NAME

perlgit - Detailed information about git and the Perl repository

DESCRIPTION

This document provides details on using git to develop Perl. If you are just interested in working on a quick patch, see perlhack first. This document is intended for people who are regular contributors to Perl, including those with write access to the git repository.

CLONING THE REPOSITORY

All of Perl's source code is kept centrally in a Git repository at *github.com*.

You can make a read-only clone of the repository by running:

```
  % git clone git://github.com/Perl/perl5.git perl
```

This uses the git protocol (port 9418).

If you cannot use the git protocol for firewall reasons, you can also clone via http:

```
  % git clone https://github.com/Perl/perl5.git perl
```

WORKING WITH THE REPOSITORY

Once you have changed into the repository directory, you can inspect it. After a clone the repository will contain a single local branch, which will be the current branch as well, as indicated by the asterisk.

```
  % git branch
  * blead
```

Using the -a switch to "branch" will also show the remote tracking branches in the repository:

```
  % git branch -a
  * blead
    origin/HEAD
    origin/blead
  ...
```

The branches that begin with "origin" correspond to the "git remote" that you cloned from (which is named "origin"). Each branch on the remote will be exactly tracked by these branches. You should NEVER do work on these remote tracking branches. You only ever do work in a local branch. Local branches can be configured to automerge (on pull) from a designated remote tracking branch. This is the case with the default branch "blead" which will be configured to merge from the remote tracking branch "origin/blead".

You can see recent commits:

```
  % git log
```

And pull new changes from the repository, and update your local repository (must be clean first)

```
  % git pull
```

Assuming we are on the branch "blead" immediately after a pull, this command would be more or less equivalent to:

```
  % git fetch
  % git merge origin/blead
```

In fact if you want to update your local repository without touching your working directory you do:

```
  % git fetch
```

And if you want to update your remote-tracking branches for all defined remotes simultaneously you can do

```
  % git remote update
```

Neither of these last two commands will update your working directory, however both will update the remote-tracking branches in your repository.

To make a local branch of a remote branch:

```
  % git checkout -b maint-5.10 origin/maint-5.10
```

To switch back to blead:

```
  % git checkout blead
```

Finding out your status

The most common git command you will use will probably be

```
 % git status
```

This command will produce as output a description of the current state of the repository, including modified files and unignored untracked files, and in addition it will show things like what files have been staged for the next commit, and usually some useful information about how to change things. For instance the following:

```
 % git status
 On branch blead
 Your branch is ahead of 'origin/blead' by 1 commit.

 Changes to be committed:
   (use "git reset HEAD <file>..." to unstage)

         modified:   pod/perlgit.pod

 Changes not staged for commit:
   (use "git add <file>..." to update what will be committed)
   (use "git checkout -- <file>..." to discard changes in working
                                                     directory)

         modified:   pod/perlgit.pod

 Untracked files:
   (use "git add <file>..." to include in what will be committed)

         deliberate.untracked
```

This shows that there were changes to this document staged for commit, and that there were further changes in the working directory not yet staged. It also shows that there was an untracked file in the working directory, and as you can see shows how to change all of this. It also shows that there is one commit on the working branch "blead" which has not been pushed to the "origin" remote yet. **NOTE**: This output is also what you see as a template if you do not provide a message to "git commit".

Patch workflow

First, please read perlhack for details on hacking the Perl core. That document covers many details on how to create a good patch.

If you already have a Perl repository, you should ensure that you're on the *blead* branch, and your repository is up to date:

```
  % git checkout blead
  % git pull
```

It’s preferable to patch against the latest blead version, since this is where new development occurs for all changes other than critical bug fixes. Critical bug fix patches should be made against the relevant maint branches, or should be submitted with a note indicating all the branches where the fix should be applied.

Now that we have everything up to date, we need to create a temporary new branch for these changes and switch into it:

```
  % git checkout -b orange
```

which is the short form of

```
  % git branch orange
  % git checkout orange
```

Creating a topic branch makes it easier for the maintainers to rebase or merge back into the master blead for a more linear history. If you don’t work on a topic branch the maintainer has to manually cherry pick your changes onto blead before they can be applied.

That’ll get you scolded on perl5-porters, so don’t do that. Be Awesome.

