1. 1. [First time](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#First_time)
2. 2. [Getting the source](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Getting_the_source)
3. 3. [Best practices](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Best_practices)
   1. 3.1. [Working with master branch only (simple method)](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Working_with_master_branch_only_(simple_method))
   2. 3.2. [Conflicts](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Conflicts)
   3. 3.3. [Working with branches](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Working_with_branches)
   4. 3.4. [Deleting/reverting commits](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Deleting.2Freverting_commits)
      1. 3.4.1. [Local commit (not yet pushed)](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Local_commit_(not_yet_pushed))
      2. 3.4.2. [Pushed commit](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Pushed_commit)
   5. 3.5. [Do not rebase merge-commits](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Do_not_rebase_merge-commits)
   6. 3.6. [Finding commits from old CVS branches](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Finding_commits_from_old_CVS_branches)
4. 4. [Project branches](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Project_branches)
   1. 4.1. [Create the project branch](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Create_the_project_branch)
   2. 4.2. [Merge changes from master (or whatever branch you merged from)](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Merge_changes_from_master_(or_whatever_branch_you_merged_from))
   3. 4.3. [Merging changes to master (or whatever branch you merged from)](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Merging_changes_to_master_(or_whatever_branch_you_merged_from))
   4. 4.4. [Building](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Building)
5. 5. [Maintenance (stable) branches](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Maintenance_(stable)_branches)
   1. 5.1. [Creating a maintenance branch](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Creating_a_maintenance_branch)
   2. 5.2. [Cherry-picking changes](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Cherry-picking_changes)
6. 6. [Vendor branches](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches)
   1. 6.1. [Creating a vendor branch](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Creating_a_vendor_branch)
   2. 6.2. [Updating a vendor branch](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Updating_a_vendor_branch)
   3. 6.3. [Things to consider for vendor branches](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Things_to_consider_for_vendor_branches)
7. 7. [Locking a branch for future updates](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Locking_a_branch_for_future_updates)
8. 8. [Branching iBOS](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Branching_iBOS)
   1. 8.1. [BECS and verification libraries](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#BECS_and_verification_libraries)
   2. 8.2. [Submodules: ox and libtransfer](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Submodules:_ox_and_libtransfer)
      1. 8.2.1. [Making a change on a submodule](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Making_a_change_on_a_submodule)
9. 9. [Restoring a locally deleted file](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Restoring_a_locally_deleted_file)
10. 10. [Adding default LD\_LIBRARY\_PATH on Solaris](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Adding_default_LD_LIBRARY_PATH_on_Solaris)
11. 11. [Converting CVS modules to git repositories](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Converting_CVS_modules_to_git_repositories)
12. 12. [Adding a repository to Waystream master git sites](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Adding_a_repository_to_Waystream_master_git_sites)
13. 13. [Syncing master repositories between Kista and Shanghai](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Syncing_master_repositories_between_Kista_and_Shanghai)
14. 14. [Miscellaneous](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Miscellaneous)

At Waystream we use Git (<http://git-scm.com/>) to manage our source code.

(Please look at documentation in S:\reference\vendors\git\ (or [\\dimsum\Documents\NCS\Git](file:///\\dimsum\Documents\NCS\Git) if you are in SH) for information on how to use git.)

First time

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=1#Vendor_branches)

When using git on a new computer/system you must first set your name and e-mail address.

To see if it already set use:

git config --list

To set it use:

git config --global user.name "Your Name"

git config --global user.email "younam@waystream.com"

**Note:** in the e-mail address we use the short name (i.e. the unix login name) since the full name is present in user.name. It should be the same as your bugzilla account name, if not you will not by default be able to commit log messages into bugzilla bugs.

It is very important and useful to config push.default to nothing:

git config --global push.default nothing

**Note:** With this configured, you must specify which branch to push, this protect you from accidently pushing all your mathing branches to the remote server.

