Committer’s Guide

Abstract

This document provides information for the FreeBSD committer community. All new committers should read this document before they start, and existing committers are strongly encouraged to review it from time to time.

Almost all FreeBSD developers have commit rights to one or more repositories. However, a few developers do not, and some of the information here applies to them as well. (For instance, some people only have rights to work with the Problem Report database.) Please see Issues Specific to Developers Who Are Not Committers for more information.

This document may also be of interest to members of the FreeBSD community who want to learn more about how the project works.

Table of Contents

1. Administrative Details . . . 2
2. OpenPGP Keys for FreeBSD . . . 2
3. Kerberos and LDAP web Password for FreeBSD Cluster . . . 4
4. Commit Bit Types . . . 5
5. Git Primer . . . 7
6. Version Control History . . . 41
7. Setup, Conventions, and Traditions . . . 42
8. Pre-Commit Review . . . 47
9. Commit Log Messages . . . 48
10. Preferred License for New Files . . . 56
11. Keeping Track of Licenses Granted to the FreeBSD Project . . . 57
12. SPDX Tags in the tree . . . 57
13. Developer Relations . . . 58
14. If in Doubt... . . . 59
15. Bugzilla . . . 59
16. Phabricator . . . 60
17. Who’s Who . . . 60
18. SSH Quick-Start Guide . . . 61
19. Coverity® Availability for FreeBSD Committers . . . 62
20. The FreeBSD Committers' Big List of Rules . . . 62
21. Support for Multiple Architectures . . . 71
22. Ports Specific FAQ . . . 75
23. Issues Specific to Developers Who Are Not Committers . . . 81

24. Information About Analytics . . . 81
25. Miscellaneous Questions . . . 81
26. Benefits and Perks for FreeBSD Committers . . . 82

1. Administrative Details

Login Methods	ssh(1), protocol 2 only
Main Shell Host	`freefall.FreeBSD.org`
Reference Machines	`ref*.FreeBSD.org`, `universe*.freeBSD.org` (see also FreeBSD Project Hosts)
SMTP Host	`smtp.FreeBSD.org:587` (see also SMTP Access Setup).
`src/` Git Repository	`ssh://git@gitrepo.FreeBSD.org/src.git`
`doc/` Git Repository	`ssh://git@gitrepo.FreeBSD.org/doc.git`
`ports/` Git Repository	`ssh://git@gitrepo.FreeBSD.org/ports.git`
Internal Mailing Lists	developers (technically called all-developers), doc-developers, doc-committers, ports-developers, ports-committers, src-developers, src-committers. (Each project repository has its own -developers and -committers mailing lists. Archives for these lists can be found in the files /local/mail/repository-name-developers-archive and /local/mail/repository-name-committers-archive on `freefall.FreeBSD.org`.)
Core Team monthly reports	/home/core/public/reports on the `FreeBSD.org` cluster.
Ports Management Team monthly reports	/home/portmgr/public/monthly-reports on the `FreeBSD.org` cluster.
Noteworthy `src/` Git Branches:	`stable/n` (`n`-STABLE), `main` (-CURRENT)

ssh(1) is required to connect to the project hosts. For more information, see SSH Quick-Start Guide.

Useful links:

- FreeBSD Project Internal Pages
- FreeBSD Project Hosts
- FreeBSD Project Administrative Groups

2. OpenPGP Keys for FreeBSD

Cryptographic keys conforming to the OpenPGP (*Pretty Good Privacy*) standard are used by the

FreeBSD project to authenticate committers. Messages carrying important information like public SSH keys can be signed with the OpenPGP key to prove that they are really from the committer. See PGP & GPG: Email for the Practical Paranoid by Michael Lucas and https://en.wikipedia.org/wiki/Pretty_Good_Privacy for more information.

2.1. Creating a Key

Existing keys can be used, but should be checked with documentation/tools/checkkey.sh first. In this case, make sure the key has a FreeBSD user ID.

For those who do not yet have an OpenPGP key, or need a new key to meet FreeBSD security requirements, here we show how to generate one.

1. Install security/gnupg. Enter these lines in ~/.gnupg/gpg.conf to set minimum acceptable defaults for signing and new key preferences (see the GnuPG options documentation for more details):

```
# Sorted list of preferred algorithms for signing (strongest to weakest).
personal-digest-preferences SHA512 SHA384 SHA256 SHA224
# Default preferences for new keys
default-preference-list SHA512 SHA384 SHA256 SHA224 AES256 CAMELLIA256 AES192
CAMELLIA192 AES CAMELLIA128 CAST5 BZIP2 ZLIB ZIP Uncompressed
```

2. Generate a key:

```
% gpg --full-gen-key
gpg (GnuPG) 2.1.8; Copyright (C) 2015 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Warning: using insecure memory!
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) 2048 ①
Requested keysize is 2048 bits
Please specify how long the key should be valid.
     0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 3y ②
```

```
Key expires at Wed Nov  4 17:20:20 2015 MST
Is this correct? (y/N) y
GnuPG needs to construct a user ID to identify your key.

Real name: Chucky Daemon ③
Email address: notreal@example.com
Comment:
You selected this USER-ID:
"Chucky Daemon <notreal@example.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
You need a Passphrase to protect your secret key.
```

① 2048-bit keys with a three-year expiration provide adequate protection at present (2022-10).

② A three year key lifespan is short enough to obsolete keys weakened by advancing computer power, but long enough to reduce key management problems.

③ Use your real name here, preferably matching that shown on government-issued ID to make it easier for others to verify your identity. Text that may help others identify you can be entered in the `Comment` section.
After the email address is entered, a passphrase is requested. Methods of creating a secure passphrase are contentious. Rather than suggest a single way, here are some links to sites that describe various methods: https://world.std.com/~reinhold/diceware.html, https://www.iusmentis.com/security/passphrasefaq/, https://xkcd.com/936/, https://en.wikipedia.org/wiki/Passphrase.

Protect the private key and passphrase. If either the private key or passphrase may have been compromised or disclosed, immediately notify accounts@FreeBSD.org and revoke the key.

Committing the new key is shown in Steps for New Committers.

3. Kerberos and LDAP web Password for FreeBSD Cluster

The FreeBSD cluster requires a Kerberos password to access certain services. The Kerberos password also serves as the LDAP web password, since LDAP is proxying to Kerberos in the cluster. Some of the services which require this include:

- Bugzilla

To create a new Kerberos account in the FreeBSD cluster, or to reset a Kerberos password for an existing account using a random password generator:

```
% ssh kpasswd.freebsd.org
```



This must be done from a machine outside of the FreeBSD.org cluster.