Then make your changes. For example, if Leon Brocard changes his name to Orange Brocard, we should change his name in the AUTHORS file:

```
  % perl -pi -e 's{Leon Brocard}{Orange Brocard}' AUTHORS
```

You can see what files are changed:

```
  % git status
  On branch orange
  Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)

     modified:   AUTHORS
```

And you can see the changes:

```
% git diff
diff --git a/AUTHORS b/AUTHORS
index 293dd70..722c93e 100644
--- a/AUTHORS
+++ b/AUTHORS
@@ -541,7 +541,7 @@    Lars Hecking              <lhecking@nmrc.ucc.ie>
 Laszlo Molnar                  <laszlo.molnar@eth.ericsson.se>
 Leif Huhn                      <leif@hale.dkstat.com>
 Len Johnson                    <lenjay@ibm.net>
-Leon Brocard                   <acme@astray.com>
+Orange Brocard                 <acme@astray.com>
 Les Peters                     <lpeters@aol.net>
 Lesley Binks                   <lesley.binks@gmail.com>
 Lincoln D. Stein               <lstein@cshl.org>
```

Now commit your change locally:

```
% git commit -a -m 'Rename Leon Brocard to Orange Brocard'
Created commit 6196c1d: Rename Leon Brocard to Orange Brocard
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The "-a" option is used to include all files that git tracks that you have changed. If at this time, you only want to commit some of the files you have worked on, you can omit the "-a" and use the command "git add *FILE* ..." before doing the commit. "git add --interactive" allows you to even just commit portions of files instead of all the changes in them.

The "-m" option is used to specify the commit message. If you omit it, git will open a text editor for you to compose the message interactively. This is useful when the changes are more complex than the sample given here, and, depending on the editor, to know that the first line of the commit message doesn't exceed the 50 character legal maximum. See "Commit message" in perlhack for more information about what makes a good commit message.

Once you've finished writing your commit message and exited your editor, git will write your change to disk and tell you something like this:

```
Created commit daf8e63: explain git status and stuff about remotes
 1 files changed, 83 insertions(+), 3 deletions(-)
```

If you re-run "git status", you should see something like this:

```
 % git status
 On branch orange
 Untracked files:
   (use "git add <file>..." to include in what will be committed)

       deliberate.untracked

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

When in doubt, before you do anything else, check your status and read it carefully, many questions are answered directly by the git status output.

You can examine your last commit with:

```
  % git show HEAD
```

and if you are not happy with either the description or the patch itself you can fix it up by editing the files once more and then issue:

```
  % git commit -a --amend
```

Now, create a fork on GitHub to push your branch to, and add it as a remote if you haven't already, as described in the GitHub documentation at <https://help.github.com/en/articles/working-with-forks>:

```
  % git remote add fork git@github.com:MyUser/perl5.git
```

And push the branch to your fork:

```
  % git push -u fork orange
```

You should now submit a Pull Request (PR) on GitHub from the new branch to blead. For more information, see the GitHub documentation at <https://help.github.com/en/articles/creating-a-pull-request-from-a-fork>.

You can also send patch files to perl5-porters@perl.org <mailto:perl5-porters@perl.org> directly if the patch is not ready to be applied, but intended for discussion.

To create a patch file for all your local changes:

```
  % git format-patch -M blead..
  0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
```

Or for a lot of changes, e.g. from a topic branch:

```
  % git format-patch --stdout -M blead.. > topic-branch-changes.patch
```

If you want to delete your temporary branch, you may do so with:

```
 % git checkout blead
 % git branch -d orange
 error: The branch 'orange' is not an ancestor of your current HEAD.
 If you are sure you want to delete it, run 'git branch -D orange'.
 % git branch -D orange
 Deleted branch orange.
```

A note on derived files

Be aware that many files in the distribution are derivative--avoid patching them, because git won't see the changes to them, and the build process will overwrite them. Patch the originals instead. Most utilities (like perldoc) are in this category, i.e. patch *utils/perldoc.PL* rather than *utils/perldoc*. Similarly, don't create patches for files under *$src_root/ext* from their copies found in *$install_root/lib*. If you are unsure about the proper location of a file that may have gotten copied while building the source distribution, consult the *MANIFEST*.

Cleaning a working directory

The command "git clean" can with varying arguments be used as a replacement for "make clean".

To reset your working directory to a pristine condition you can do:

```
  % git clean -dxf
```

However, be aware this will delete ALL untracked content. You can use

```
  % git clean -Xf
```

to remove all ignored untracked files, such as build and test byproduct, but leave any manually created files alone.

