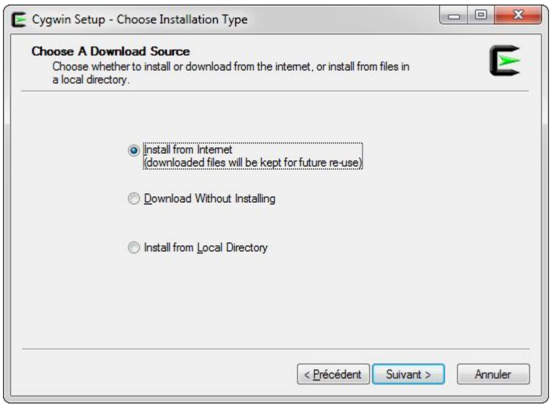

Figure 44: Cygwin setup step 2

A.2 Choose destination

Select the directory where you want *Cygwin* to be installed.

Click on next to go to the next step.

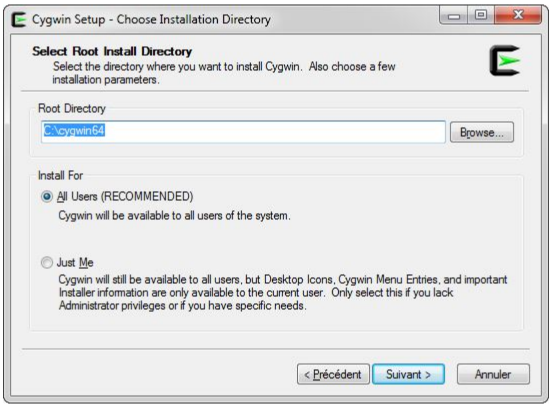

Figure 45: Cygwin setup step 3

Select where you want to store downloaded packages used to install *Cygwin*. You should keep the downloaded files, at the end of the setup a log file will be created, this will be used in case you want to add packages or update them.

Click on next to go to the next step.

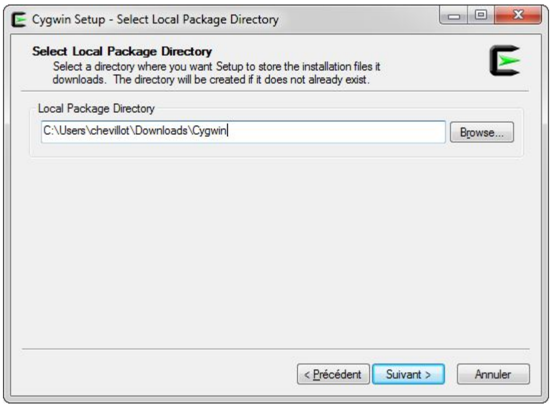

Figure 46: Cygwin setup step 4

A.3 Configure internet connection

Configure how setup should connect to the internet to retrieve the packages.

Click on next to go to the next step.

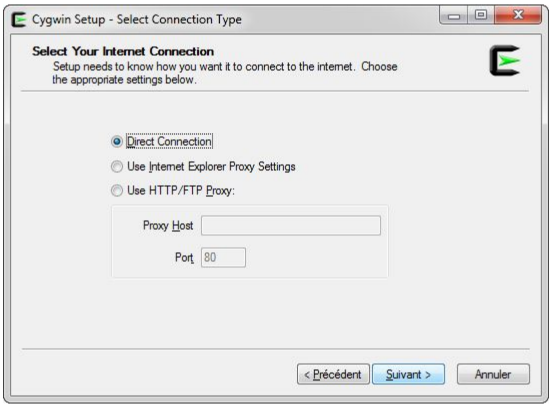

Figure 47: Cygwin setup step 5

Select a download site where to download all packages from. It is better to select a site which is close to you. For example in France you would select something like: http://gd.tuwien.ac.at. It only affects the time taken to download the packages. If it takes too long you can always cancel the process, restart the setup and choose another mirror site.

Click on next to go to the next step.