A Kerberos password can also be set manually by logging into `freefall.FreeBSD.org` and running:

```
% kpasswd
```



Unless the Kerberos-authenticated services of the FreeBSD.org cluster have been used previously, `Client unknown` will be shown. This error means that the `ssh kpasswd.freebsd.org` method shown above must be used first to initialize the Kerberos account.

4. Commit Bit Types

The FreeBSD repository has a number of components which, when combined, support the basic operating system source, documentation, third party application ports infrastructure, and various maintained utilities. When FreeBSD commit bits are allocated, the areas of the tree where the bit may be used are specified. Generally, the areas associated with a bit reflect who authorized the allocation of the commit bit. Additional areas of authority may be added at a later date: when this occurs, the committer should follow normal commit bit allocation procedures for that area of the tree, seeking approval from the appropriate entity and possibly getting a mentor for that area for some period of time.

Committer Type	*Responsible*	*Tree Components*
src	srcmgr@	src/
doc	doceng@	doc/, ports/, src/ documentation
ports	portmgr@	ports/

Commit bits allocated prior to the development of the notion of areas of authority may be appropriate for use in many parts of the tree. However, common sense dictates that a committer who has not previously worked in an area of the tree seek review prior to committing, seek approval from the appropriate responsible party, and/or work with a mentor. Since the rules regarding code maintenance differ by area of the tree, this is as much for the benefit of the committer working in an area of less familiarity as it is for others working on the tree.

Committers are encouraged to seek review for their work as part of the normal development process, regardless of the area of the tree where the work is occurring.

4.1. Policy for Committer Activity in Other Trees

Committers are trusted to use good judgment when working outside their primary area. The Cross Repo Commit Policy describes the overall framework. The following guidelines apply:

4.1.1. General Guidelines

These changes do not require review or approval from a committer in the target repository:

- All committers may modify src/share/misc/committers-*.dot, src/usr.bin/calendar/calendars/calendar.freebsd, and ports/astro/xearth/files.
- All committers may commit documentation changes to src files, such as manual pages, READMEs, fortune databases, calendar files, and comment fixes, subject to the normal care and tending of commits.
- Obvious trivial fixes — typos, broken links, and similar trivial corrections — may be committed by any committer to any repository (see also Documentation Implicit (Blanket) Approval).
- When you have recently introduced a build breakage that's trivial to resolve, though a revert and a full review usually is better for anything non-trivial.

4.1.2. Non-Obvious Changes

For changes beyond those described here, committers should seek review from committers with the appropriate bit in the target repository. When in doubt, seek review, even if you have a commit bit in the repo. Always test build and test run or read files affected by the change.

4.1.3. Acquiring Additional Commit Bits

Committers can acquire an additional bit by the usual process of finding a mentor who will propose them to srcmgr, doceng, or portmgr, as appropriate.

4.1.4. Documentation Implicit (Blanket) Approval

Some types of fixes have "blanket approval" from the Documentation Engineering Team <doceng@FreeBSD.org>, allowing any committer to fix those categories of problems on any part of the doc tree. These fixes do not need approval or review from a doc committer if the author doesn't have a doc commit bit.

Blanket approval applies to these types of fixes:

- Typos
- Trivial fixes

  Punctuation, URLs, dates, paths and file names with outdated or incorrect information, and other common mistakes that may confound the readers.

Over the years, some implicit approvals were granted in the doc tree. This list shows the most common cases:

- Changes in documentation/content/en/books/porters-handbook/versions/_index.adoc

  __FreeBSD_version Values (Porter's Handbook), mainly used for src committers.

- Changes in doc/shared/contrib-additional.adoc

Additional FreeBSD Contributors maintenance.

- All Steps for New Committers, doc related
- Security advisories; Errata Notices; Releases;

  Used by Security Officer Team <security-officer@FreeBSD.org> and Release Engineering Team <re@FreeBSD.org>.

- Changes in website/content/en/donations/donors.adoc

  Used by Donations Liaison Office <donations@FreeBSD.org>.

Before any commit, a build test is necessary; see the 'Overview' and 'The FreeBSD Documentation Build Process' sections of the FreeBSD Documentation Project Primer for New Contributors for more details.

5. Git Primer

5.1. Git basics

When one searches for "Git Primer" a number of good ones come up. Daniel Miessler's A git primer and Willie Willus' Git - Quick Primer are both good overviews. The Git book is also complete, but much longer https://git-scm.com/book/en/v2. There is also this website https://dangitgit.com/ for common traps and pitfalls of Git, in case you need guidance to fix things up. Finally, an introduction targeted at computer scientists has proven helpful to some at explaining the Git world view.

This document will assume that you've read through it and will try not to belabor the basics (though it will cover them briefly).

5.2. Git Mini Primer

This primer is less ambitiously scoped than the old Subversion Primer, but should cover the basics.

5.2.1. Scope

If you want to download FreeBSD, compile it from sources, and generally keep up to date that way, this primer is for you. It covers getting the sources, updating the sources, bisecting and touches briefly on how to cope with a few local changes. It covers the basics, and tries to give good pointers to more in-depth treatment for when the reader finds the basics insufficient. Other sections of this guide cover more advanced topics related to contributing to the project.

The goal of this section is to highlight those bits of Git needed to track sources. They assume a basic understanding of Git. There are many primers for Git on the web, but the Git Book provides one of the better treatments.

5.2.2. Getting Started For Developers

This section describes the read-write access for committers to push the commits from developers or contributors.

Daily use



In the examples below, replace `${repo}` with the name of the desired FreeBSD repository: `doc`, `ports`, or `src`.

- Clone the repository:

```
% git clone -o freebsd --config remote.freebsd.fetch='+refs/notes/*:refs/notes/*'
https://git.freebsd.org/${repo}.git
```

Then you should have the official mirrors as your remote:

```
% git remote -v
freebsd  https://git.freebsd.org/${repo}.git (fetch)
freebsd  https://git.freebsd.org/${repo}.git (push)
```

- Configure the FreeBSD committer data:

The commit hook in repo.freebsd.org checks the "Commit" field matches the committer's information in FreeBSD.org. The easiest way to get the suggested config is by executing `/usr/local/bin/gen-gitconfig.sh` script on freefall:

```
% gen-gitconfig.sh
[...]
% git config user.name (your name in gecos)
% git config user.email (your login)@FreeBSD.org
```

- Set the push URL:

```
% git remote set-url --push freebsd git@gitrepo.freebsd.org:${repo}.git
```

Then you should have separated fetch and push URLs as the most efficient setup:

```
% git remote -v
freebsd  https://git.freebsd.org/${repo}.git (fetch)
freebsd  git@gitrepo.freebsd.org:${repo}.git (push)
```

Again, note that `gitrepo.freebsd.org` has been canonicalized to `repo.freebsd.org`.

