This is an in-depth guide intended for experienced programmers involved with merging branches and dealing with merge conflicts

Please read the following very carefully, know the standard GitHub usage guide well, and be especially careful when following any procedures. Bad things can happen if you don’t know what you are doing, and if you do know what you are doing, even worse things can happen.

Working with pull requests

Working with pull requests is typically easy because there is a nice big green button that you can click to merge the pull request on the remote repo. Unless there is some sort of conflict, Github will always be able to merge easily. However, there is a certain process that needs to be followed afterward regarding the left over branches. If something doesn’t seem right, try exiting the repo in Github for Windows, and reentering it at least twice. There doesn’t seem to be any other way to refresh it.

Depending on the preferred style of workflow, it may be necessary to delete branches to maintain a clean environment. There is no detriment to deleting branches already merged into their parent branch. The Github network graph is unaffected and you have one less branch that you would’ve never touched anyways.

Ideally, have each person unpublish the branch they created. Another method is just clicking on the delete branch button in the pull request after accepting it. The final method is under the branches page in Github. Unfortunately, I still don’t quite understand how to delete a branch in a way that smoothly disconnects everyone.

If the branch is already gone on Github, but still shows in Github for Windows and can’t be removed through that, then use the following.

Prune all remote repo branches

git fetch –p

This removes the references to the deleted branch. It should output which branches it deleted. Then delete the now local-only branch normally in Github for Windows.

List all branches

git branch –a

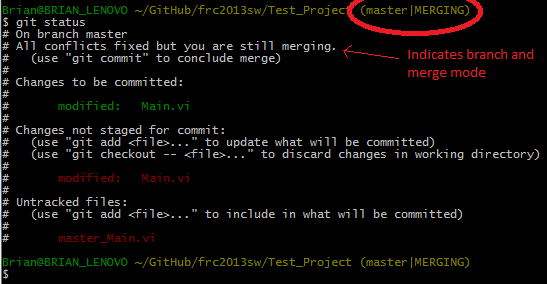
You can use this command to see all branches. Using without the –a option will only show some branches.

Useful Git Tools

The following are all using the Git Bash. Although different command line interfaces should be mostly the same, there may be some differences.

The status command

git status

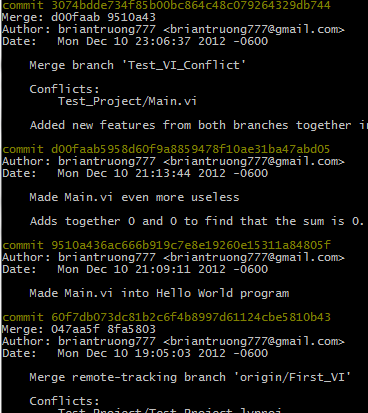


The output of the status command shows your current branch, files to be committed, files not staged for commit (will not be committed), and untracked files (files that were never committed before). It also gives you common commands for common actions. **Use the status command whenever you have no idea what you are doing. Then use it again just to make sure.**

Note: the example above is not typical and was taken from a merge in process.

The commit log

git log



The log command shows a log of all the commits on the current branch starting with the newest one on top. Press ‘q’ to quit, ‘h’ for help. ‘j’ goes down ‘k’ goes up or just use the arrow keys to move in all four directions. Don’t scroll with the mouse; it doesn’t work as you think.

Notice that each commit has a long hexadecimal identifier. In Git Bash, you can double click to highlight, right click to copy, and when typing in text to the Bash, right click to paste.

Access help files

git <tool> -h

git help <tool>

Use the previous with any tool in place of <tool>. This includes status, add, etc. The first command shows you the tool’s possible options and basic usage. The second option opens a local html file detailing exact usage.

Making/deleting/using tags

As explained in the GitHub\_Guide\_2013.pptx, tags are simply references to a particular commit so it can be referred to by a nice name rather than hexadecimal. Although they can easily be used in the GitHub website, they cannot be made/deleted there or in GitHub for Windows. You will need to push all tags to the server manually (see later steps).

Listing tags

git tag

This lists out all the tags for this project.

Making tags

git tag <tag\_name>

Makes a new tag at your current commit

Deleting tags locally

git tag –d <tag\_name>

This deletes the tag only locally, not on the server.

Deleting tags on the server

git push <remote\_repo> :<tag\_name>

The <remote\_repo> will probably be “origin”. This deletes the tag on the server only. Don’t forget the colon before the <tag\_name>.

Pushing tags to server

git push --tags <remote\_repo>

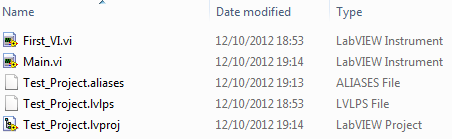
The <remote\_repo> will probably be “origin”. Only pushes new tags to server.