If you only want to cancel some uncommitted edits, you can use "git checkout" and give it a list of files to be reverted, or "git checkout -f" to revert them all.

If you want to cancel one or several commits, you can use "git reset".

Bisecting

"git" provides a built-in way to determine which commit should be blamed for introducing a given bug. "git bisect" performs a binary search of history to locate the first failing commit. It is fast, powerful and flexible, but requires some setup and to automate the process an auxiliary shell script is needed.

The core provides a wrapper program, *Porting/bisect.pl*, which attempts to simplify as much as possible, making bisecting as simple as running a Perl one-liner. For example, if you want to know when this became an error:

```
perl -e 'my $a := 2'
```

you simply run this:

```
.../Porting/bisect.pl -e 'my $a := 2;'
```

Using *Porting/bisect.pl*, with one command (and no other files) it's easy to find out

- Which commit caused this example code to break?
- Which commit caused this example code to start working?
- Which commit added the first file to match this regex?
- Which commit removed the last file to match this regex?

usually without needing to know which versions of perl to use as start and end revisions, as *Porting/bisect.pl* automatically searches to find the earliest stable version for which the test case passes. Run "Porting/bisect.pl --help" for the full documentation, including how to set the "Configure" and build time options.

If you require more flexibility than *Porting/bisect.pl* has to offer, you'll need to run "git bisect" yourself. It's most useful to use "git bisect run" to automate the building and testing of perl revisions. For this you'll need a shell script for "git" to call to test a particular revision. An example script is *Porting/bisect-example.sh*, which you should copy **outside** of the repository, as the bisect process will reset the state to a clean checkout as it runs. The instructions below assume that you copied it as *~/run* and then edited it as appropriate.

You first enter in bisect mode with:

```
% git bisect start
```

For example, if the bug is present on "HEAD" but wasn't in 5.10.0, "git" will learn about this when you enter:

```
% git bisect bad
% git bisect good perl-5.10.0
Bisecting: 853 revisions left to test after this
```

This results in checking out the median commit between "HEAD" and "perl-5.10.0". You can then run the bisecting process with:

```
% git bisect run ~/run
```

When the first bad commit is isolated, "git bisect" will tell you so:

```
ca4cfd28534303b82a216cfe83a1c80cbc3b9dc5 is first bad commit
commit ca4cfd28534303b82a216cfe83a1c80cbc3b9dc5
Author: Dave Mitchell <davem@fdisolutions.com>
Date:   Sat Feb 9 14:56:23 2008 +0000

    [perl #49472] Attributes + Unknown Error
    ...

bisect run success
```

You can peek into the bisecting process with "git bisect log" and "git bisect visualize". "git bisect reset" will get you out of bisect mode.

Please note that the first "good" state must be an ancestor of the first "bad" state. If you want to search for the commit that *solved* some bug, you have to negate your test case (i.e. exit with 1 if OK and 0 if not) and still mark the lower bound as "good" and the upper as "bad". The "first bad commit" has then to be understood as the "first commit where the bug is solved".

"git help bisect" has much more information on how you can tweak your binary searches.

Following bisection you may wish to configure, build and test perl at commits identified by the bisection process. Sometimes, particularly with older perls, "make" may fail during this process. In this case you may be able to patch the source code at the older commit point. To do so, please follow the suggestions provided in "Building perl at older commits" in perlhack.

Topic branches and rewriting history

Individual committers should create topic branches under **yourname/some_descriptive_name**:

```
% branch="$yourname/$some_descriptive_name"
% git checkout -b $branch
... do local edits, commits etc ...
% git push origin -u $branch
```

Should you be stuck with an ancient version of git (prior to 1.7), then "git push" will not have the "-u" switch, and you have to replace the last step with the following sequence:

```
% git push origin $branch:refs/heads/$branch
% git config branch.$branch.remote origin
% git config branch.$branch.merge refs/heads/$branch
```

If you want to make changes to someone else’s topic branch, you should check with its creator before making any change to it.

You might sometimes find that the original author has edited the branch’s history. There are lots of good reasons for this. Sometimes, an author might simply be rebasing the branch onto a newer source point. Sometimes, an author might have found an error in an early commit which they wanted to fix before merging the branch to blead.

Currently the master repository is configured to forbid non-fast-forward merges. This means that the branches within can not be rebased and pushed as a single step.