Getting the source

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=2#Vendor_branches)

To get the source (in git called cloning):

git clone ssh://git.int.waystream.com/git/<repository>

or if you are working on the same machine as we have the central git repositories (e.g. the machine bianyi in shanghai):

git clone /git/<repository>

You can see which repositories we have by going to [http://git.int.waystream.com/](http://git.int.packetfront.com/). Here are some examples:

git clone ssh://git.int.waystream.com/git/becs

git clone ssh://git.int.waystream.com/git/ibos

git clone ssh://git.int.waystream.com/git/systemtest

Best practices

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=3#Vendor_branches)

Working with master branch only (simple method)

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=4#Vendor_branches)

1. Make a clone of the repository as described above. After the clone the master branch will be checked out by default.
2. If some time as passed since you cloned or last updated you should first pull in any changes from the central repository:
   1. If you have any uncommited changes either commit them first (see below) or do a: git stash.
   2. Do a: git pull -–rebase.
   3. If you get conflicts you have to solve them (see below).
   4. If you did a git stash in the step above now do a: git stash apply. [1)](http://inside.int.packetfront.com/joomla/index.php?option=com_jd-wiki&Itemid=106&id=rd-devel:git" \l "fn1" \o "http://inside.int.packetfront.com/joomla/index.php?option=com_jd-wiki&Itemid=106&id=rd-devel:git#fn1" \t "_blank)
3. Make your changes (add/modify/remove files and/or directorires).
4. Use git gui to commit your changes locally:
   1. Start it with: git gui
   2. Select files you want to include in the commit by pressing the icon to left of the file name in the list “Unstaged Changes”.
   3. You can also select to commit only some changes in a file by right-clicking on the changed line/hunk in the diff view.
   4. If you change your mind and don’t want to commit some file you press its icon in the list “Staged Changes”.
   5. Inspect the diff’s carefully so you don’t commit something unintentionally.
   6. Enter a commit message and press “Commit”.
5. If you want to change the last commit (either what was commited or the commit message) you can do that as long as you haven’t already pushed it to the central (or any other) repository:
   1. Make any changes to any files you want.
   2. Start git gui (or press “Rescan” if it is already open).
   3. Select the “Amend Last Commit” radio button to the right.
   4. Select any files you want in the “Unstaged Changes” (or deselect files in the “Staged Changes” list).
   5. Change the commit message if you want.
   6. Press “Commit”.
6. When you want send your commits to the central repository:
   1. git push origin master
   2. If you get an error similar to this: ! [rejected] master → master (non-fast forward) . This means someone else has pushed a commit since you last did a pull. Solve this by first doing a git pull –rebase (if you get conflicts see below). Then repeat the git push origin master.

Conflicts

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=5#Vendor_branches)

If you get any conflicts when doing a git pull -–rebase

1. Edit the files that have the conflicts and solve the conflicts. If you forget what files had conflicts you can do a: git status
2. Once the conflicts have been solved do git add <file> on each file that had a conflict.
3. Run git rebase –continue

Working with branches

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=6#Vendor_branches)

If you are to do work in a public branch you should almost always create a small topic branch to do your work in. This is to keep your changes isolated as long as possible and also gives you a chance to clean up before publishing your changes. And before you publish your changes you should rebase your topic branch based on the branch point (this will make the history more clean). So the following is a recommended workflow:

1. git checkout -b topic1 master - Make, and checkout, a topic1 branch based on current master.
2. Make and commit changes in topic branch
3. Before merging your changes upwards update source branch and rebase:
   1. git checkout master
   2. git pull or git fetch and git merge origin/master if you want more control
   3. git checkout topic1
   4. git rebase master
4. Now you have a clean history in topic1 branch and can merge it back
   1. git checkout master
   2. git merge topic1 - This should be a simple fast forward merge
5. And push your changes - git push origin master

Step 3 can be done now and then to keep your topic branch updated to changes in upstream.

You can also create your topic branch as a remote tracking branch, then you don’t have to go via your local copy of the master branch when pulling/pushing:

1. git fetch origin (to get the latest and greatest)
2. git checkout -b topic1 origin/master
3. Add/modify/delete files and/or directories.
4. Commit using git gui (or git add and git commit).
5. git push origin topic1:master
6. If you get a non-fast forward error you first have to do a git pull –rebase and then redo the push.

Deleting/reverting commits

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=7#Vendor_branches)

If you have made a commit that you want remove there are a number of ways depending on the situation.

