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.

Command line examples explanation

For command line examples, angle brackets (<>) are used to indicate that the word (including brackets) must be replaced with the appropriate item for the situation when typed into the console. For example,

whereis <your\_name>

should be replaced with

whereis Brian\_Truong

if you have the same name as me. The replacement must be one word. Use a backslash followed by a space to represent a space:

whereis Brian\ Truong

By the way, don’t think the above command actually works. I’ve covered my tracks too well for that.

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. You can also hit f5 to refresh it as well.

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.

Have someone unpublish the branch to delete it. Alternatively, try clicking on the delete branch button in the pull request after accepting it. The final method is under the branches page in Github (the website). 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 or through the command line.

Delete a branch

git branch –d <branch\_name>

List all branches

git branch –a

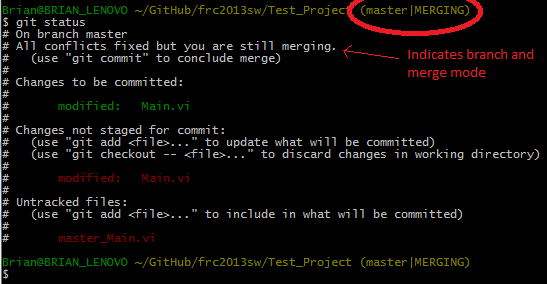
You can use this command to see all branches. Using without the –a option will only show some branches. Note that as of the time this was written, this command also shows all pull requests which have a “pr” in their directory tree.

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

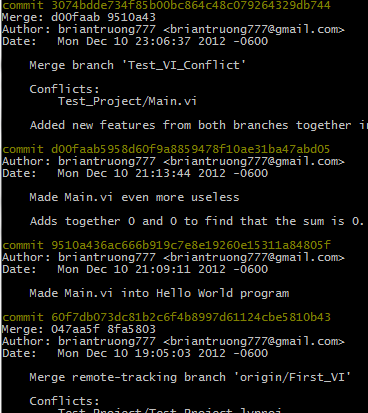


The output of the status command shows your current branch, files to be committed, files not staged for commit (will not be committed), and untracked files (files that were never committed before). It also gives you common commands for common actions. **Use the status command whenever you have no idea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

The fetch command

git fetch

This command is half of what the Github for Window’s sync button does. The fetch command only downloads information from the remote repo. It does not change any local branches. It does give you up-to-date copies of the repo’s branch which is accessible by the branch name “origin/<your\_branch>”. Running a git status after fetching will show you whether your local branch is still up-to-date.

The pull command

git pull

This is just a fetch followed by a merge command that merges the remote repo’s branch into your local copy of the branch. If the merge fails, you will automatically enter merge conflict mode.

The push command

git push

This essentially uploads all commits made to the local branch to its equivalent remote branch on the github servers. This operation can fail if the remote branch is ahead of your local branch (someone changed it first). In such an event, just use a fetch, and then merge from the remote branch to yours (which together is equivalent to a pull command).

The fetch, pull, and push commands can be used instead of the Github for Window’s sync button at all times. It may be preferable to do so to avoid confusion as to what exactly Github for Windows is doing.

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 repository.

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>

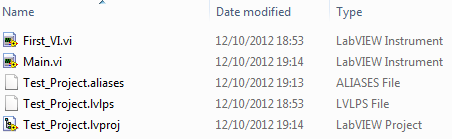
The <remote\_repo> will probably be “origin”. Only pushes new tags to server.

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. **All three files are text files.** You can open them in notepad or vim like any other text file.

\*.aliases files:

These files store the ip addresses of devices that are registered in the project. The computer’s ip address will change when the project is opened on different computers, but the cRio’s ip address should always be the same and matching the cRio’s actual ip address. Generally, throwing away changes to this file is perfectly fine as well as keeping changes since Labview changes this file to match the computer frequently.

\*.lvlps files:

These files seem to only store the position of the project explorer window on the screen. Generally can always throw away changes to this file. Keeping changes will only cause minor inconvenience when project is later opened on a different computer.

\*.lproj files:

These files are xml files that maintain 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. Also, if any dependencies change, this file will be altered too. **If two different Git branches modify this file (which can easily happen), you will have to manually merge the changes to this file.**

Although all three are text files and thus editable in notepad, only the \*.lvproj file is an xml file. 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. Also, that was a lie.

Xml file structure

The first line of an xml file declares the formatting, version, etc. and shouldn’t be touched.

In xml files, each item consists of a beginning tag and an end tag.

<item>

</item>

A file can only have one item, but that item can have infinite amounts of inner-items. Although inner-items don’t have to be indented (or even on different lines), they typically are for readability.