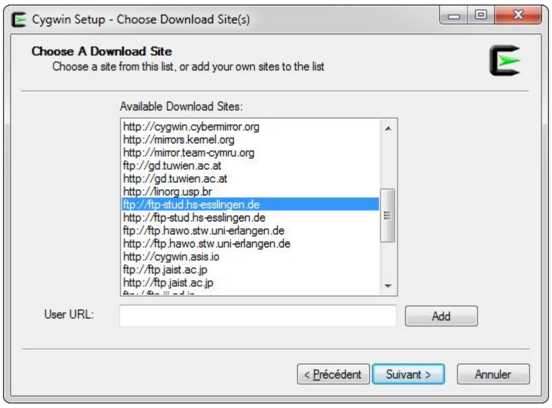

Figure 48: Cygwin setup step 6

A.4 Install packages

The following packages have to be installed to be able to work with the environment:

Package name	Version	Package full name
make	$\geq v4.0$	make: The GNU version of the 'make' utility
git	$\geq v2.1$	git: Distributed version control system
python	$\geq v2.7$	python: Python language interpreter
gettext	N/A	gettext: GNU Internationalization library and core utilities
xorg-server	N/A	xorg-server: X.Org X servers
xorg-docs	N/A	xorg-docs: X.Org X documentation
xinit	N/A	xinit: X.Org X server launcher

Table 86: Cygwin mandatory packages

The easiest way to select a package for installation is to change the view to '*Full*' by clicking on the '*View*' button on the top-right corner. In the search box, enter the package name (warning: do not type 'enter' as this will go to the next step. If you do you can always go back on e step once the package is installed). Click on the right of the ⟲ symbol until you find a version that matches the above mentioned one.

Once all packages with the correct version are selected for installation, click on '*next*' yet.

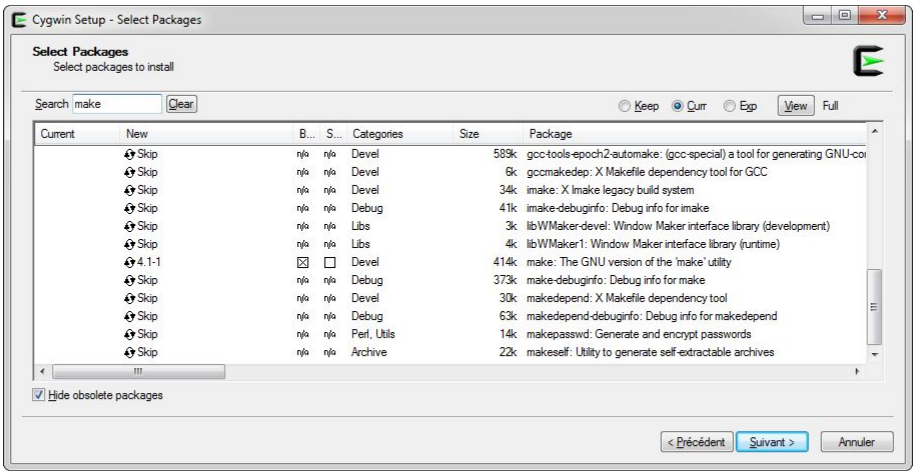

Figure 49: Cygwin setup step 7

Click on '*next*' to accept all dependencies.

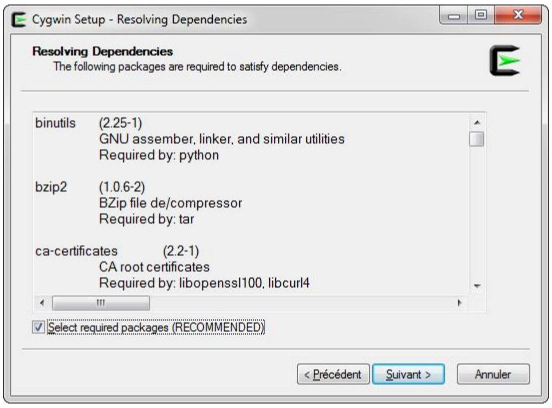

Figure 50: Cygwin setup step 8

A.5 Create shortcut

Select '*Create icon on desktop*' to have a shortcut to the shell interface. Click on '*Terminer*' to finish installation.