- Install commit message template hook:

For doc repository:

```
% cd .git/hooks
% ln -s ../../.hooks/prepare-commit-msg
```

For ports repository:

```
% git config --add core.hooksPath .hooks
```

For src repository:

```
% cd .git/hooks
% ln -s ../../tools/tools/git/hooks/prepare-commit-msg
```

"admin" branch

The `access` and `mentors` files are stored in an orphan branch, `internal/admin`, in each repository.

Following example is how to check out the `internal/admin` branch to a local branch named `admin`:

```
% git config --add remote.freebsd.fetch '+refs/internal/*:refs/internal/*'
% git fetch
% git checkout -b admin internal/admin
```

Alternatively, you can add a worktree for the `admin` branch:

```
git worktree add -b admin ../${repo}-admin internal/admin
```

For browsing `internal/admin` branch on web: `https://cgit.freebsd.org/${repo}/log/?h=internal/admin`

For pushing, specify the full refspec:

```
git push freebsd HEAD:refs/internal/admin
```

5.2.3. Keeping Current With The FreeBSD src Tree

First step: cloning a tree. This downloads the entire tree. There are two ways to download. Most people will want to do a deep clone of the repository. However, there are times when you may wish to do a shallow clone.

Branch Names

FreeBSD-CURRENT uses the `main` branch.

`main` is the default branch.

For FreeBSD-STABLE, branch names include `stable/15` and `stable/14`.

For FreeBSD-RELEASE, release engineering branch names include `releng/15.1` and `releng/14.5`.

https://www.freebsd.org/releng/ shows:

- `main` and `stable/⋯` branches open
- `releng/⋯` branches, each of which is frozen when a release is tagged.

Examples:

- tag release/15.1.0 on the releng/15.1 branch
- tag release/14.5.0 on the releng/14.5 branch.

Repositories

Please see the Administrative Details for the latest information on where to get FreeBSD sources. $URL below can be obtained from that page.

Note: The project doesn't use submodules as they are a poor fit for our workflows and development model. How we track changes in third-party applications is discussed elsewhere and generally of little concern to the casual user.

Deep Clone

A deep clone pulls in the entire tree, as well as all the history and branches. It is the easiest to do. It also allows you to use Git's worktree feature to have all your active branches checked out into separate directories but with only one copy of the repository.

```
% git clone -o freebsd $URL -b branch [<directory>]
```

— will create a deep clone. `branch` should be one of the branches listed in the previous section. If no `branch` is given: the default (`main`) will be used. If no `<directory>` is given: the name of the new directory will match the name of the repo (doc, ports or src).

You will want a deep clone if you are interested in the history, plan on making local changes, or plan on working on more than one branch. It is the easiest to keep up to date as well. If you are interested in the history, but are working with only one branch and are short on space, you can also use --single-branch to only download the one branch (though some merge commits will not reference the merged-from branch which may be important for some users who are interested in detailed versions of history).

Shallow Clone

A shallow clone copies just the most current code, but none or little of the history. This can be useful when you need to build a specific revision of FreeBSD, or when you are just starting out and plan to track the tree more fully. You can also use it to limit history to only so many revisions.

However, see below for a significant limitation of this approach.

```
% git clone -o freebsd -b branch --depth 1 $URL [dir]
```

This clones the repository, but only has the most recent version in the repository. The rest of the history is not downloaded. Should you change your mind later, you can do `git fetch --unshallow` to get the old history.



When you make a shallow clone, you will lose the commit count in your uname output. This can make it more difficult to determine if your system needs to be updated when a security advisory is issued.

Building

Once you've downloaded, building is done as described in the handbook, e.g.:

```
% cd src
% make buildworld
% make buildkernel
% make installkernel
% make installworld
```

so that won't be covered in depth here.

If you want to build a custom kernel, the kernel config section of the FreeBSD Handbook recommends creating a file MYKERNEL under sys/${ARCH}/conf with your changes against GENERIC. To have MYKERNEL disregarded by Git, it can be added to .git/info/exclude.

Updating

To update both types of trees uses the same commands. This pulls in all the revisions since your last update.

```
% git pull --ff-only
```

will update the tree. In Git, a 'fast forward' merge is one that only needs to set a new branch pointer and doesn't need to re-create the commits. By always doing a fast forward merge/pull, you'll ensure that you have an exact copy of the FreeBSD tree. This will be important if you want to maintain local patches.

See below for how to manage local changes. The simplest is to use `--autostash` on the `git pull` command, but more sophisticated options are available.

5.2.4. Selecting a Specific Version

In Git, `git checkout` checks out both branches and specific versions. Git's versions are the long

hashes rather than a sequential number.

When you checkout a specific version, just specify the hash you want on the command line (the git log command can help you decide which hash you might want):

```
% git checkout 08b8197a74
```

and you have that checked out. You will be greeted with a message similar to the following:

```
Note: checking out '08b8197a742a96964d2924391bf9fdfeb788865d'.

You are in a 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 08b8197a742a hook gpiokeys.4 to the build
```

where the last line is generated from the hash you are checking out and the first line of the commit message from that revision. The hash can be abbreviated to the shortest unique length. Git itself is inconsistent about how many digits it displays.

5.2.5. Bisecting

Sometimes, things go wrong. The last version worked, but the one you just updated to does not. A developer may ask you to bisect the problem to track down which commit caused the regression.

Git makes bisecting changes easy with a powerful `git bisect` command. Here's a brief outline of how to use it. For more information, you can view https://www.metaltoad.com/blog/beginners-guide-git-bisect-process-elimination or https://git-scm.com/docs/git-bisect for more details. The man git-bisect page is good at describing what can go wrong, what to do when versions won't build, when you want to use terms other than 'good' and 'bad', etc, none of which will be covered here.

`git bisect start --first-parent` will start the bisection process. Next, you need to tell a range to go through. `git bisect good XXXXXX` will tell it the working version and `git bisect bad XXXXX` will tell it the bad version. The bad version will almost always be HEAD (a special tag for what you have checked out). The good version will be the last one you checked out. The `--first-parent` argument is necessary so that subsequent `git bisect` commands do not try to check out a vendor branch which lacks the full FreeBSD source tree.



If you want to know the last version you checked out, you should use `git reflog`:

```
5ef0bd68b515 (HEAD -> main, freebsd/main, freebsd/HEAD) HEAD@{0}: pull
```

```
--ff-only: Fast-forward
a8163e165c5b (upstream/main) HEAD@{1}: checkout: moving from
b6fb97efb682994f59b21fe4efb3fcfc0e5b9eeb to main
...
```

shows me moving the working tree to the `main` branch (a816...) and then updating from upstream (to 5ef0...). In this case, bad would be HEAD (or 5ef0bd68b515) and good would be a8163e165c5b. As you can see from the output, HEAD@{1} also often works, but isn't foolproof if you have done other things to your Git tree after updating, but before you discover the need to bisect.