<item\_a>

<item\_b></item\_b>

</item\_a>

An item can either have a string inside it, or it can have multiple inner-items (not both).

<parent>

<child1>Hello there!

This string spans multiple lines which can happen.</child1>

<child2>

<grandchild></grandchild>

</child2>

</parent>

Notice that indentation and line breaks don’t matter for the tags, but you should still keep it readable. Notice the string inside an item will maintain any indents and line breaks.

Notice that item “grandchild” contains nothing. Such empty items can be shortened to the following form.

<grandchild/>

Each item can have multiple properties. Properties are name-value pairs where the value is a string. Properties are separated by spaces. For example, the following item\_c has two properties.

<item\_c Name=”Item C” Type=”Example Item”/>

Properties can be used with the long form as well.

<item\_c Name=”Item C” Type=”Example Item”>

<item\_d Name=”None”>This is item\_d</item\_d>

</item\_c>

Notice the end tag doesn’t list the properties.

This should be enough information to understand xml files. I hope.

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 each 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.

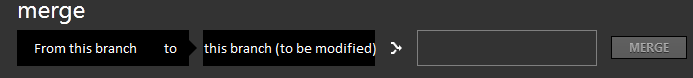
**Obviously, try not to have multiple branches change the same files.** Otherwise, you will have to manually merge them too.

Merging using the command line to address 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 for further help.

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.

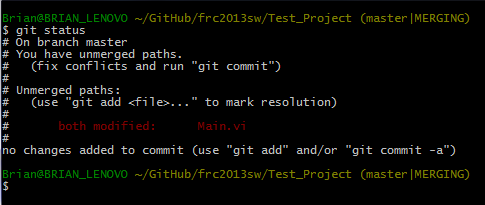
When you run the status command, all files to be committed should be displayed in green. Those that are suffering from a merge conflict should be in red. **You must now manually merge by editing the conflicting files and then adding them into the merging commit.**

Addressing merge conflicts manually

Uh oh. I’m going to be honest. This isn’t going to be easy. However, there are tools available to help. The usage of those tools is described in a later section. Regardless, the general steps of addressing merge conflicts don’t change. 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. 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> <new\_file>



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

git checkout <source\_branch> -- <old\_file>



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, add changes to the file you can open through the Labview project window (in this example, the one named Main.vi). You can also just add the changes to one file, delete the other file, and finally rename the remaining file to the correct name.
   1. Details on how to merge changes for particular types of files can be found in a later section about 1-2 pages down.

Adding a file into merging commit

git add <file>

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. Also know that this command only adds the current version of the file. **If the file is later changed, you will need to add it again.**

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

git rm --cached <file>

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?

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!

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

Whatever you do, don’t forget to remove the markers Git has added. The markers are added to text files that are in conflict, so they won’t be in files without conflicts. The markers that git adds are structured as follows:

~~~

<<<<<<< HEAD

changes from current branch

=======

changes from source branch

>>>>>>> source\_branch

~~~

The ~~~ represents the rest of the file above and below this section. There can be multiple markers like this in a single file. Use ctrl-f to find these markers (search for =======, <<<<<<<, or >>>>>>). The term HEAD represents the current branch.

In all cases, the markers must be removed before Labview can open the project. For this reason, **you must remove these markers first before you can open the project or any \*.vi file.** Specifically remove the lines that start with either <<<<<<<, >>>>>>>, or =======. You should remove the entire line (do not leave a blank line in its place).

For \*.aliases and \*.lvlps files, I wouldn’t recommend actually merging them. Just keep the changes from one branch and remove the markers.

For \*.lvproj files, know the basics of xml from the “Typical Labview Project Structure” section near the beginning (or look up xml files yourself). The \*.lvproj file basically records every file added to the project. It also keeps track of virtual folders (folders only in the project explorer window) and file dependencies. The dependency files are very likely to cause merge conflicts.

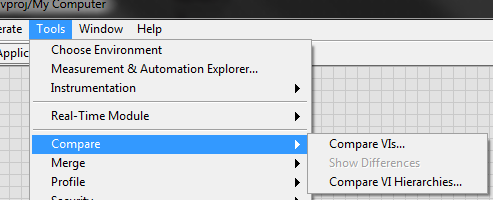
The best way to fix such merge conflicts is to add any new files from both branches and making sure the final version has no duplicates. In other words, try not to lose any files. It’s easy to remove unneeded files from the project, but finding the missing file out of hundreds can be difficult. Plus, removing files willy-nilly will probably result in missing dependencies, but having a few extras won’t cause problems.