Local commit (not yet pushed)

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=8#Vendor_branches)

If you want to remove the latest commit on your local branch (that has not yet been pushed to a central shared repository) you can simply use git reset:

git reset --hard HEAD^

HEAD^ points the prevoius commit so this commit will simply move your local branch head to point to that commitIf you want to remove the two latest commits you can use HEAD^^ instead. You can also specify any commit <SHA> and then you will remove any commits after that. The --hard means that it will update the checked out files to match that commit.

If you for example have made 5 local commits that you have not pushed and you want to delete one or more that is not the latest commit you can use the very powerful command of interactive rebase:

git rebase -i HEAD~5

Then it will open an editor containing one line per commit and if for example you wanted to delete commit 2 and 4 you simply remove those lines, save and exit the editor.

If you want to remove just parts of the last commit you can also use git gui and press the "Amend Last Commit" radio button. You can then remove changes to whole files by clicking on the file in the Staged Changes area (they will then be moved up to the Unstaged Changes area). You can also unstage part of changes to a file by selecting the file in the Staged Changes area and then right-clicking on the lines you want to unstage.

Pushed commit

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=9#Vendor_branches)

If you want to remove a commit that has already been pushed to a central shared repository you cannot use any of the methods described above. This is because with the methods above you will actually change the history which will not work on a shared repository since other people might have continued working on the branch in the old history. However, even if the method described here will work on non-pushed commits also the methods described above are prefered since it creates a cleaner history.

So for pushed commits you will instead have to create a new commit that undo the changes of the commit you want to remove.The simplest way to do this is using git revert:

1. git revert <sha>
2. Compile and test
3. git push origin <branch>

Note: when you do git revert you will automatically get a commint message saying 'Reverting "<original commit message>"'. This should be kept but you should also add a line or two describing why you removed the commit.

Do not rebase merge-commits

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=10#Vendor_branches)

If you rebase a merge-commmit it will get ”translated” into a single commit that no longer has the vendor branch as an ancestor (and this is usually not what you want).

I.e. if you have:

 master-branch

 ^

 |     ^

 X     A (fix merge stuff)

 |     |

 Y     M (merge of ipinfusion into master)

 |     |

 Z--—-/ \

 |      |

 W      ^-ipinfusion-branch

 |      |

 P      I (import of 7.10.3)

 |      |

Explanation: you have imported new version 7.10.3 into ipfinfusion branch and then at point Z on master you merge ipinfusion and get commit M. Then you do a commit A with some more fixes. While doing this someone else has committed Y and X to master.

At this point you \*cannot\* rebase M and A on master because then you get:

 master-branch

 ^

 A' (fix merge stuff)

 |

 M' (merge of ipinfusion into master)

 |

 X

 |

 Y

 |

 Z

 |

 W      ^-ipinfusion-branch

 |      |

 P      I (import of 7.10.3)

 |      |

The content of M' and A' will be correct (i.e. you will have the new 7.10.3 version in master), but it will be detached from the vendor branch. So the next time you try to import a new version from the vendor you will be in trouble.

So if someone has done changes to master while you did your merge so you can’t push it directly you have two options:

a) Simply merge master with with your new merge branch and then push it backs you get:

  master-branch

    ^

    |

    B (merge master and merge-branch)

   / \

  /   \

 X     A (fix merge stuff)

 |     |

 Y     M (merge of ipinfusion into master)

 |     |

 Z--—-/ \

 |      |

 W      ^-ipinfusion-branch

 |      |

 P      I (import of 7.10.3)

 |      |

b) Or if you really want to avoid this ”bubble” in the history you have to: first redo the merge onto X and then cherry-pick A. But this is a lot more work since you have to resolve all the old conflicts again plus any new that might have come up. You will then get:

 master-branch

 ^

 |

 A' (fix merge stuff)

 |

 M2 (merge of ipinfusion into master)

 |\

 X \

 |  \

 Y   ^-ipinfusion-branch

 |   |

 Z   I (import of 7.10.3)

 |   |

 W

 |

 P

 |

Note: these things can be very tricky, so I strongly recommend that you always always run gitk before and after you do a merge or rebase to se that you get what you expected.