Set the 'good' version first, then set the bad (though the order doesn't matter). When you set the bad version, it will give you some statistics on the process:

```
% git bisect start --first-parent
% git bisect good a8163e165c5b
% git bisect bad HEAD
Bisecting: 1722 revisions left to test after this (roughly 11 steps)
[c427b3158fd8225f6afc09e7e6f62326f9e4de7e] Fixup r361997 by balancing parens.  Duh.
```

You would then build/install that version. If it's good you'd type `git bisect good` otherwise `git bisect bad`. If the version doesn't compile, type `git bisect skip`. You will get a similar message to the above after each step. When you are done, report the bad version to the developer (or fix the bug yourself and send a patch). `git bisect reset` will end the process and return you back to where you started (usually tip of `main`). Again, the git-bisect manual (linked above) is a good resource for when things go wrong or for unusual cases.

5.2.6. Signing the commits, tags, and pushes, with GnuPG

Git knows how to sign commits, tags, and pushes. When you sign a Git commit or a tag, you can prove that the code you submitted came from you and wasn't altered while you were transferring it. You also can prove that you submitted the code and not someone else.

A more in-depth documentation on signing commits and tags can be found in the Git Tools - Signing Your Work chapter of the Git's book.

The rationale behind signing pushes can be found in the commit that introduced the feature.

The best way is to simply tell Git you always want to sign commits, tags, and pushes. You can do this by setting a few configuration variables:

```
% git config --add user.signingKey LONG-KEY-ID
% git config --add commit.gpgSign true
% git config --add tag.gpgSign true
% git config --add push.gpgSign if-asked
```



To avoid possible collisions, make sure you give a long key id to Git. You can get the

long id with: `gpg --list-secret-keys --keyid-format LONG`.



To use specific subkeys, and not have GnuPG to resolve the subkey to a primary key, attach `!` to the key. For example, to encrypt for the subkey `DEADBEEF`, use `DEADBEEF!`.

Verifying signatures

Commit signatures can be verified by running either `git verify-commit <commit hash>`, or `git log --show-signature`.

Tag signatures can be verified with `git verify-tag <tag name>`, or `git tag -v <tag name>`.

5.2.7. Ports Considerations

The ports tree operates the same way. The branch names are different and the repositories are in different locations.

The cgit repository web interface for use with web browsers is at https://cgit.FreeBSD.org/ports/ . The production Git repository is at https://git.FreeBSD.org/ports.git and at ssh://anongit@git.FreeBSD.org/ports.git (or `anongit@git.FreeBSD.org:ports.git`).

There is also a mirror on GitHub, see External mirrors for an overview. The *latest* branch is `main`. The *quarterly* branches are named `yyyyQn` for year 'yyyy' and quarter 'n'.

Commit message formats

A hook is available in the ports repository to help you write up your commit messages in .hooks/prepare-commit-message. It can be enabled by running `git config --add core.hooksPath .hooks`.

The main point being that a commit message should be formatted in the following way:

```
category/port: Summary.

Description of why the changes where made.

PR:     12345
```



The first line is the subject of the commit, it contains what port was changed, and a summary of the commit. It should contain 50 characters or less.

A blank line should separate it from the rest of the commit message.

The rest of the commit message should be wrapped at the 72 characters boundary.

Another blank line should be added if there are any metadata fields, so that they are easily distinguishable from the commit message.

5.2.8. Managing Local Changes

This section addresses tracking local changes. If you have no local changes you can skip this section.

One item that is important for all of them: all changes are local until pushed. Unlike Subversion, Git uses a distributed model. For users, for most things, there is very little difference. However, if you have local changes, you can use the same tool to manage them as you use to pull in changes from FreeBSD. All changes that you have not pushed are local and can easily be modified (git rebase, discussed below does this).

Keeping local changes

The simplest way to keep local changes (especially trivial ones) is to use `git stash`. In its simplest form, you use `git stash` to record the changes (which pushes them onto the stash stack). Most people use this to save changes before updating the tree as described above. They then use `git stash apply` to re-apply them to the tree. The stash is a stack of changes that can be examined with `git stash list`. The git-stash man page (https://git-scm.com/docs/git-stash) has all the details.

This method is suitable when you have tiny tweaks to the tree. When you have anything non trivial, you'll likely be better off keeping a local branch and rebasing. Stashing is also integrated with the `git pull` command: just add `--autostash` to the command line.

Keeping a local branch

It is much easier to keep a local branch with Git than Subversion. In Subversion you need to merge the commit, and resolve the conflicts. This is manageable, but can lead to a convoluted history that's hard to upstream should that ever be necessary, or hard to replicate if you need to do so. Git also allows one to merge, along with the same problems. That's one way to manage the branch, but it's the least flexible.

In addition to merging, Git supports the concept of 'rebasing' which avoids these issues. The `git rebase` command replays all the commits of a branch at a newer location on the parent branch. We will cover the most common scenarios that arise using it.

Create a branch

Let's say you want to make a change to FreeBSD's ls command to never, ever do color. There are many reasons to do this, but this example will use that as a baseline. The FreeBSD ls command changes from time to time, and you'll need to cope with those changes. Fortunately, with Git rebase it usually is automatic.

```
% cd src
% git checkout main
% git checkout -b no-color-ls
% cd bin/ls
% vi ls.c     # hack the changes in
% git diff    # check the changes
diff --git a/bin/ls/ls.c b/bin/ls/ls.c
index 7378268867ef..cfc3f4342531 100644
--- a/bin/ls/ls.c
```

```
+++ b/bin/ls/ls.c
@@ -66,6 +66,7 @@ __FBSDID("$FreeBSD$");
 #include <stdlib.h>
 #include <string.h>
 #include <unistd.h>
+#undef COLORLS
 #ifdef COLORLS
 #include <termcap.h>
 #include <signal.h>
% # these look good, make the commit...
% git commit ls.c
```

The commit will pop you into an editor to describe what you've done. Once you enter that, you have your own **local** branch in the Git repo. Build and install it like you normally would, following the directions in the handbook. Git differs from other version control systems in that you have to tell it explicitly which files to commit. I have opted to do it on the commit command line, but you can also do it with `git add` which many of the more in depth tutorials cover.