The only way you will ever be allowed to rebase or modify the history of a pushed branch is to delete it and push it as a new branch under the same name. Please think carefully about doing this. It may be better to sequentially rename your branches so that it is easier for others working with you to cherry-pick their local changes onto the new version. (XXX: needs explanation).

If you want to rebase a personal topic branch, you will have to delete your existing topic branch and push as a new version of it. You can do this via the following formula (see the explanation about "refspec"’s in the git push documentation for details) after you have rebased your branch:

```
# first rebase
% git checkout $user/$topic
% git fetch
% git rebase origin/blead
```

```
  # then "delete-and-push"
  % git push origin :$user/$topic
  % git push origin $user/$topic
```

NOTE: it is forbidden at the repository level to delete any of the "primary" branches. That is any branch matching "m!^(blead|maint|perl)!". Any attempt to do so will result in git producing an error like this:

```
  % git push origin :blead
  *** It is forbidden to delete blead/maint branches in this repository
  error: hooks/update exited with error code 1
  error: hook declined to update refs/heads/blead
  To ssh://perl5.git.perl.org/perl
   ! [remote rejected] blead (hook declined)
   error: failed to push some refs to 'ssh://perl5.git.perl.org/perl'
```

As a matter of policy we do **not** edit the history of the blead and maint-* branches. If a typo (or worse) sneaks into a commit to blead or maint-*, we'll fix it in another commit. The only types of updates allowed on these branches are "fast-forwards", where all history is preserved.

Annotated tags in the canonical perl.git repository will never be deleted or modified. Think long and hard about whether you want to push a local tag to perl.git before doing so. (Pushing simple tags is not allowed.)

Grafts

The perl history contains one mistake which was not caught in the conversion: a merge was recorded in the history between blead and maint-5.10 where no merge actually occurred. Due to the nature of git, this is now impossible to fix in the public repository. You can remove this mis-merge locally by adding the following line to your ".git/info/grafts" file:

```
 296f12bbbbaa06de9be9d09d3dcf8f4528898a49 434946e0cb7a32589ed92d18008aaa1d88515930
```

It is particularly important to have this graft line if any bisecting is done in the area of the "merge" in question.

WRITE ACCESS TO THE GIT REPOSITORY

Once you have write access, you will need to modify the URL for the origin remote to enable pushing. Edit *.git/config* with the **git-config**(1) command:

```
  % git config remote.origin.url git@github.com:Perl/perl5.git
```

You can also set up your user name and e-mail address. Most people do this once globally in their *~/.gitconfig* by doing something like:

```
‘ Bjarmason"ig --global user.name "AEvar Arnfjoerd
  % git config --global user.email avarab@gmail.com
```

However, if you’d like to override that just for perl, execute something like the following in *perl*:

```
  % git config user.email avar@cpan.org
```

It is also possible to keep "origin" as a git remote, and add a new remote for ssh access:

```
  % git remote add camel git@github.com:Perl/perl5.git
```

This allows you to update your local repository by pulling from "origin", which is faster and doesn’t require you to authenticate, and to push your changes back with the "camel" remote:

```
  % git fetch camel
  % git push camel
```

The "fetch" command just updates the "camel" refs, as the objects themselves should have been fetched when pulling from "origin".

Working with Github pull requests

Pull requests typically originate from outside of the "Perl/perl.git" repository, so if you want to test or work with it locally a vanilla "git fetch" from the "Perl/perl5.git" repository won’t fetch it.

However Github does provide a mechanism to fetch a pull request to a local branch. They are available on Github remotes under "pull/", so you can use "git fetch pull/*PRID*/head:*localname*" to make a local copy. eg. to fetch pull request 9999 to the local branch "local-branch-name" run:

```
  git fetch origin pull/9999/head:local-branch-name
```

and then:

```
  git checkout local-branch-name
```

Note: this branch is not rebased on "blead", so instead of the checkout above, you might want:

```
  git rebase origin/blead local-branch-name
```

which rebases "local-branch-name" on "blead", and checks it out.

Alternatively you can configure the remote to fetch all pull requests as remote-tracking branches. To do this edit the remote in *.git/config*, for example if your github remote is "origin" you'd have:

```
[remote "origin"]
        url = git@github.com:/Perl/perl5.git
        fetch = +refs/heads/*:refs/remotes/origin/*
```

Add a line to map the remote pull request branches to remote-tracking branches:

```
[remote "origin"]
        url = git@github.com:/Perl/perl5.git
        fetch = +refs/heads/*:refs/remotes/origin/*
        fetch = +refs/pull/*/head:refs/remotes/origin/pull/*
```

and then do a fetch as normal:

```
git fetch origin
```

This will create a remote-tracking branch for every pull request, including closed requests.