Finding commits from old CVS branches

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=11#Vendor_branches)

The **git blame <file>** is a very useful command see when a specific line in a file changed (so you can look at the commit message and maybe understand better why it was changed). However, for repositories that have been converted from CVS (see section below) there is a a catch. Branches merged in CVS will only show up as a single commit in GIT because the conversion from CVS to GIT was not perfect. So for example in ibos when you do git blame on files you might find that many lines comes from this commit:

commit b464bbcac56d778c5bf84cda41a510988bc55ef3  
Author: Erik Anggard <[eriang@packetfront.com](mailto:eriang@packetfront.com)>  
Date:   Tue Apr 17 02:13:37 2007 +0000  
  
  
    \* Merge of PHOENIX into HEAD.

This was when we merged the code of PHOENIX project (the ASR5k project). So this is not so informative. What you do if you get to this commit and want to find a more specific commit is this: **git blame IBOS\_PHOENIX\_BRANCH <file>**

Some times you will have to do this several times, i.e. a commit on IBOS\_PHOENIX\_BRANCH might come from another merge, e.g: RT1\_BRANCH, then you have to do **git blame IBOS\_RT1\_BRANCH <file>**to continue the search for the commit.

Project branches

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=12#Vendor_branches)

Projects that needs their development separated from the master branch should create a project branch that is kept up to date with master (to avoid extra testing), and finally, when the project is ready for a release, pushes the project changes to master.

Create the project branch

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=13#Vendor_branches)

The BRANCH\_NAME should be SITE.PROJNAME[-FEATURE], e.g ki.aperto-macff

1. git branch BRANCH\_NAME <branch point> or omit <branch point> if you want to branch from your current branch.
2. git config branch.BRANCH\_NAME.remote origin
3. git config branch.BRANCH\_NAME.merge refs/heads/BRANCH\_NAME

(An alternativ way of doing this:

1. git branch BRANCH\_NAME <branch point> or omit <branch point> if you want to branch from your current branch.
2. git push origin BRANCH\_NAME
3. git branch -d BRANCH\_NAME
4. git branch BRANCH\_NAME origin/BRANCH\_NAME

this is one step more but easier to remember.)

Note: If you are getting any error related to push like read only repository then please check your push url in .gitconfig and make sure it points to proper url.

Merge changes from master (or whatever branch you merged from)

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=14#Vendor_branches)

To merge changes that happened in master to your project branch:

1. Stand in your project branch
2. git merge master
3. git push origin <project branch>
4. Resolve eventual conflicts.

This should be done now and then to keep the project branch up to date.

Merging changes to master (or whatever branch you merged from)

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=15#Vendor_branches)

To merge changes in project branch to master:

1. Make sure both master and local project branch is up to date:
   1. git checkout <project-branch>
   2. git pull –rebase
   3. git checkout master
   4. git pull –rebase
2. Stand in master branch.
3. git merge <project branch>
4. Solve any conflicts you might get (edit files, do git add and git commit).
5. Start gitk and check that it looks ok.
6. git push origin master
7. If you get a non-fast-forward error \*do not\* rebase the merge! Instead do git reset –hard HEAD^ to remove the merge and restart with step 1 above.

An alternative way is to do this:

1. git fetch origin
2. git checkout -b tmp\_merge origin/master
3. git merge origin/<project-branch>
4. (solve any conflicts, git add, git commit)
5. git push origin tmp\_merge:master
6. git branch -d tmp\_merge

(If you get non-fast-forward error in step 5, start over from step 1).

Building

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=16#Vendor_branches)

To build a nightly **iBOS** image based on the project branch, do (as builder@linker in tagbuild/):

./ibos\_tag\_compile.sh -d -b BRANCH\_NAME -n

To build a nightly **BECS** based on the project branch, do (as builder@ridley in tagbiuld/):

./tag\_compile\_becs.sh -b BRANCH\_NAME -n

Maintenance (stable) branches

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=17#Vendor_branches)

Maintenance branches (aka stable branches) are like project branches except you don't merge changes from master and you will never merge them back into master. Let's say for example that you are working on the master branch to make the first 1.4.x release. When you have built the first 1.4.1-R (or earlier if you need to start working on features for 1.5.x in parallell) you will branch of a 1.4 branch. We do this so that we can easily control what will go into a maintenance release, e.g. when we later want to release a 1.4.2 we only want selected bug fixes to go into that release and not features developed for 1.5.x.

Creating a maintenance branch

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=18#Vendor_branches)

See "Create a project branch" above.

