

Getting started with Git centralized workflow for PSoC Creator

This document explains how to set up a Git repository and how to use it with centralized workflow. Centralized workflow does not unleash the full potential of Git. This workflow suits small teams (or teams with little or no version control experience). If you already are a master of Git, feel free to use more advanced workflows.

More about centralized workflow:

<https://www.atlassian.com/git/tutorials/comparing-workflows#centralized-workflow>

Download and Install Git

- Go to <https://git-scm.com/download/win>
- Install Git using the default values (just keep pressing Next until you are done)

Create a GitLab Account

Each member of the team needs a GitLab account to get full access to the repository. Go to **gitlab.com** and create an account. Alternatively, you can use GitHub.

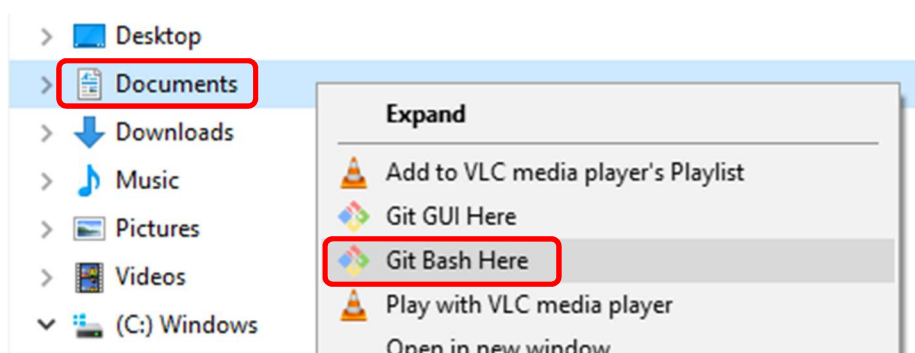
Create a repository

NOTE: Only one member of your group needs to set up a repository for your project. The following steps should only be completed by a single member of your team. The other team members will be added as collaborators to give them full access to the repository. However, it is important that everyone in the team takes part in the process to learn and understand how the process goes.

1. Log in to GitLab
2. Create a new **empty** project in GitLab

Give the newly created repository a name. Do not change any other settings. You will import the repository from your PC. Once your repository has been created, you will be directed to a page with instructions on how to add content to your repository. **Read further before taking any action to see the additional steps.** Keep the GitLab instruction page open in your browser. You will need the address of your newly created repository later during setup.

3. Copy sources from distribution repository
 - a. Select a directory where you want to copy the project to. Cloning will create a subdirectory for the project, so the directory does not have to be empty.
 - b. Right-click on the directory and select **Git Bash Here**:



- c. Run the following command in the Git Bash terminal:

```
git clone https://gitlab.metropolia.fi/lansk/Zumo.git
```

- d. You will be asked to enter your username and password. Use your Metropolia credentials (the ones you use for Oma)

4. Import the repository into GitLab

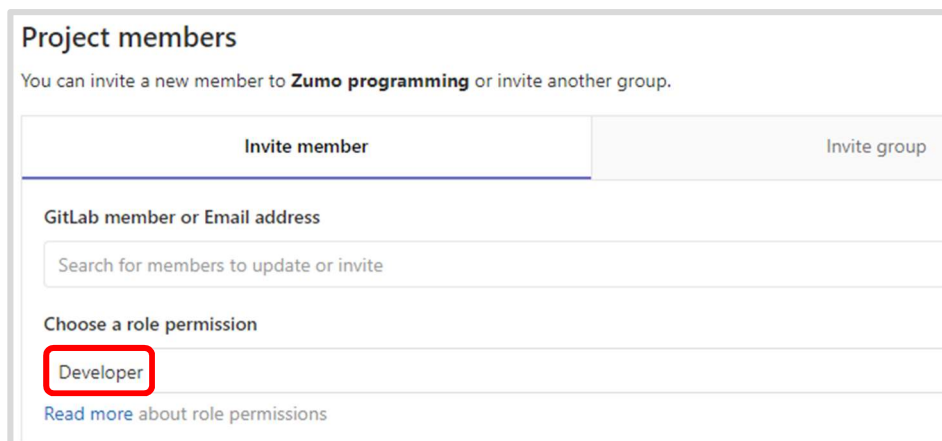
Enter your information into Git:

```
git config --global user.email my_email@metropolia.fi
git config --global user.name "Etunimi Sukunimi"
git config --global core.editor nano
```

The actual import:

```
cd existing_repo
git remote rename origin distribution-origin
git remote add origin https://gitlab.com/username/repo_name
git push -u origin --all
# You will be asked to enter your GitLab username and password
git push -u origin --tags
```

5. Add your team members as collaborators to the repository
 - a. On the page of the repo in GitLab go to Settings -> Members
 - b. Invite your team members to the project as Developers:



Project members

You can invite a new member to **Zumo programming** or invite another group.