Time to update

When it is time to bring in a new version, it is almost the same as w/o the branches. You would update like you would above, but there is one extra command before you update, and one after. The following assumes you are starting with an unmodified tree. It is important to start rebasing operations with a clean tree (Git requires this).

```
% git checkout main
% git pull --ff-only
% git rebase -i main no-color-ls
```

This will bring up an editor that lists all the commits in it. For this example, do not change it at all. This is typically what you are doing while updating the baseline (though you also use the Git rebase command to curate the commits you have in the branch).

Once you are done with the above, you have to move the commits to ls.c forward from the old version of FreeBSD to the newer one.

Sometimes there are merge conflicts. That is OK. Do not panic. Instead, handle them the same as any other merge conflicts. To keep it simple, I will just describe a common issue that may arise. A pointer to a complete treatment can be found at the end of this section.

Let's say the includes changes upstream in a radical shift to terminfo as well as a name change for the option. When you updated, you might see something like this:

```
Auto-merging bin/ls/ls.c
CONFLICT (content): Merge conflict in bin/ls/ls.c
error: could not apply 646e0f9cda11... no color ls
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
```

```
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 646e0f9cda11... no color ls
```

which looks scary.

If you bring up an editor, you will see it is a typical 3-way merge conflict resolution that you may be familiar with from other source code systems (the rest of ls.c has been omitted):

```
<<<<<<< HEAD
#ifdef COLORLS_NEW
#include <terminfo.h>
=======
#undef COLORLS
#ifdef COLORLS
#include <termcap.h>
>>>>>>> 646e0f9cda11... no color ls
```

The new code is first, and your code is second.

The right fix here is to just add a #undef COLORLS_NEW before #ifdef and then delete the old changes:

```
#undef COLORLS_NEW
#ifdef COLORLS_NEW
#include <terminfo.h>
```

save the file.

The rebase was interrupted, so you have to complete it:

```
% git add ls.c
% git rebase --continue
```

which tells Git that ls.c has been fixed and to continue the rebase operation. Since there was a conflict, you will get kicked into the editor to update the commit message if necessary. If the commit message is still accurate, just exit the editor.

If you get stuck during the rebase, do not panic. git rebase --abort will take you back to a clean slate. It is important, though, to start with an unmodified tree. An aside: The above mentioned `git reflog` comes in handy here, as it will have a list of all the (intermediate) commits that you can view or inspect or cherry-pick.

For more on this topic, https://www.freecodecamp.org/news/the-ultimate-guide-to-git-merge-and-git-rebase/ provides a rather extensive treatment. It is a good resource for issues that arise occasionally but are too obscure for this guide.

Switching to a Different FreeBSD Branch

If you wish to shift from stable/15 to the current branch. If you have a deep clone, the following will suffice:

```
% git checkout main
% # build and install here...
```

If you have a local branch, though, there are one or two caveats. First, rebase will rewrite history, so you will likely want to do something to save it. Second, jumping branches tends to cause more conflicts. If we pretend the example above was relative to stable/15, then to move to `main`, I'd suggest the following:

```
% git checkout no-color-ls
% git checkout -b no-color-ls-stable-15   # create another name for this branch
% git rebase -i stable/15 no-color-ls --onto main
```

What the above does is checkout no-color-ls. Then create a new name for it (`no-color-ls-stable-15`) in case you need to get back to it. Then you rebase onto the `main` branch. This will find all the commits to the current no-color-ls branch (back to where it meets up with the stable/15 branch) and then it will replay them onto the `main` branch creating a new no-color-ls branch there (which is why I had you create a place holder name).

5.3. MFC (Merge From Current) Procedures

5.3.1. Summary

MFC workflow can be summarized as `git cherry-pick -x` plus `git commit --amend` to adjust the commit message. For multiple commits, use `git rebase -i` to squash them together and edit the commit message.

5.3.2. Single commit MFC

```
% git checkout stable/15
% git cherry-pick -x $HASH --edit
```

For MFC commits, for example a vendor import, you would need to specify one parent for cherry-pick purposes. Normally, that would be the "first parent" of the branch you are cherry-picking from, so:

```
% git checkout stable/15
% git cherry-pick -x $HASH -m 1 --edit
```

If things go wrong, you'll either need to abort the cherry-pick with `git cherry-pick --abort` or fix it

up and do a `git cherry-pick --continue`.

Once the cherry-pick is finished, push with `git push`. If you get an error due to losing the commit race, use `git pull --rebase` and try to push again.

5.3.3. MFC to RELENG branch

MFCs to branches that require approval require a bit more care. The process is the same for either a typical merge or an exceptional direct commit.

- Merge or direct commit to the appropriate `stable/X` branch first before merging to the `releng/X.Y` branch.
- Use the hash that's in the `stable/X` branch for the MFC to `releng/X.Y` branch.
- Leave both "cherry picked from" lines in the commit message.
- Be sure to add the `Approved by:` line when you are in the editor.

```
% git checkout releng/15.2
% git cherry-pick -x $HASH --edit
```

If you forget to add the `Approved by:` line, you can do a `git commit --amend` to edit the commit message before you push the change.

5.3.4. Multiple commit MFC

```
% git checkout -b tmp-branch stable/X
% for h in $HASH_LIST; do git cherry-pick -x $h; done
% git rebase -i stable/X
# mark each of the commits after the first as 'squash'
# Update the commit message to reflect all elements of commit, if necessary.
# Be sure to retain the "cherry picked from" lines.
% git push freebsd HEAD:stable/X
```

If the push fails due to losing the commit race, rebase and try again:

```
% git checkout stable/X
% git pull
% git checkout tmp-branch
% git rebase stable/X
% git push freebsd HEAD:stable/X
```

Once the MFC is complete, you can delete the temporary branch:

```
% git checkout stable/X
% git branch -d tmp-branch
```

5.3.5. MFC a vendor import

Vendor imports are the only thing in the tree that creates a merge commit in the `main` branch. Cherry picking merge commits into stable/XX presents an additional difficulty because there are two parents for a merge commit. Generally, you'll want the first parent's diff since that's the diff to `main` (though there may be some exceptions).

```
% git cherry-pick -x -m 1 $HASH
```

is typically what you want. This will tell cherry-pick to apply the correct diff.

There are some, hopefully, rare cases where it's possible that the `main` branch was merged backwards by the conversion script. Should that be the case (and we've not found any yet), you'd change the above to `-m 2` to pickup the proper parent. Just do:

```
% git cherry-pick --abort
% git cherry-pick -x -m 2 $HASH
```

to do that. The `--abort` will cleanup the failed first attempt.

5.3.6. Redoing a MFC

If you do a MFC, and it goes horribly wrong and you want to start over, then the easiest way is to use `git reset --hard` like so:

```
% git reset --hard freebsd/stable/15
```

though if you have some revs you want to keep, and others you don't, using `git rebase -i` is better.