Cherry-picking changes

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=19#Vendor_branches)

Since we do not merge master into a maintenance branch or vice versa we need some other to copy changes (commits) from one branch to another, this is called cherry-picking. Here is an example of how you cherry-pick a commit from master to 1.4 branch:

1. git log origin/master (find SHA of commit you want to cherry-pick)
2. git checkout 1.4
3. git cherry-pick -x <sha>
4. Solve conflicts if needed, compile and test
5. git push origin 1.4

Some notes:

* Use -x when cherry-picking this will add a line to the commit message saying which commit it was cherry-picked from, e.g. "cherry-picked from commit 0d9370f51f5705cdcd1d4f5a970719eba3b2e19e)"
* Always commit bug fixes to master first and then cherry-pick it into the maintenance branch(es) where it is needed. (The reason for this is that if you commit to the maintenance branch first and then forget to commit it to master we will get regression which customers usualy strongly dislike.)

Vendor branches

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=20#Vendor_branches)

Vendor branches are specific branches where you keep code that you have imported from some other vendor (or open source project). Git does not have a specific concept of vendor branches like for example CVS does (actually, CVS handling of vendor branches can be quite confusing and error-prone). In git a vendor branch is just like any other branch, it is just the content that differs. But it is still a very good idea to use vendor branches with git for a number of reasons:

* It makes upgrading the vendor software much easier.

When you receive a new version of the vendor software you will do these steps:

1. Check out the vendor branch
2. Remove all old files from the previous version
3. untar/unzip/copy the files from the new version
4. Make a commit on the vendor branch saying 'Imported version x.y.z of software xxx'
5. Push the commit to the central repository
6. Checkout master (or what ever branch you want the new version to appear in)
7. Do 'git merge <vendor-branch>'
8. Solve any conflicts and commit the merge
9. Compile and test
10. Push to the central repo

In step 7 when you do the merge git will know that it only has to apply changes since the last merge of the vendor branch. So if you haven't modified the same code lines as those being updated the merge will not give any conflicts.

* It makes it easy to see what parts of the code has been added/changed by the vendor and which has been changed by us. When doing git log or git blame code added/changed by the vendor will be in commits with messages 'Imported version ...' (and you can see that they where made on the vendor branch). Code added/modified by us will only appear on the master (or other none-vendor branches).
* It makes it easy to see what the vendor has changed between between two versions. (You can simply do a git diff between two commits on the vendor branch).
* It makes it easier to estimate how much work an upgrade will be. You can very easily do step 1-4 and 6-7 above and see how many conflicts you get (and take quick look at the type of conflicts). You will then have a good idea of the amount of work needed. After that you can reset your local vendor branch and master branch if you decide not to continue with the upgrade.

Creating a vendor branch

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=21#Vendor_branches)

The first time you create a vendor branch you need to take some steps to get a clean history on the vendor branch. In the example below I will use a project call 'proj' and vendor software called 'foo' from vendor 'bar' and we will import the code into a subdirectory dist/bar in the project.

1. Use an existing clone or create a cloned repository of your project: git clone <ssh://git.int.waystream.com/git/proj>
2. cd proj
3. git symbolic-ref HEAD refs/heads/bar (this will create a new empty branch called bar)
4. rm .git/index
5. git clean -fdx
6. git commit --allow-empty -m "Created Bar vendor branch"
7. Merge the empty vendor branch into master but keep master untouched:  git checkout master and then  git merge -s ours bar
8. Extract vendor code on the vendor branch: git checkout bar ,  mkdir dist, cd dist, tar -xzf foo-1.0.1.tar.gz and mv foo-1.0.1 foo
9. Add and commit: git add dist/foo and git commit -m "Imported foo version 1.0.1"
10. Merge code into master: git checkout master and git merge bar (you should not get any conflicts now since the vendor code is in its own subdirectory under dist/)
11. Push bar and master: git push origin bar:bar and git push origin master:master

After this has been done you can start doing changes in the vendor code in the master branch (or some other branch that is not the vendor branch) if needed.