Invite member **Invite group**

GitLab member or Email address

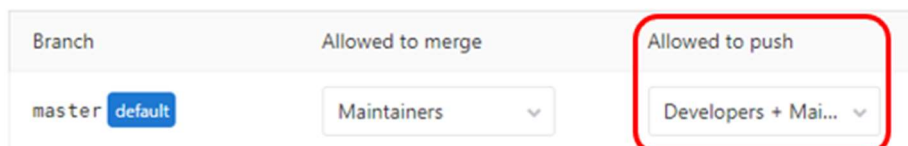
Search for members to update or invite

Choose a role permission

Developer

[Read more](#) about role permissions

6. Allow Developers to push changes to the repository
 - a. On the page of the repo in GitLab go to Settings -> Repository -> Protected Branches
 - b. Change the **"Allowed to push"** options to Developers + Maintainers:



Branch	Allowed to merge	Allowed to push
master default	Maintainers	Developers + Mai...

7. You are done with creating the repository. Now you and your team can start working on the project and learning how to use Git for version control.

Clone the repository

NOTE: The person who created the repository does not need to clone – the other team members do.

1. Open a Git Bash terminal in the directory you want to clone the repository into.
2. Clone the repository. The directory does not need to be empty as the cloning will create a subdirectory that has the same name as your project in GitLab.

Run the following command in the terminal to clone the repository (the address is the address of your repository in GitLab):

```
git clone https://gitlab.com/username/repo_name
```

Enter your information into Git:

```
git config --global user.email my_email@metropolia.fi  
git config --global user.name "Firstname Lastname"  
git config --global core.editor nano
```

Start working with Git

1. Making local changes

Not that you have created a local copy of the project by cloning, you can start making your own changes to the code. Once you have made a change and would like to save your progress, it's time to make a local **commit**. Make sure you have saved your progress in PSoC Creator (Ctrl + S). First you will need to **stage** the files that you have changed and want to add to the commit. This allows you to only include changes to files that are necessary to the specific change you want to commit:

```
# Check which files have been changed
git status

# Stage the file(s) you want to include in the commit
git add file1 file2

# Commit the staged file(s)
git commit -m "Commit message"
```

The role of the commit message is to specify what was changed to make it easier to track down a specific commit. Here are some principles for making commits:

1. Commit early and commit often

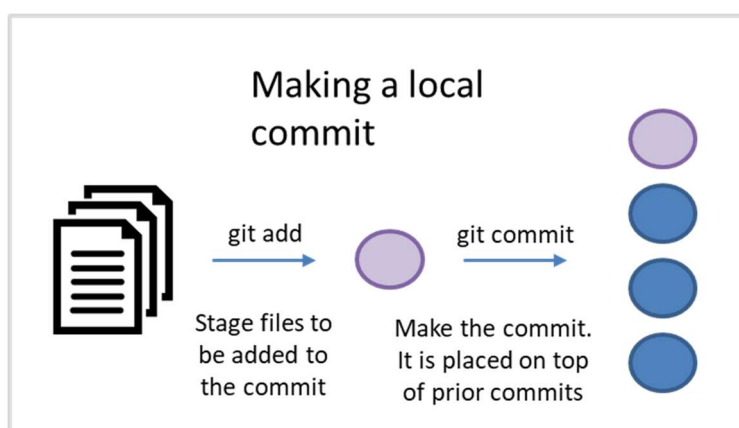
Make a commit after each small, meaningful change that adds, removes or modifies a single, isolated feature or function or to save your current progress. Think of a commit as a checkpoint in your work. When changing the code in small steps, it is also easier to see when and how you made a mistake in case something suddenly stops working.

2. Write meaningful commit messages

Just as you want the names of variables and functions in the program itself to be consistent and descriptive, you want the same for your commit messages. You want to understand what you have done when you later look at your old commits.

3. Keep your local and the remote repository in sync

Remember to pull updates made by your team members from the remote repository and to push your own changes so that the others can apply your changes into their local repositories. This is the only way you will get your team member's updates and they will get yours.