Fetching tags from server

Git fetch –tags <remote\_repo>

The <remote\_repo> will probably be “origin”. Only pulls new tags from server.

Typical Labview Project Structure



The \*.aliases, \*.lvlps, and \*.lvproj files are all integral to the Labview project. In all honesty, I don’t really know what the \*.aliases and \*.lvlps files do. **However, all three files are text files.** You can open them in notepad or vim like any other text file. In particular, \*.lvproj is an xml file that maintains the project directory structure you see when you open a Labview project. If any files or virtual folders are added, removed, or renamed through the project window in Labview, the \*.lvproj file is altered. **If two different Git branches modify this file (which can easily happen), you will have to manually merge the changes to this file.**

Avoiding merge conflicts

Ideally, if someone needs a new files added, removed, or renamed, this is done by altering the master branch to create a \*.lvproj file that has an xml entry for the new file/virtual folder. Then the change is added to sub-branches by merging from master to sub-branch. Such universal changes should be merged to **all** branches for consistency.

If those branches have not touched the \*.lvproj since branching off, Git should be able to easily merge since only the master branch has changed the file. Otherwise, you will need to resolve the merge conflict.

Much like the \*.lvproj file, the \*.aliases and \*.lvlps files also exist at the project level, and thus need to be watched carefully.

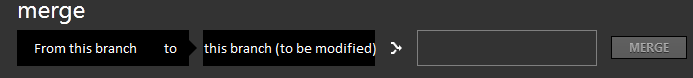
**Obviously, don’t have multiple branches change the same files.** Otherwise, you will have to manually merge them too.

Merging from sub-branch to master branch using the command line when there is a merge conflict

**The following should only be done if GitHub cannot automatically merge through a pull request.** Either you failed to follow the previous advice, something unexpected occurred, or this was just completely unavoidable. Just remember, you are not in a happy place. You should also make sure you have no uncommitted work or open Labview files (but this should be a given for any major alterations to the project).

The easier way to start a merge:

1. In GitHub for Windows, click on the branch icon, then click on manage branches
2. Drag the appropriate branches with the source branch on the left and destination branch on the right, and click merge



1. GitHub for Windows should complain of a merge conflict
2. Click on the Open Shell button (or something similar) to open the shell
3. You should now be in merge mode
4. Skip “The harder way to start a merge”

The harder way to start a merge (not really):

1. Open your command line of choice to the repo directory
2. Checkout the destination branch (the one to be modified)

git checkout destination\_branch

1. Merge from the source branch to the current branch (the destination branch)

git merge source\_branch

1. Git should complain of a merge conflict
2. You should now be in merge mode

Use the status command to see what files are conflicting and what files are already in merging commit

git status

See the section on the status command near the beginning of the document.

You can also use the diff command to compare differences between text files.

git diff

When using the diff tool, press ‘q’ to quit and ‘h’ for help. Use the help to figure out the rest.

By default, Git adds new text in conflicting text files that indicate where the differences are and what are the different versions. **These additional lines of text do not follow xml format and thus Labview will fail to open the \*.lvproj file if they are not removed.** See later sections on how to merge the \*.lvproj file.

**If at any point, something goes wrong or you messed up, use the following to abort the merge and return to normal operation.**

git merge --abort

You will lose any uncommitted work (but you shouldn’t have any of that anyways). You can then checkout any branch and try again. Or give up.

When you run the status command, all files to be committed should be displayed in green. Those that are suffering from a merge conflict should be in red. **You must now manually merge by editing the conflicting files and then adding them into the merging commit.**

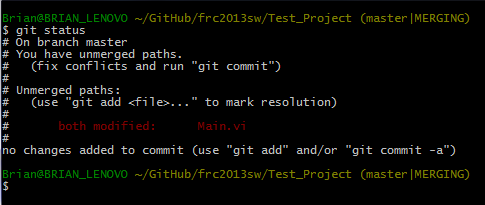
Okay, I sort of lied. Labview has a merge tool that is probably available for you to use. Depending on how much time I’ve spent or future admins, these tools may already be integrated into your Git environment. Otherwise, there will be a section on them later in this document.

Addressing merge conflicts in the \*.vi files manually

Uh oh. I’m going to be honest. This isn’t going to be easy. You are probably going to have to open the two files, side by side. Some of the following can be done in Windows Explorer, but I will show the commands for Git Bash. Feel free to use your favorite command line interface. You will need to open the \*.vi files at some point so Labview or Windows Explorer is needed to do that. Be careful with modifying files with Git while the files are open in Labview (or just don’t do it).