Updating a vendor branch

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=22#Vendor_branches)

After some time you might receive a new version of the vendor code, this is what you do (following the example from above):

1. cd proj
2. git checkout bar
3. rm -rf dist/foo
4. cd dist && tar -xzf foo-1.0.2.tar.gz and mv foo-1.0.2 foo
5. git add -A foo and git commit -m "Imported foo 1.0.2"
6. git checkout master and git merge bar
7. Solve any conflicts and commit the merge
8. Compile and test
9. Push bar and master: git push origin bar:bar and git push origin master:master

Things to consider for vendor branches

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=23#Vendor_branches)

* The vendor code may be placed in multiple directories or have files directly in the root of the project but it is often easier if it has one single directory (like the foo directory in the example above).
* It is important the only code changes from the vendor goes into to the vendor branch and that no changes from the vendor goes directly into master (or other branches). Otherwise you will get more conflicts and it will be harder to keep track of changes.
* It is easiest if you include all the code from the vendor on the vendor branch. However, if the vendor code contains a lot of files and/or directories that you don't want in your project you might want to clean those out before the first import. But if you do that you will have to clean them out on each subsequent upgrade import. And you should never add or modify files before you import.

Locking a branch for future updates

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=24#Vendor_branches)

Once you have done a final merge of a branch into another branch you don't want people to continue submitting changes to the old branch. E.g. if you have a done feature development on a feature branch that is then merged into master branch you then want any future bug fix to to be done directly on the master branch. Sometimes you also want to change name on a branch without removing the old branch name. In these cases you want to lock the old branch from further commits. You have to do this in both gerrit (prevent submits) and git-kista (prevent direct push to the old branch). Here is how it is locked in git:

1. ssh to git-kista
2. cd /git/<repository> (e.g. /git/ibos)
3. echo 'Branch XXX has should no longer be used, use branch YYY instead' > locked/refs/heads/XXX  
   (where XXX is the name of the branch to lock)
4. make sure the update hook is installed:

$ ls -la hooks/update

lrwxrwxrwx 1 builder pf 30 Jun  1 05:47 hooks/update -> /git/hooks/update-commit-check  
(if it is not then create this symlink)

Here is a link to how to lock it in gerrit: [Locking a branch in gerrit](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Gerrit#Locking_a_branch_in_gerrit)

Branching iBOS

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=25#Vendor_branches)

(Previously you had to branch becs, libtransfer and ox when you branch ibos. As of commit 05c77bc0f6212c389703ad597acce6c3c1c45182 (2008-12-15) you no longer need to do this. Here is a link [old\_branching\_ibos](http://inside.int.packetfront.com/joomla/index.php?option=com_jd-wiki&Itemid=106&id=rd-devel:old_branching_ibos" \o "rd-devel:old_branching_ibos" \t "_blank) to the text that described how you did that.)

iBOS is branched just like any other git repository. Local branches that you will not push to one of the central common repositories can have any name. Project and stable branches that will be shared by several developers and therefor will be pushed to one of the central repositories must be prefixed with “ki.” or “sh.” depending on if the main development is in Kista or Shanghai. E.g: ki.rainbow (branch for the project rainbow running in Kista) sh.ibos-3.17-18 (branch for iBOS version 3.17-18 maintained in Shanghai)

BECS and verification libraries

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=26#Vendor_branches)

When building the command verification libraries the build script needs to checkout BECS. It will checkout the version of the commit ID (SHA) stored in ibos/ibos.ox/BECS\_VERSION. This version should every now and then be updated in the master branch of ibos to match the head of the becs master branch. Normally BECS\_VERSION will always point to some commit on the becs master branch. If however you need to make a change to the becs code (needed for a branch in ibos) and you don’t want to include some other changes that have been included on becs master you might have to make a branch in BECS. E.g. if becs master branch looks like this:

A -> B -> C -> D (master)

And BECS\_VERSION points to commit B and you want to commit a change X to becs. Normally you would just commit it to the head of the becs master and point BECS\_VERSION there like this:

A -> B -> C -> D -> X (master)

But if you don’t want changes made in C and D you will have to create a new branch in becs (call it the same as the ibos branch or ibos-stableN):

A -> B -> C -> D (master)

|

\--> X (ibos-stable1)

Note: the reason we don’t use the git submodule mechanism (see below) for becs is that we don’t want to checkout the complete becs code everytime we checkout ibos.