Changing your last commit

In case you forgot to include some changes or a commit message in your last commit, you can stage the changes if needed and commit again using the `--amend` option:

```
git add forgotten_file
git commit --amend -m "Commit message"
```

2. Pulling updates from the central repository

Your local repository is completely isolated from the central repository. Changes made by others will not be applied to your local repository until you pull them from the central repository.

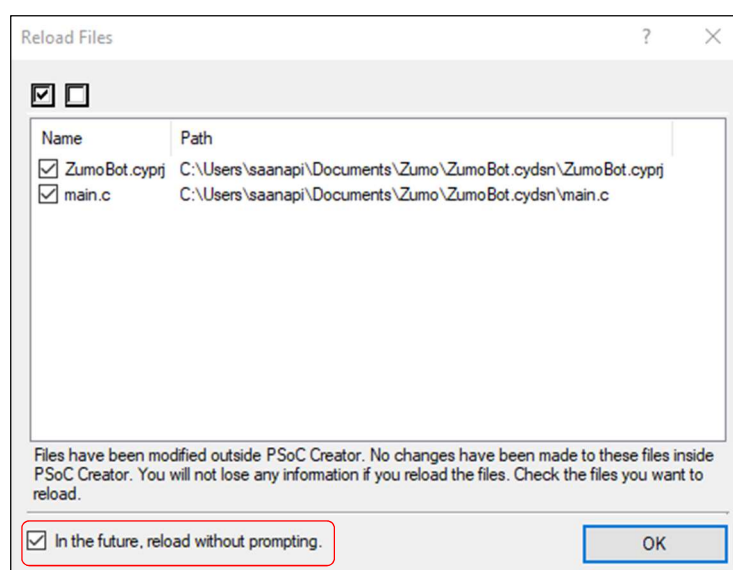
```
# Get updates made to the central repository
git pull --rebase origin master
```

With the `--rebase` option, the pull command adds all the new changes in the remote master on top of the local master and then places all local commits on top of that one by one. Without the `--rebase` option the remote changes and your local commits are **merged** into single merge commit.

Before you can run the **git pull** command you must either commit, discard or stash any local, uncommitted changes. Stashing changes for later use is explained in the next subchapter, **'Stashing uncommitted changes'**.

When pulling changes from a remote repository, you might face a **merge conflict** which must be handled before the updates can be applied. Chapter 3 discusses what merge conflicts are and how they can be fixed.

The first time you run a Git pull with the PSoC Creator software open, a window will pop up warning that files have been modified outside PSoC Creator. When this happens, check the box saying: **"In the future, reload without prompting."** This means that changes to files will be reloaded automatically and there is no need for manually refreshing the interface when you for example pull updates using Git:



Stashing uncommitted changes

Git pull will fail in case you have any uncommitted, local changes. Before pulling updates to your local repository, you must either **commit** or **stash** those changes. In case you are not ready to commit your current work yet, you can **stash** your uncommitted changes for later use using the **git stash** command:

```
# Stash local changes for later use
git stash

# Now you can pull updates
git pull --rebase origin master

# Add your previously stashed changes back to your current progress
git stash pop
```

Pulling updates from the distribution repository

NOTE: The person who created the team repository should do this.

It is also possible to apply updates that have been made to the distribution repository where you first got your project repository from. The team member who created the repository renamed the **origin** of their local version of the project **distribution-origin** before adding the team's new remote repository as **origin**. This was done to separate the distribution repository from the **origin** of your own project repository:

Name of the remote repository	Description	Address of the remote repository
origin	The remote central repository of your own project	https://gitlab.com/username/repo_name
distribution-origin	The remote distribution repository	https://gitlab.metropolia.fi/lansk/Zumo.git

Pulling updates made to the distribution repository works the same way as when pulling from your own central repository. Before pulling updates from the distribution repository, you must make sure you have the latest changes to your own remote repository:

```
# Get updates made to your own remote repository
git pull --rebase origin master

# Get updates made to the distribution repository
git pull distribution-origin master

# Push the changes to the remote repository
git push origin master
```

3. Fixing merge conflicts

Pulling updates from the remote master sometimes results in a **merge conflict**. This happens when there are changes to the same lines of code in both the updated version and your local commits. This may also happen when merging local branches together (More about branches in Chapter 7). When a merge conflict happens, you must choose which changes you want to keep.

Merge conflict after a **git pull --rebase** operation:

```
saanapi@SAANAPI MINGW64 ~/Documents/Zumo (master) $
$ git pull --rebase origin master                                # Running git pull
From https://gitlab.com/saanalect/zumo-programming
* branch          master      -> FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: Changed the startup message                            # Name of the commit(s)
Using index info to reconstruct a base tree...                  # with conflicts
M       ZumoBot.cysdn/main.c
Falling back to patching base and 3-way merge...
Auto-merging ZumoBot.cysdn/main.c
CONFLICT (content): Merge conflict in ZumoBot.cysdn/main.c     # Conflicting file(s)
error: Failed to merge in the changes.
. . .
```

You can also check which files have conflicts using the **git status** command.