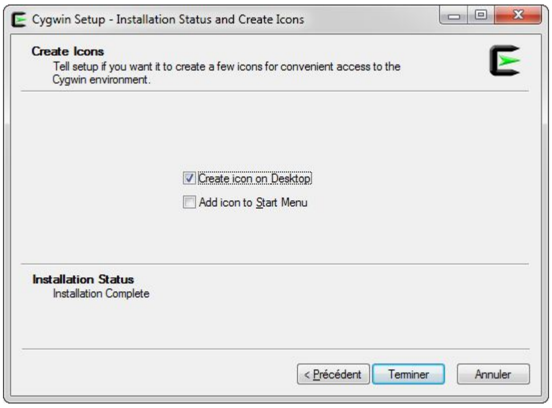

Figure 51: Cygwin setup step 9

The shortcut created by the Cygwin setup will open a terminal starting in your home directory (from Cygwin). You can easily create a new shortcut (copy/paste the existing one) and configure it to start in another directory, for example the $atlas-lar-ldpb-firware$ directory (or anything else).

Copy/paste the existing shortcut. Right-click on the new shortcut, choose '*properties*' and replace the *target* by:

- *path_to_cygwin*\bin\mintty.exe /bin/env CHERE_INVOKING=1 /bin/bash -l

The *start in* should be set to the path where you would like to start.

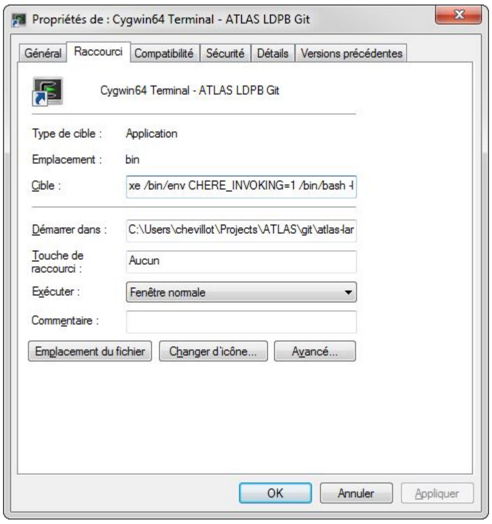

Figure 52: Cygwin shortcut

B GIT for Windows installation

In order to access GIT repositories from the Tortoise GIT GUI, you need to install GIT for Windows. You can download the latest version of the software from:

- https://msysgit.github.io/

Click on next to go to the next step:

Figure 53: GIT for windows installation step 1

Click on next to go to the next step:

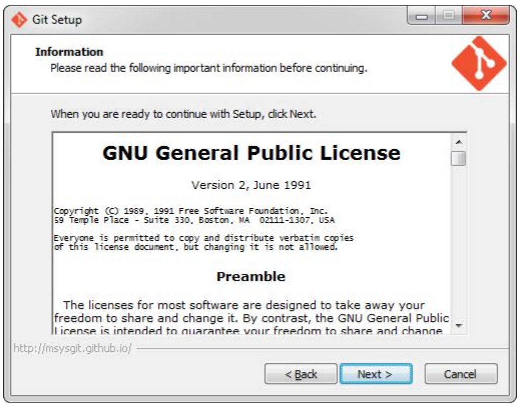

Figure 54: GIT for windows installation step 2

Click on next to go to the next step:

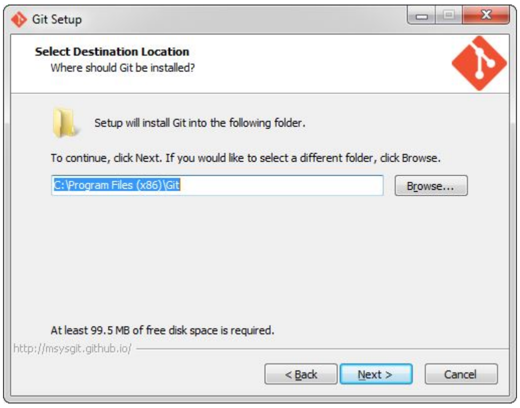

Figure 55: GIT for windows installation step 3

Click on next to go to the next step:

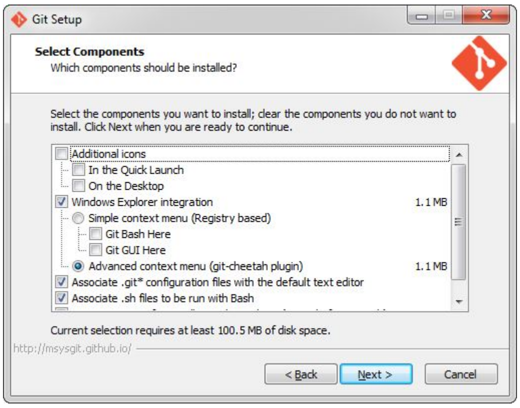

Figure 56: GIT for windows installation step 4

Click on next to go to the next step:

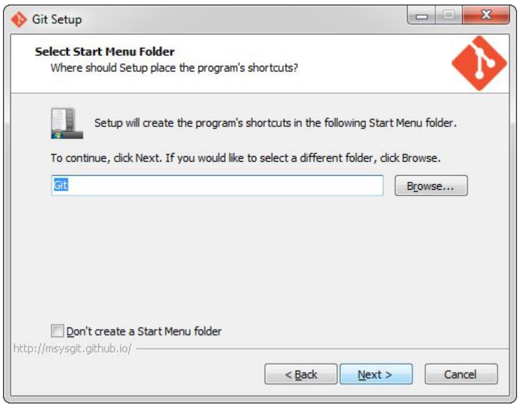

Figure 57: GIT for windows installation step 5

Click on next to go to the next step:

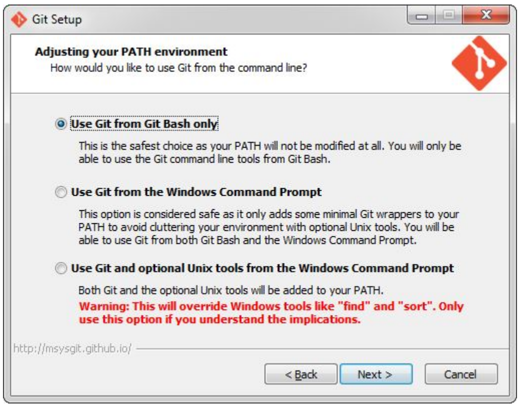

Figure 58: GIT for windows installation step 6

Select 'Checkout as-is, unix style'. Click on next to go to the next step:

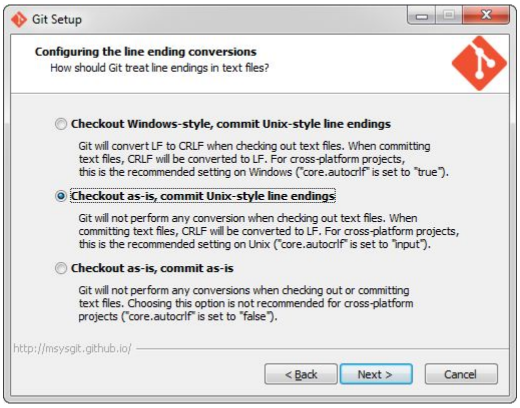

Figure 59: GIT for windows installation step 7

Click on next to go to the next step:

Figure 60: GIT for windows installation step 8

C Tortoise GIT installation

Tortoise GIT GUI can be used to access GIT repositories using a graphical interface on Windows. It has the same functionality as the command line. You will need to install GIT for windows which is described in section B.

You can download the latest version of the software from:

- https://code.google.com/p/tortoisegit/

Default settings are used here, you can customize to your needs.