Submodules: ox and libtransfer

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=27#Vendor_branches)

iBOS also has two submodules (see [git-submodule(1)](http://www.kernel.org/pub/software/scm/git/docs/git-submodule.html" \o "http://www.kernel.org/pub/software/scm/git/docs/git-submodule.html)): ox and libtransfer. These are separate git repositories that are checked out when building ibos (ox is checked out as util/ox/ox and libtransfer as ibos.lib/libtransfer/libtransfer). Much like BECS\_VERSION for becs (see above) the submodules are simply pointers to a commit in an external git repository. So the same reasoning about branching also applies to libtransfer and ox as for becs: normally the submodules will point to a commit on the master branch of each submodule but if you need to make a change and don’t want to include other changes on the master branch you will have to make a branch.

Making a change on a submodule

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=28#Vendor_branches)

This is how you make a change to a submodule (libtransfer will be used in this example):

1. Go to your ibos repository: cd ibos
2. Checkout the ibos branch that you want to make the change in: git checkout master
3. Run tmake once (this will update the submodules): tmake
4. Goto the submodule: cd ibos.lib/libtransfer/libtransfer
5. If you want to make the change on the master branch of the submodule: git checkout master   
   If you want to make a new branch: git checkout -b ibos-stable3
6. Make any changes to files under the ibos.lib/libtransfer/libtransfer
7. Compile ibos and verify that the changes work
8. Commit the changes in the submodule: cd ibos.lib/libtransfer/libtransfer; git gui (or git commit ...)
9. Check the changes: cd ibos.lib/libtransfer && git diff   
   It will show something like:
10. diff --git a/ibos.lib/libtransfer/libtransfer b/ibos.lib/libtransfer/libtransfer
11. index fd62bae..4d16b43 160000
12. --- a/ibos.lib/libtransfer/libtransfer
13. +++ b/ibos.lib/libtransfer/libtransfer
14. @@ -1 +1 @@
15. -Subproject commit fd62bae922f3e8c22e7aada2a5b379089db0705f
16. +Subproject commit 4d16b439eb5b671fdbe444d809a8507973e3fa53
17. Now go back to the submodule and do a git diff/log to make sure that the changes between the two commits does not contain something you do not want:   
    cd ibos.lib/libtransfer/libtransfer; git log -p fd62bae922f3e8c22e7aada2a5b379089db0705f 4d16b439eb5b671fdbe444d809a8507973e3fa53   
    (Because when you switched branch above you might have included more commits than you intended.)
18. If it looks ok then do the commit in ibos: cd ibos.lib/libtransfer; git gui (or git commit ...)
19. Push the submodule first: cd ibos.lib/libtransfer/libtransfer; git push origin master   
    Or if you created a new branch above: cd ibos.lib/libtransfer/libtransfer; git push origin ibos-stable3
20. Then push the ibos commit: cd ..; git push origin <ibos-branch>

**Note: when making a change to becs or a submodule it is important to push the changes in becs or the submodule first and then push the change in ibos. Otherwise there will be a period of time where BECS\_VERSION or the submodule commit ID points to a commit that is not yet available in the central repository and if someone else makes a fetch in during that time they will get an error when they try to build ibos.**

Restoring a locally deleted file

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=29#Vendor_branches)

If you happen to delete a file from your filesystem, which is managed by Git, you can restore the file by doing:

git checkout – <filename1> <filename2> <filename3..>

List deleted files by:

git ls-files -d

**CVS:** cvs up -C <filename>

Adding default LD\_LIBRARY\_PATH on Solaris

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=30#Vendor_branches)

This is needed to make git work on Solaris boxen if installed in /usr/local or so.