5.3.7. Considerations when MFCing

When committing source commits to stable and releng branches, we have the following goals:

- Clearly mark direct commits distinct from commits that land a change from another branch.
- Avoid introducing known breakage into stable and releng branches.
- Allow developers to determine which changes have or have not been landed from one branch to another.

With Subversion, we used the following practices to achieve these goals:

- Using `MFC` and `MFS` tags to mark commits that merged changes from another branch.
- Squashing fixup commits into the main commit when merging a change.
- Recording mergeinfo so that `svn mergeinfo --show-revs` worked.

With Git, we will need to use different strategies to achieve the same goals. This document aims to

define best practices when merging source commits using Git that achieve these goals. In general, we aim to use Git's native support to achieve these goals rather than enforcing practices built on Subversion's model.

One general note: due to technical differences with Git, we will not be using Git "merge commits" (created via `git merge`) in stable or releng branches. Instead, when this document refers to "merge commits", it means a commit originally made to `main` that is replicated or "landed" to a stable branch, or a commit from a stable branch that is replicated to a releng branch with some variation of `git cherry-pick`.

5.3.8. Finding Eligible Hashes to MFC

Git provides some built-in support for this via the `git cherry` and `git log --cherry` commands. These commands compare the raw diffs of commits (but not other metadata such as log messages) to determine if two commits are identical. This works well when each commit from `main` is landed as a single commit to a stable branch, but it falls over if multiple commits from `main` are squashed together as a single commit to a stable branch. The project makes extensive use of `git cherry-pick -x` with all lines preserved to work around these difficulties and is working on automated tooling to take advantage of this.

5.3.9. Commit message standards

Marking MFCs

The project has adopted the following practice for marking MFCs:

- Use the `-x` flag with `git cherry-pick`. This adds a line to the commit message that includes the hash of the original commit when merging. Since it is added by Git directly, committers do not have to manually edit the commit log when merging.

When merging multiple commits, keep all the "cherry picked from" lines.

Trim Metadata?

One area that was not clearly documented with Subversion (or even CVS) is how to format metadata in log messages for MFC commits. Should it include the metadata from the original commit unchanged, or should it be altered to reflect information about the MFC commit itself?

Historical practice has varied, though some of the variance is by field. For example, MFCs that are relevant to a PR generally include the PR field in the MFC so that MFC commits are included in the bug tracker's audit trail. Other fields are less clear. For example, Phabricator shows the diff of the last commit tagged to a review, so including Phabricator URLs replaces the main commit with the landed commits. The list of reviewers is also not clear. If a reviewer has approved a change to `main`, does that mean they have approved the MFC commit? Is that true if it's identical code only, or with merely trivial rework? It's clearly not true for more extensive reworks. Even for identical code what if the commit doesn't conflict but introduces an ABI change? A reviewer may have ok'd a commit for `main` due to the ABI breakage but may not approve of merging the same commit as-is. One will have to use one's best judgment until clear guidelines can be agreed upon.

For MFCs regulated by re@, new metadata fields are added, such as the Approved by tag for approved commits. This new metadata will have to be added via `git commit --amend` or similar after the original commit has been reviewed and approved. We may also want to reserve some metadata fields in MFC commits such as Phabricator URLs for use by re@ in the future.

Preserving existing metadata provides a very simple workflow. Developers use `git cherry-pick -x` without having to edit the log message.

If instead we choose to adjust metadata in MFCs, developers will have to edit log messages explicitly via the use of `git cherry-pick --edit` or `git commit --amend`. However, as compared to svn, at least the existing commit message can be pre-populated and metadata fields can be added or removed without having to re-enter the entire commit message.

The bottom line is that developers will likely need to curate their commit message for MFCs that are non-trivial.

5.4. Vendor Imports with Git

This section describes the vendor import procedure with Git in detail.

5.4.1. Branch naming convention

All vendor branches and tags start with `vendor/`. These branches and tags are visible by default.



This chapter follows the convention that the `freebsd` origin is the origin name for the official FreeBSD Git repository. If you use a different convention, replace `freebsd` with the name you use instead in the examples below.

We will explore an example for updating NetBSD's mtree that is in our tree. The vendor branch for this is `vendor/NetBSD/mtree`.

5.4.2. Updating an old vendor import

The vendor trees usually have only the subset of the third-party software that is appropriate to FreeBSD. These trees are usually tiny in comparison to the FreeBSD tree. Git worktrees are thus quite small and fast and the preferred method to use. Make sure that whatever directory you choose below (the `../mtree`) does not currently exist.

```
% git worktree add ../mtree vendor/NetBSD/mtree
```

5.4.3. Update the Sources in the Vendor Branch

Prepare a full, clean tree of the vendor sources. Import everything but merge only what is needed.

This example assumes the NetBSD source is checked out from their GitHub mirror in `~/git/NetBSD`. Note that "upstream" might have added or removed files, so we want to make sure deletions are propagated as well. net/rsync is commonly installed, so I'll use that.

```
% cd ../mtree
% rsync -va --del --exclude=".git" ~/git/NetBSD/usr.sbin/mtree/ .
% git add -A
% git status
...
% git diff --staged
...
% git commit -m "Vendor import of NetBSD's mtree at 2020-12-11"
[vendor/NetBSD/mtree 8e7aa25fcf1] Vendor import of NetBSD's mtree at 2020-12-11
 7 files changed, 114 insertions(+), 82 deletions(-)
% git tag -a vendor/NetBSD/mtree/20201211
```

It is critical to verify that the source code you are importing comes from a trustworthy source. Many open-source projects use cryptographic signatures to sign code changes, git tags, and/or source code tarballs. Always verify these signatures, and use isolation mechanisms like jails, chroot, in combination with a dedicated, non-privileged user account that is different from the one you regularly use (see the Updating the FreeBSD source tree section below for more details), until you are confident that the source code you are importing looks safe. Following the upstream development and occasionally reviewing the upstream code changes can greatly help in improving code quality and benefit everyone involved. It is also a good idea to examine the git diff results before importing them into the vendor area.

Always run the `git diff` and `git status` commands and examine the results carefully. When in doubt, it is useful to do a `git annotate` on the vendor branch or the upstream git repository to see who and why a change was made.

In the example above we used `-m` to illustrate, but you should compose a proper message in an editor (using a commit message template).

It is also important to create an annotated tag using `git tag -a`, otherwise the push will be rejected. Only annotated tags are allowed to be pushed. The annotated tag gives you a chance to enter a commit message. Enter the version you are importing, along with any salient new features or fixes in that version.

5.4.4. Updating the FreeBSD Copy

At this point you can push the import to `vendor` into our repo.

```
% git push --follow-tags freebsd vendor/NetBSD/mtree
```

`--follow-tags` tells `git push` to also push tags associated with the locally committed revision.