**I also recommend opening the project in Labview after finishing to confirm everything is fine and to let Labview fix any of your screw-ups.**

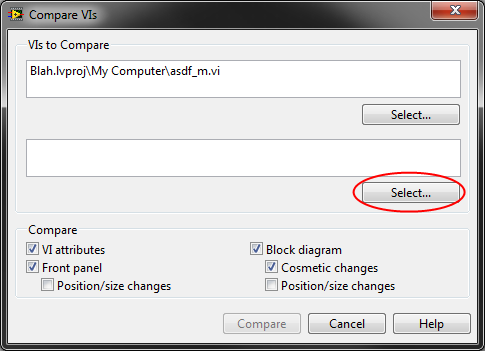
Using the Labview diff tool

This tool is invaluable for fixing merge conflicts since it clearly lists every single difference between two vi’s.

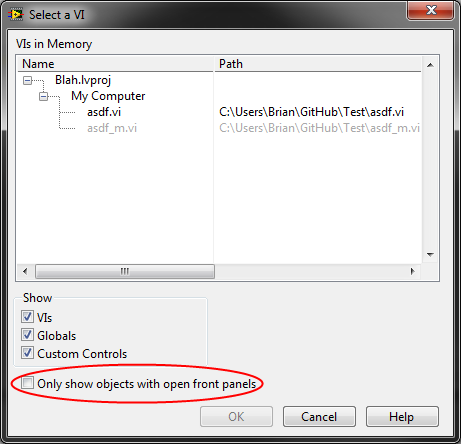
1. Open the project
2. Open the two vi’s in question
3. In either of the two vi’s, go to Tools->Compare->Compare VIs…



1. Click on the second “Select…” button

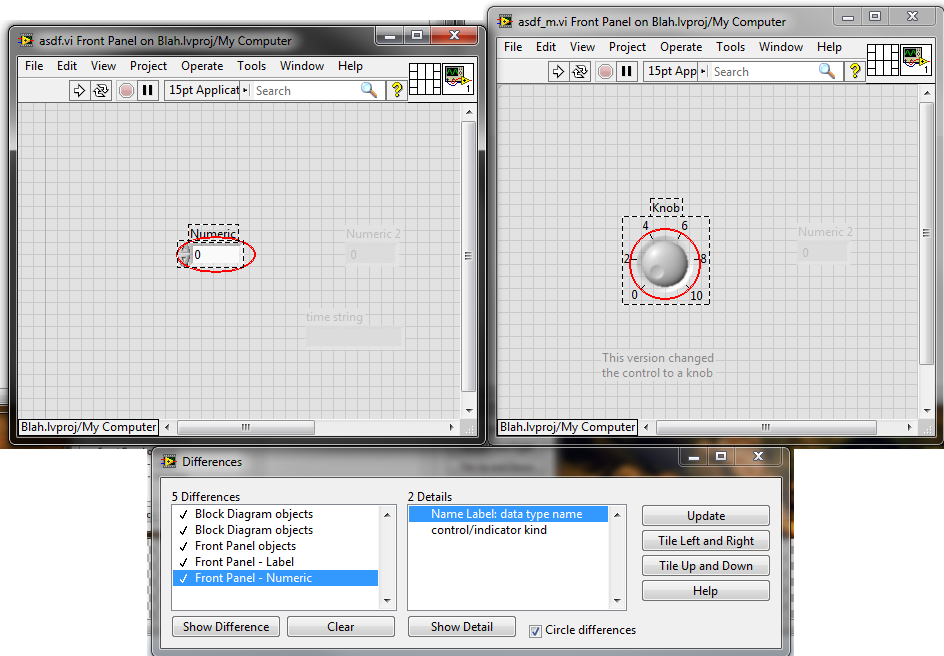


1. Checkbox the “Only show objects with open front panels”



1. Choose the correct vi
2. Click on the “Compare” button

You will now see the two vi’s opened side-by-side with a smaller window indicating all the differences between the two. Double-clicking on a difference will cause Labview to highlight the difference in both vi’s.



You can now make changes in either file which will be saved to the respective file. The list of differences is only recalculated when you click the update button, so don’t worry about losing the list while making changes. You only need to change one of them. Just make sure you later rename the vi with all the changes to its correct name.

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

git 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. You can use this format: “origin/<branch\_name>” for <commit> to overwrite the local repo with the commit/branch from the remote repo.

You can also use other versions of reset for less dangerous purposes. Instead of using the “--hard” option, look up the other options and see what they do.

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.

For the push, you will need to use the force option.

git push -f

The force option means your local branch will completely replace the remote repo’s equivalent branch. You will lose any new commits in the remote branch.

Use this command whenever you need to violently shove a kicking and screaming branch into its rightful place beneath the ground. Don’t forget the gravestone. No but seriously, this is the actual command that changes history. Use your time machine wisely.