Git marks the conflicted lines of code directly into the affected files. You can use the **PSoC Creator** or any text editor such as **Notepad++** or **Microsoft Code Writer** to fix the conflicts. The following caption shows how Git marks conflicting lines in the code:

```
<<<<<< HEAD
    printf("\nHello, World!\n");          // The line(s) in remote master
=====                                // Divides the conflicting parts
    printf("Program starts\n");          // Local version of the line(s)
>>>>>> Changed the startup message      // The name of the commit
```

In this case you want to keep your local changes so you remove the markers as well as the remote master's version and should be left with:

```
printf("Program starts\n");
```

For the change to take effect, you must now **save your changes** in the editor and **continue the rebasing**. In case the merge conflict appears when **merging** instead of rebasing, you need to create a new merge commit after you have fixed the conflict:

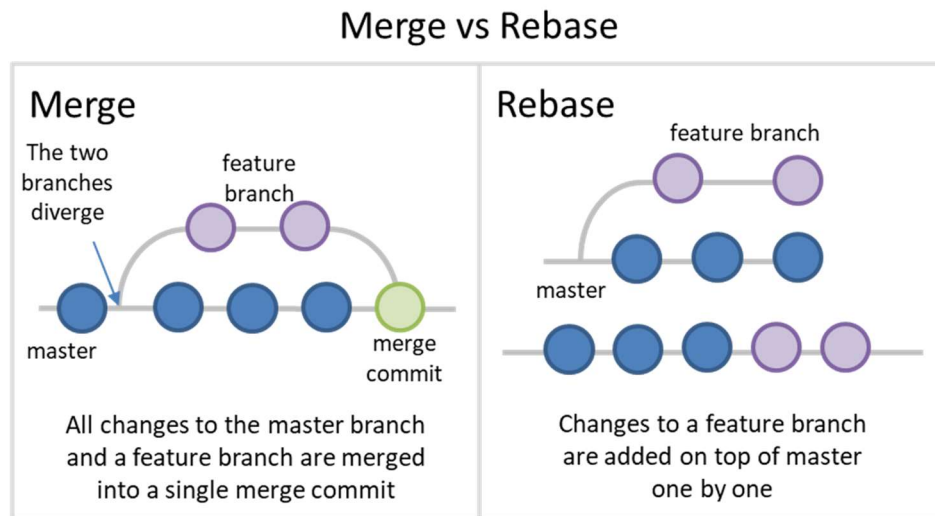
REBASE:

```
git add Zumobot.cysdn/main.c      # Stage the fixed files
git rebase --continue              # Continue the rebase
```

MERGE:

```
git add Zumobot.cysdn/main.c      # Stage the fixed files
git commit -m "Commit message"    # Create a merge commit
```


Note that sometimes you may have conflicts in several commits, which means that during a rebase you have to solve them one commit at a time. The following figure shows the difference between merge and rebase operations.



4. Pushing local commits to the central repository

Once you have made a local commit, the changes must be pushed to the remote central repository so that other team members are able access them. Before pushing the changes, you must first make sure that your local repository is up to date with the current state of the central repository.

```
# Fetch updates to the central repository
git pull --rebase origin master

# Fix possible merge conflicts to complete the rebase

# Push your local commits to the central repository
git push origin master
```

5. Project history

The **git log --oneline** command can be used for viewing a brief version of the commit history of your local repository. The command lists the different commits you have made so far and your most recent commit shows at the top of the list. The log also shows the commits where your different branches point to. Each commit has a unique **commit ID** that is shown before the name of the commit:

```
git log --oneline

1cc80b8 (HEAD -> master) Increased delay
e83f614 Added variable for delay (origin/master)
05a8710 Changed message for motor controls
f1af2a9 Changed startup message

-----

HEAD: This is where your current working branch points to.
master: This is the last commit of your local master.
origin/master: This is where your remote master points to.
```

If you run the **git log** command without the **--oneline** option you will get a longer version of the log that includes additional information such as who made the commit and when. Furthermore, any commit messages with more than one line of text will be shown in full.