5.4.5. Updating the FreeBSD source tree

Now you need to update the mtree in FreeBSD. The sources live in `contrib/mtree` since it is upstream software.

From time to time, we may have to make changes to the contributed code to better satisfy FreeBSD's

needs. Whenever possible, please try to contribute the local changes back to the upstream projects, this helps them to better support FreeBSD, and also saves your time for future conflict resolutions when importing updates.

```
% cd ../src
% git subtree merge -P contrib/mtree vendor/NetBSD/mtree
```

This would generate a subtree merge commit of `contrib/mtree` against the local `vendor/NetBSD/mtree` branch. Examine the diff from the merge result and the contents of the upstream branch. If the merge reduced our local changes to more trivial difference like blank line or indenting changes, try amending the local changes to reduce diff against upstream, or try to contribute the remaining changes back to the upstream project. If there were conflicts, you would need to fix them before committing. Include details about the changes being merged in the merge commit message.

Some open-source software includes a `configure` script that generates files used to define how the code is built; usually, these generated files like `config.h` should be updated as part of the import process. When doing this, always keep in mind that these scripts are executable code running under the current user's credentials. This process should always be run in an isolated environment, ideally inside a jail that does not have network access, and with an unprivileged account; or, at minimum, a dedicated account that is different from the user account you normally use for everyday purposes or for pushing to the FreeBSD source code repository. This minimizes the risk of encountering bugs that can cause data loss or, in worse cases, maliciously planted code. Using an isolated jail also prevents the configure scripts from detecting locally installed software packages, which may lead to unexpected results.

When testing your changes, run them in a chroot or jailed environment, or even within a virtual machine first, especially for kernel or library modifications. This approach helps prevent adverse interactions with your working environment. It can be particularly beneficial for changes to libraries that many base system components use, among others.

5.4.6. Rebasing your change against latest FreeBSD source tree

Because the current policy recommends against using merges, if the upstream FreeBSD `main` moved forward before you get a chance to push, you would have to redo the merge.

Regular `git rebase` or `git pull --rebase` doesn't know how to rebase a merge commit **as a merge commit**, so instead of that you would have to recreate the commit.

The following steps should be taken to easily recreate the merge commit as if `git rebase --merge-commits` worked properly:

- cd to the top of the repo
- Create a side branch `XXX` with the **contents** of the merged tree.
- Update this side branch `XXX` to be merged and up-to-date with FreeBSD's `main` branch.
  - In the worst case scenario, you would still have to resolve merge conflicts, if there was any, but this should be really rare.
  - Resolve conflicts, and collapse multiple commits down to 1 if need be (without conflicts,

there's no collapse needed)

- checkout `main`
- create a branch `YYY` (allows for easier unwinding if things go wrong)
- Re-do the subtree merge
- Instead of resolving any conflicts from the subtree merge, checkout the contents of XXX on top of it.
  - The trailing `.` is important, as is being at the top level of the repo.
  - Rather than switching branches to XXX, it splats the contents of XXX on top of the repo
- Commit the results with the prior commit message (the example assumes there's only one merge on the XXX branch).
- Make sure the branches are the same.
- Do whatever review you need, including having others check it out if you think that's needed.
- Push the commit, if you 'lost the race' again, just redo these steps again (see below for a recipe)
- Delete the branches once the commit is upstream. They are throw-a-way.

The commands one would use, following the above example of mtree, would be like so (the `#` starts a comment to help link commands to descriptions above):

```
% cd ../src         # CD to top of tree
% git checkout -b XXX       # create new throw-away XXX branch for merge
% git fetch freebsd     # Get changes from upstream from upstream
% git merge freebsd/main    # Merge the changes and resolve conflicts
% git checkout -b YYY freebsd/main # Create new throw-away YYY branch for redo
% git subtree merge -P contrib/mtree vendor/NetBSD/mtree # Redo subtree merge
% git checkout XXX .        # XXX branch has the conflict resolution
% git commit -c XXX~1       # -c reuses the commit message from commit before rebase
% git diff XXX YYY      # Should be empty
% git show YYY          # Should only have changes you want, and be a merge commit
from vendor branch
```

Note: if things go wrong with the commit, you can reset the `YYY` branch by reissuing the checkout command that created it with -B to start over:

```
% git checkout -B YYY freebsd/main # Create new throw-away YYY branch if starting over
is just going to be easier
```

5.4.7. Pushing the changes

Once you think you have a set of changes that are good, you can push it to a fork off GitHub or GitLab for others to review. One nice thing about Git is that it allows you to publish rough drafts of your work for others to review. While Phabricator is good for content review, publishing the updated vendor branch and merge commits lets others check the details as they will eventually appear in the repository.

After review, when you are sure it is a good change, you can push it to the FreeBSD repo:

```
% git push freebsd YYY:main # put the commit on upstream's 'main' branch
% git branch -D XXX     # Throw away the throw-a-way branches.
% git branch -D YYY
```

Note: I used `XXX` and `YYY` to make it obvious they are terrible names and should not leave your machine. If you use such names for other work, then you'll need to pick different names, or risk losing the other work. There is nothing magic about these names. Upstream will not allow you to push them, but never the less, please pay attention to the exact commands above. Some commands use syntax that differs only slightly from typical uses and that different behavior is critical to this recipe working.

5.4.8. How to redo things if need be

If you've tried to do the push in the previous section and it fails, then you should do the following to 'redo' things. This sequence keeps the commit with the commit message always at XXX~1 to make committing easier.

```
% git checkout -B XXX YYY   # recreate that throw-away-branch XXX and switch to it
% git merge freebsd/main    # Merge the changes and resolve conflicts
% git checkout -B YYY freebsd/main # Recreate new throw-away YYY branch for redo
% git subtree merge -P contrib/mtree vendor/NetBSD/mtree # Redo subtree merge
% git checkout XXX .        # XXX branch has the conflict resolution
% git commit -c XXX~1       # -c reuses the commit message from commit before rebase
```

Then go check it out as above and push as above when ready.

5.5. Creating a new vendor branch

There are a number of ways to create a new vendor branch. The recommended way is to create a new repository and then merge that with FreeBSD. If one is importing `glorbnitz` into the FreeBSD tree, release 3.1415. For the sake of simplicity, we will not trim this release. It is a simple user command that puts the nitz device into different magical glorb states and is small enough trimming will not save much.

5.5.1. Create the repo

```
% cd /some/where
% mkdir glorbnitz
% cd glorbnitz
% git init
% git checkout -b vendor/glorbnitz
```

At this point, you have a new repo, where all new commits will go on the `vendor/glorbnitz` branch.