To remove those remote-tracking branches, remove the line added above and prune:

```
git fetch -p origin # or git remote prune origin
```

Accepting a patch

If you have received a patch file generated using the above section, you should try out the patch.

First we need to create a temporary new branch for these changes and switch into it:

```
% git checkout -b experimental
```

Patches that were formatted by "git format-patch" are applied with "git am":

```
% git am 0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
Applying Rename Leon Brocard to Orange Brocard
```

Note that some UNIX mail systems can mess with text attachments containing 'From '. This will fix them up:

```
 % perl -pi -e's/^>From /From /' \
                 0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
```

If just a raw diff is provided, it is also possible use this two-step process:

```
 % git apply bugfix.diff
 % git commit -a -m "Some fixing" \
                    --author="That Guy <that.guy@internets.com>"
```

Now we can inspect the change:

```
 % git show HEAD
 commit b1b3dab48344cff6de4087efca3dbd63548ab5e2
 Author: Leon Brocard <acme@astray.com>
 Date:   Fri Dec 19 17:02:59 2008 +0000

   Rename Leon Brocard to Orange Brocard

 diff --git a/AUTHORS b/AUTHORS
 index 293dd70..722c93e 100644
 --- a/AUTHORS
 +++ b/AUTHORS
 @@ -541,7 +541,7 @@ Lars Hecking              <lhecking@nmrc.ucc.ie>
  Laszlo Molnar                  <laszlo.molnar@eth.ericsson.se>
  Leif Huhn                      <leif@hale.dkstat.com>
  Len Johnson                    <lenjay@ibm.net>
 -Leon Brocard                   <acme@astray.com>
 +Orange Brocard                 <acme@astray.com>
  Les Peters                     <lpeters@aol.net>
  Lesley Binks                   <lesley.binks@gmail.com>
  Lincoln D. Stein               <lstein@cshl.org>
```

If you are a committer to Perl and you think the patch is good, you can then merge it into blead then push it out to the main repository:

```
  % git checkout blead
  % git merge experimental
  % git push origin blead
```

If you want to delete your temporary branch, you may do so with:

```
% git checkout blead
% git branch -d experimental
error: The branch 'experimental' is not an ancestor of your current
HEAD.  If you are sure you want to delete it, run 'git branch -D
experimental'.
% git branch -D experimental
Deleted branch experimental.
```

Committing to blead

The 'blead' branch will become the next production release of Perl.

Before pushing *any* local change to blead, it's incredibly important that you do a few things, lest other committers come after you with pitchforks and torches:

- Make sure you have a good commit message. See "Commit message" in perlhack for details.
- Run the test suite. You might not think that one typo fix would break a test file. You'd be wrong. Here's an example of where not running the suite caused problems. A patch was submitted that added a couple of tests to an existing *.t*. It couldn't possibly affect anything else, so no need to test beyond the single affected *.t*, right? But, the submitter's email address had changed since the last of their submissions, and this caused other tests to fail. Running the test target given in the next item would have caught this problem.
- If you don't run the full test suite, at least "make test_porting". This will run basic sanity checks. To see which sanity checks, have a look in *t/porting*.
- If you make any changes that affect miniperl or core routines that have different code paths for miniperl, be sure to run "make minitest". This will catch problems that even the full test suite will not catch because it runs a subset of tests under miniperl rather than perl.

On merging and rebasing

Simple, one-off commits pushed to the 'blead' branch should be simple commits that apply cleanly. In other words, you should make sure your work is committed against the current position of blead, so that you can push back to the master repository without merging.

Sometimes, blead will move while you're building or testing your changes. When this happens, your push will be rejected with a message like this:

```
To ssh://perl5.git.perl.org/perl.git
 ! [rejected]        blead -> blead (non-fast-forward)
```

```
error: failed to push some refs to 'ssh://perl5.git.perl.org/perl.git'
To prevent you from losing history, non-fast-forward updates were
rejected Merge the remote changes (e.g. 'git pull') before pushing
again.  See the 'Note about fast-forwards' section of 'git push --help'
for details.
```

When this happens, you can just *rebase* your work against the new position of blead, like this (assuming your remote for the master repository is "p5p"):

```
  % git fetch p5p
  % git rebase p5p/blead
```

You will see your commits being re-applied, and you will then be able to push safely. More information about rebasing can be found in the documentation for the **git-rebase**(1) command.