1. Use the status command to see which files are conflicting

git status



1. The file currently in your directory is the file **from the current branch.** Rename it to an appropriate name so you can later differentiate it (remember the .vi on the end)

mv old\_file\_name new\_file\_name



1. Checkout the other version of the file from the source branch

git checkout source\_branch -- old\_file\_name



1. You now have both versions of the file, one of them renamed from before



1. Open the project, and manually **add the merged changes to the file with its original name.** In other words, to the vi you open in the Labview project (in this example, the one named Main.vi).
2. Move on to the “Adding a file into merging commit…” section

Addressing merge conflicts in the \*.lvproj, \*.aliases, and \*.lvlps files

Although all three are text files and thus editable in notepad, only the \*.lvproj file is an xml. If you don’t know xml, luckily for you, xml files are not that hard to understand. You really should just look them up. They are quite standard and I intend on teaching only the minimum here, and by minimum I mean nothing. If you know html, you should be right at home (html is really just a type of xml). Just look at the rest of the file and how it represents the data in the project viewer in Labview.

Whatever you do, don’t forget to remove the markers Git has added. The markers are added wherever there are conflicts so they won’t be in files without conflicts.

Adding a file into merging commit

git add file\_name

This command adds a file to the “Changes to be committed” section in the status command’s output. **Only files in the “Changes to be committed” section will actually be committed.** (No, don’t say it.) Anyways, be sure you don’t accidentally add your renamed files or any other junk. Again, use the status command to confirm what you are doing before committing.

Removing a file from the “Changes to be committed” section

git rm --cached file\_name

This command removes a file from the Git index and **does not** affect the file in your working directory. **Not to be confused with the rm command which will delete your file from the directory. Permanantly. Make sure “git” is in front of “rm”.**

When you have finished fixing all conflicting files, simply make a merging commit with the newly added files.

Making a merging commit (really just the normal commit process)

git commit

A text editor will show up allowing you to write the commit’s message. Be as descriptive as possible on exactly what changes you made to each file. Too much information is better than too little when it comes to fixing merge conflicts. If you want to cancel, make the text file blank, and the commit will be cancelled.

Don’t forget to delete the renamed files you have. Otherwise, when you go back to GitHub for Windows, they will show up to annoy you. You can also delete them in GitHub for Windows by right clicking them and selecting “discard changes”.

Exiting the command line

exit

In almost all command line interfaces, just type exit to exit. Logical, right?

Sync

Back in GitHub for Windows, you should see your new masterpiece of a commit on the top of the list. Of course, it isn’t synced yet, so go ahead and triumphantly hit that sync button. You deserve it!

Congratulations, you have completed a (hopefully) successful manual merge. You are now familiar in the ways of Git. I wish you luck on your future journeys and merges.

Using the Labview merge tool

It is possible I or some future admin set up Git to somehow use the Labview merge tools. In that case, just use whatever process is given there. Otherwise, the following probably does exactly what those tools do except manually. This is a simplistic guide that assumes you already know how to manually merge from above.

1. Use GitHub’s network branch to find the commit where the branches diverged
2. Get the vi in question from that commit
   1. This vi will be the base vi
3. Open the merge tool through Tools->Merge->Merge VIs…
4. Choose the appropriate files for each of the three text boxes
   1. Yours and theirs can be switched, I don’t think it matters
5. Your screen will fill with many windows
   1. Two windows will be the normal front panel and block diagram editor where you create the new resultant vi from the merge
   2. Two windows will show the front panel and block diagram of the three vi’s side-by-side
   3. One window will list all the conflicts between the vi’s
6. That conflict viewer will have check marks and x’s indicating from which vi the resultant vi will take from
7. Select conflicts and click on the buttons on the side to decide
   1. All conflicts must be handled in this manner in order to save the resultant vi
8. Manually change the resultant vi as needed
9. Click on “Close” in the conflict viewer and save the resultant vi

The following are extra things that you probably won’t need, but are still worth knowing. They are also extremely dangerous.

Deleting commits

It is possible to delete the newest commits and essentially make as if they never existed. **This is incredibly dangerous because you will be removing those commits on all remote repos too.** In other words, many hours of work could easily disappear when users pull from the servers only to be missing a commit they needed. **You should only use this if you have not pushed the commits to a remote repo.** Otherwise, others may see and use commits that will shortly fail to exist, causing their work to be wasted.

There is an alternative to this called a revert. That operation simply makes a new commit that undoes a previous commit (or commits). If you have already pushed the commits, use that instead. That way, other people can still access their work in the unwanted commit. Just use the button for it in GitHub for Windows.

Reset both the working directory and index (Git’s history of commits) to a certain commit

reset --hard <commit>

For <commit>, you can put in the hexadecimal commit identifiers, a certain branch (will only be useful if chosen branch is behind current branch), or a tag (I think). You can use this format: “origin/<branch\_name>” for <commit> to overwrite the local repo with the commit/branch from the remote repo.

Afterwards, you will need to push your deletion of the commits to the remote repo. This is your last chance to undo the deletion using that reset thing.