Figure 61: Tortoise GIT installation step 1

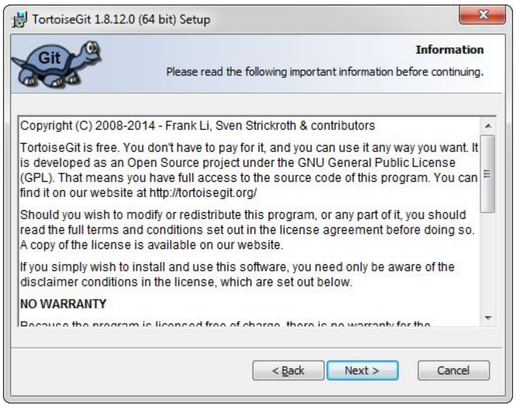

Figure 62: Tortoise GIT installation step 2

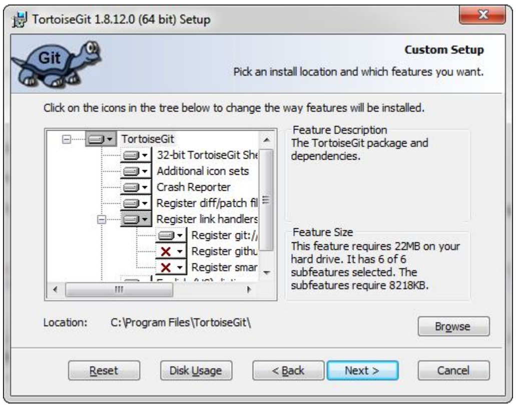

Figure 63: Tortoise GIT installation step 3

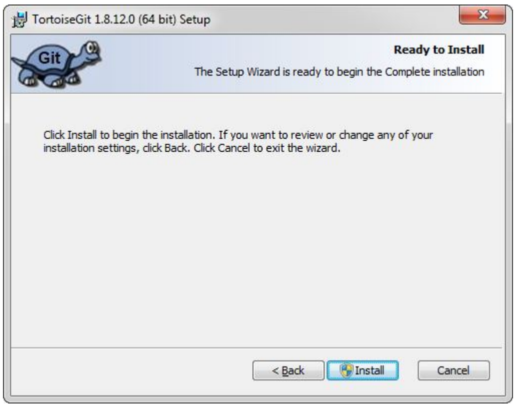

Figure 64: Tortoise GIT installation step 4

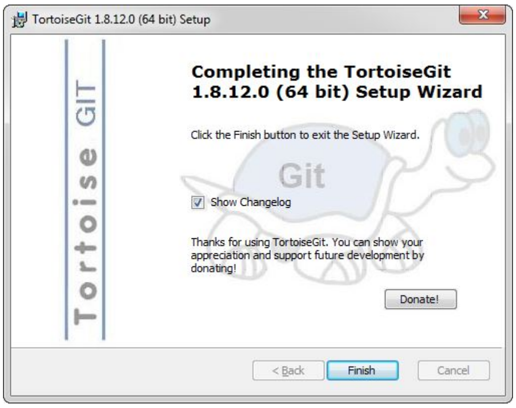

Figure 65: Tortoise GIT installation step 5

List of Figures

List of Tables

References

[1] Getting Started - First-Time Git Setup: http://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

[2] Notepad++: https://notepad-plus-plus.org/fr

[3] Meld: http://meldmerge.org/

[4] GIT webfront: https://git.cern.ch/web/atlas-lar-ldpb-firmware.git

[5] JIRA webfront: https://its.cern.ch/jira/browse/LDPBFW/?selectedTab=com.atlassian.jira.jira-projects-plugin:summary-panel

[6] JIRA documentation: https://confluence.atlassian.com/display/JIRA063/JIRA+Documentation

[7] Processing JIRA issues with GIT commit messages https://confluence.atlassian.com/display/JIRACLOUD/Processing+JIRA+issues+with+commit+messages

[8] LAr-LATOME-FW firmware documentation: https://git.cern.ch/web/atlas-lar-ldpb-firmware.git/blob/HEAD:/LATOME/doc/LAr-LATOME-FW/LAr-LATOME-FW.pdf

[9] LAr-LATOME-FW firmware access rights: https://git.cern.ch/web/gitolite-admin.git/blob/HEAD:/conf/subs/atlas-lar-ldpb-firmware.conf

[10] ATLAS Lar LDPB Firmware JIRA: https://its.cern.ch/jira/browse/LDPBFW/?selectedTab=com.atlassian.jira.jira-projects-plugin:summary-panel