For larger sets of commits that only make sense together, or that would benefit from a summary of the set's purpose, you should use a merge commit. You should perform your work on a topic branch, which you should regularly rebase against blead to ensure that your code is not broken by blead moving. When you have finished your work, please perform a final rebase and test. Linear history is something that gets lost with every commit on blead, but a final rebase makes the history linear again, making it easier for future maintainers to see what has happened. Rebase as follows (assuming your work was on the branch "committer/somework"):

```
  % git checkout committer/somework
  % git rebase blead
```

Then you can merge it into master like this:

```
  % git checkout blead
  % git merge --no-ff --no-commit committer/somework
  % git commit -a
```

The switches above deserve explanation. "--no-ff" indicates that even if all your work can be applied linearly against blead, a merge commit should still be prepared. This ensures that all your work will be shown as a side branch, with all its commits merged into the mainstream blead by the merge commit.

"--no-commit" means that the merge commit will be *prepared* but not *committed*. The commit is then actually performed when you run the next command, which will bring up your editor to describe the commit. Without "--no-commit", the commit would be made with nearly no useful message, which would greatly diminish the value of the merge commit as a placeholder for the work's description.

When describing the merge commit, explain the purpose of the branch, and keep in mind that this description will probably be used by the eventual release engineer when reviewing the next perldelta document.

Committing to maintenance versions

Maintenance versions should only be altered to add critical bug fixes, see perlpolicy.

To commit to a maintenance version of perl, you need to create a local tracking branch:

```
  % git checkout --track -b maint-5.005 origin/maint-5.005
```

This creates a local branch named "maint-5.005", which tracks the remote branch "origin/maint-5.005". Then you can pull, commit, merge and push as before.

You can also cherry-pick commits from blead and another branch, by using the "git cherry-pick" command. It is recommended to use the **-x** option to "git cherry-pick" in order to record the SHA1 of the original commit in the new commit message.

Before pushing any change to a maint version, make sure you've satisfied the steps in "Committing to blead" above.

Using a smoke-me branch to test changes

Sometimes a change affects code paths which you cannot test on the OSes which are directly available to you and it would be wise to have users on other OSes test the change before you commit it to blead.

Fortunately, there is a way to get your change smoke-tested on various OSes: push it to a "smoke-me" branch and wait for certain automated smoke-testers to report the results from their OSes. A "smoke-me" branch is identified by the branch name: specifically, as seen on github.com it must be a local branch whose first name component is precisely "smoke-me".

The procedure for doing this is roughly as follows (using the example of tonyc's smoke-me branch called win32stat):

First, make a local branch and switch to it:

```
  % git checkout -b win32stat
```

Make some changes, build perl and test your changes, then commit them to your local branch. Then push your local branch to a remote smoke-me branch:

```
  % git push origin win32stat:smoke-me/tonyc/win32stat
```

Now you can switch back to blead locally:

```
  % git checkout blead
```

and continue working on other things while you wait a day or two, keeping an eye on the results reported for your smoke-me branch at <http://perl.develop-help.com/?b=smoke-me/tonyc/win32state>.

If all is well then update your blead branch:

```
  % git pull
```

then checkout your smoke-me branch once more and rebase it on blead:

```
  % git rebase blead win32stat
```

Now switch back to blead and merge your smoke-me branch into it:

```
  % git checkout blead
  % git merge win32stat
```

As described earlier, if there are many changes on your smoke-me branch then you should prepare a merge commit in which to give an overview of those changes by using the following command instead of the last command above:

```
  % git merge win32stat --no-ff --no-commit
```

You should now build perl and test your (merged) changes one last time (ideally run the whole test suite, but failing that at least run the *t/porting/*.t* tests) before pushing your changes as usual:

```
  % git push origin blead
```

Finally, you should then delete the remote smoke-me branch:

```
  % git push origin :smoke-me/tonyc/win32stat
```

(which is likely to produce a warning like this, which can be ignored:

```
 remote: fatal: ambiguous argument
```

```
                                   'refs/heads/smoke-me/tonyc/win32stat':
    unknown revision or path not in the working tree.
    remote: Use '--' to separate paths from revisions
```

) and then delete your local branch:

```
    % git branch -d win32stat
```