Viewing an old revision

Sometimes you want to take a look at how your repository looked at a specific commit without it affecting your existing work. You can do this by checking out the commit using its **commit ID**. Let's say you wanted to take a look at how your code was when the "Changed message for motor controls" was the last made commit. As you can see from the image above, the **commit ID** for that commit is **05a8710**:

```
git checkout 05a8710
```

When you check out a specific commit this way, you end up in a **'detached HEAD'** state. This means that the state of the repository you are looking at is not tied to any branch. You are able to make changes and even commit them without it affecting any of your existing branches. When you are ready, you can go back to your local master by issuing:

```
git checkout master
```

6. Undoing things

Sometimes you notice you have changed your code in an unwanted way and would like to go back in time where everything worked.

As you know, there are several phases in local changes:

- Local changes that have not been staged
- Local changes that have been staged with **git add** but not committed
- Changes that have been committed locally with **git commit**

There are several ways of undoing changes depending on which phase you are with them.

Viewing and discarding unstaged changes

If you want to see how your current work in progress is different from your last commit, you can use the **git diff** command:

```
git diff --git a/ZumoBot.cydsn/main.c b/ZumoBot.cydsn/main.c
index 891bdf0..611b52f 100644
--- a/ZumoBot.cydsn/main.c
+++ b/ZumoBot.cydsn/main.c
@@ -59,7 +59,7 @@ void zmain(void)
{
-   printf("The program starts\n");
-
-   int delay = 1000;
+   int delay = 120;

   while(true)
```

In the image above, the **red lines** with a **minus sign** at the beginning mark lines that have been removed since the last commit. **Green lines** with a **plus sign** indicate added lines. If a line has been changed but not removed, the command will show both the old (red) and new (green) version as with the changed delay in the image above.

If you have made some local changes that you are not happy with and would like to start fresh from your last commit, you can discard the changes made to a file by running:

```
#One file
git restore file_name

#All files
git restore .
```

Unstaging files

If you have changed files and already staged them with git add, you can **unstage** them as follows:

```
# Unstage one file
git restore --staged file_name

# Or if you want to unstage all staged files
git restore --staged .
```

Undoing your last local commit

Before removing a commit, make sure that your problem is not solved by discarding local changes, changing your last commit with **git commit --amend** or making a new commit to fix whatever was wrong with the previous one. If none of these solves your problem and you are sure you want to remove your last commit, you can do so by resetting your **HEAD** to a prior commit.

SOFT RESET:

```
# If you want to keep the changes in the commit
# the --soft flag will take your HEAD back one commit
# but keep the changes staged like they were before
# you committed them:

git reset --soft HEAD^

# If you run git log, you will see that you have gone
# back one commit. If you run git status, you will see
# your modified files show as staged
```

After you have reset your **HEAD**, you can unstage one or more files, make changes, stage the files again and commit them as usual.

HARD RESET:

```
# If you are completely sure that you want to
# get rid of the last commit and the changes
# in it for good, you can use the --hard flag

git reset --hard HEAD^
HEAD is now at e83f614 Added variable for delay

# You can use git log to see that you have gone
# back one commit and verify there are no staged
# or unstaged changes running git status.
```

7. Working with local branches

So far, we have been discussing two branches, the local master that you use for writing your own code and the remote master that holds the shared, official version of your project. Besides these automatically created branches, you can also create your own local branches. This is very useful when you want to develop different parts of the project simultaneously. Here's an example:

You start working on creating your own functions for controlling the motors of the robot, so you first create a new `motor_controls` branch, separate from your local master.

```
# Update your local master with latest changes to the remote master
git pull

# Create a new branch for motor control development
git branch motor_controls

# Switch to the motor_controls branch
git checkout motor_controls
```

Let's say you also want to fix a bug that someone from your team made earlier, making the speed of the motors too high. You can create another branch for fixing that specific feature which enables you to separate these two lines of development.

```
# Switch back to master branch
git checkout master

# Create a new branch and switch to it
git checkout -b speed_bug_fix
```

You can start working on a branch the same way you would with your local master: develop code, stage files and commit changes locally. Once you are done with everything you wanted to do in a branch, you can merge it into your master branch to be pushed to the remote.

```
# Switch back to master branch
git checkout master

# Merge the commits in the branch to your local master
git merge motor_controls

# Fix possible merge conflicts

# Delete the branch if you no longer need it
git branch -d motor_controls
```

To list all the existing branches, you can run the **git branch** command without arguments. The current branch will be marked with an asterisk *. Moreover, the Git Bash will show your current branch at the end of prompt as well:

```
saanapi@SAANAPI MINGW64 ~/Documents/Zumo (motor_controls) $
```