As root, run this command:

crle -E LD\_LIBRARY\_PATH=/usr/local/lib

Converting CVS modules to git repositories

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=31#Vendor_branches)

The best method to convert a CVS module to a git repository that we have found so far is to use cvs2svn (<http://cvs2svn.tigris.org/>). We have a slightly modified (the original one has some problems with deleted files) under /git/cvs2svn-trunk. We have a wrapper script called import.sh that will setup a configuration file, do the cvs2svn and then the git fast-import. It will also print some suggestions of things that should be done before and after the convertion. This how it is used (in this example the cvs module asrboot3 is converted to a git repository called asrboot):

ssh humle

cd /git/import

./import.sh asrboot3 asrboot

Adding a repository to Waystream master git sites

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=32#Vendor_branches)

We have two master git sites, one in Kista (git-kista.int.waystream.com that points to linker) and one in Shanghai (git-shanghai.int.waystream.com that points to bianyi). The address git.int.waystream.com will point to git-kista or git-shanghai depending on which network you are on. The repositories are mirrored between Kista and Shanghai every 10 minutes (for easier access but also as form of backup). So if you have a repository that has its main site in Kista and is mirrored to Shanghai you can do git clone and fetch/pull from git-shanghai but you should always do git push to git-kista.

When adding a new repository to one of the master git sites there are a number of things that should be done: make the repository shared and writeable for the pf group, install hooks that will update bugzilla and the cvs jabber channel,  setup mirror at the remote site (git-kista or git-shanghai) and optionally add it to our gerrit system (for code reviews).  To make this easier we have added a script called add-repo.sh that asks some questions and then does everything for you.

You shold log onto the server (either git-kista or git-shanghai) that you want to be the main site (where you will push changes) for your new repository. E.g. if you have a repository called myproj on linker and I want git-kista to be the main site you would do this:

ssh git-shanghai

/git/scripts/add-repo.sh ssh://linker/home/joedoe/myproj

You will then get some questions and after that it will install the repository and do all settings:

Repository name [myproj]:

Description for the repository: My very important project

Add to gerrit? [Y/n]:

\*\*\*\*\*\*\*\*

Repository to clone: /tmp/myproj

Repository name: myproj

Description: Manufacturing tests (DRG)

Add to gerrit: yes

Central repository at: git-kista

Mirror repository at: git-shanghai

Continue? [Y/n]:

\* Cloning ssh://linker/home/joedoe/myproj to /git/myproj

Initialized empty Git repository in /git/myproj/

\* Adding description

\* Making group shared

\* Setting up receive hooks

Creating hook repo for myproj

Initialized empty Git repository in /git/hook-repos/myproj/

\* Setting up mirror at git-shanghai

Making mirror of myproj on git-kista and setting up for sync

Initialized empty Git repository in /git/myproj/

\* Adding myproj to gerrit

Permission denied (publickey).

 group\_id |         name          |                description

----------+-----------------------+-------------------------------------------

        1 | Administrators        | Gerrit Site Administrators

        2 | Anonymous Users       | Any user, signed-in or not

        3 | Registered Users      | Any signed-in user

        4 | RD                    |

        5 | RD-NCS                |

        6 | RD-Access             |

        7 | RD-BBE                |

        8 | RD-Devel              |

        9 | RD-CPE                |

       10 | Non-Interactive Users | Users who perform batch actions on Gerrit

       11 | Kista-Consultants     |

       12 | Project Owners        | Any owner of the project

(12 rows)

Enter id of group(s) that should be allowed to approve commits, submit, etc.,

or leave empty (for multiple id's separate them with space).

Group id(s): 4

Adding myproj to gerrit

INSERT 0 1

INSERT 0 1

INSERT 0 1

INSERT 0 1

INSERT 0 1

INSERT 0 1

INSERT 0 1

\* Done

Syncing master repositories between Kista and Shanghai

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=33#Vendor_branches)

If you have used the add-repo.sh script described above your repository will already be mirrored so you can skip this. But if you have added your repositroy manually you will have to do this step.

We have made a script called mirror-setup.sh in /git/scripts. Let say you have done the import of asrboot above and want to setup mirror to Shanghai, then you would to this:

ssh git-shanghai

/git/scripts/mirror-setup.sh asrboot kista

Or if you have added a repo called drgos in Shanghai and want it to be mirrored to Kista then you do:

ssh git-kista

/git/scripts/mirror-setup.sh drgos shanghai

Miscellaneous

[Edit section](http://wiki.int.waystream.com/Departments/RD/Internal_Systems/Git#Vendor_branches?action=edit&sectionId=34#Vendor_branches)

Here is link to a page describing how to move folders between git repositories while keeping the history (for that folder) intact:

[http://st-on-it.blogspot.com/2010/01...tween-git.html](http://st-on-it.blogspot.com/2010/01/how-to-move-folders-between-git.html)