Git experts can also do this right in their FreeBSD clone, using `git checkout --orphan vendor/glorbnitz` if they are more comfortable with that.

5.5.2. Copy the sources in

Since this is a new import, you can just cp the sources in, or use tar or even rsync as shown above. And we will add everything, assuming no dot files.

```
% cp -r ~/glorbnitz/* .
% git add *
```

At this point, you should have a pristine copy of glorbnitz ready to commit.

```
% git commit -m "Import GlorbNitz frobnosticator revision 3.1415"
```

As above, I used `-m` for simplicity, but you should likely create a commit message that explains what a Glorb is and why you'd use a Nitz to get it. Not everybody will know so, for your actual commit, you should follow the commit log message section instead of emulating the brief style used here.

5.5.3. Now import it into our repository

Now you need to import the branch into our repository.

```
% cd /path/to/freebsd/repo/src
% git remote add glorbnitz /some/where/glorbnitz
% git fetch glorbnitz vendor/glorbnitz
```

Note the vendor/glorbnitz branch is in the repo. At this point the `/some/where/glorbnitz` can be deleted, if you like. It was only a means to an end.

5.5.4. Tag and push

Steps from here on out are much the same as they are in the case of updating a vendor branch, though without the updating the vendor branch step.

```
% git worktree add ../glorbnitz vendor/glorbnitz
% cd ../glorbnitz
% git tag --annotate vendor/glorbnitz/3.1415
# Make sure the commit is good with "git show"
% git push --follow-tags freebsd vendor/glorbnitz
```

By 'good' we mean:

1. All the right files are present
2. None of the wrong files are present

3. The vendor branch points at something sensible
4. The tag looks good, and is annotated
5. The commit message for the tag has a quick summary of what's new since the last tag

5.5.5. Time to finally merge it into the base tree

```
% cd ../src
% git subtree add -P contrib/glorbnitz vendor/glorbnitz
# Make sure the commit is good with "git show"
% git commit --amend   # one last sanity check on commit message
% git push freebsd
```

Here 'good' means:

1. All the right files, and none of the wrong ones, were merged into contrib/glorbnitz.
2. No other changes are in the tree.
3. The commit messages look good. It should contain a summary of what's changed since the last merge to the FreeBSD `main` branch and any caveats.
4. `RELNOTES` and `UPDATING` should be updated if there is anything of note, such as user visible changes, important upgrade concerns, etc.



This hasn't connected `glorbnitz` to the build yet. How so do that is specific to the software being imported and is beyond the scope of this tutorial.

Keeping current

So, time passes. It's time now to update the tree for the latest changes upstream. When you checkout `main` make sure that you have no diffs. It's a lot easier to commit those to a branch (or use `git stash`) before doing the following.

If you are used to `git pull`, we strongly recommend using the `--ff-only` option, and further setting it as the default option. Alternatively, `git pull --rebase` is useful if you have changes staged in the `main` branch.

```
% git config --global pull.ff only
```

You may need to omit the --global if you want this setting to apply to only this repository.

```
% cd freebsd-src
% git checkout main
% git pull (--ff-only|--rebase)
```

There is a common trap, that the combination command `git pull` will try to perform a merge, which would sometimes creates a merge commit that didn't exist before. This can be harder to

recover from.

The longer form is also recommended.

```
% cd freebsd-src
% git checkout main
% git fetch freebsd
% git merge --ff-only freebsd/main
```

These commands reset your tree to the `main` branch, and then update it from where you pulled the tree from originally. It's important to switch to `main` before doing this so it moves forward. Now, it's time to move the changes forward:

```
% git rebase -i main working
```

This will bring up an interactive screen to change the defaults. For now, just exit the editor. Everything should just apply. If not, then you'll need to resolve the diffs. This github document can help you navigate this process.

Time to push changes upstream

First, ensure that the push URL is properly configured for the upstream repository.

```
% git remote set-url --push freebsd ssh://git@gitrepo.freebsd.org/src.git
```

Then, verify that user name and email are configured right. We require that they exactly match the passwd entry in FreeBSD cluster.

Use

```
freefall% gen-gitconfig.sh
```

on freefall.freebsd.org to get a recipe that you can use directly, assuming /usr/local/bin is in the PATH.

The below command merges the `working` branch into the upstream `main` branch. It's important that you curate your changes to be just like you want them in the FreeBSD source repo before doing this. This syntax pushes the `working` branch to `main`, moving the `main` branch forward. You will only be able to do this if this results in a linear change to `main` (e.g. no merges).

```
% git push freebsd working:main
```

If your push is rejected due to losing a commit race, rebase your branch before trying again:

```
% git checkout working
% git fetch freebsd
% git rebase freebsd/main
% git push freebsd working:main
```

Time to push changes upstream (alternative)

Some people find it easier to merge their changes to their local `main` before pushing to the remote repository. Also, `git arc stage` moves changes from a branch to the local `main` when you need to do a subset of a branch. The instructions are similar to the prior section:

```
% git checkout main
% git merge --ff-only `working`
% git push freebsd
```

If you lose the race, then try again with

```
% git pull --rebase
% git push freebsd
```

These commands will fetch the most recent `freebsd/main` and then rebase the local `main` changes on top of that, which is what you want when you lose the commit race. Note: merging vendor branch commits will not work with this technique.

Finding the Subversion Revision

You'll need to make sure that you've fetched the notes (see the Daily usefor details). Once you have these, notes will show up in the git log command like so:

```
% git log
```

If you have a specific version in mind, you can use this construct:

```
% git log --grep revision=XXXX
```

to find the specific revision. The hex number after 'commit' is the hash you can use to refer to this commit.

5.6. Git FAQ

This section provides a number of targeted answers to questions that are likely to come up often for users and developers.



We use the common convention of having the origin for the FreeBSD repository

> being 'freebsd' rather than the default 'origin' to allow people to use that for their own development and to minimize "whoops" pushes to the wrong repository.

5.6.1. Users

How do I track -current and -stable with only one copy of the repository?

Q: Although disk space is not a huge issue, it's more efficient to use only one copy of the repository. With SVN mirroring, I could checkout multiple trees from the same repository. How do I do this with Git?

A: You can use Git worktrees. There's a number of ways to do this, but the simplest way is to use a clone to track -current, and a worktree to track stable releases. While using a 'bare repository' has been put forward as a way to cope, it's more complicated and will not be documented here.

First, you need to clone the FreeBSD repository, shown here cloning into `freebsd-current` to reduce confusion. $URL is whatever mirror works best for you:

```
% git clone -o freebsd --config remote.freebsd.fetch='+refs/notes/*:refs/notes/*' $URL
freebsd-current
```

then once that's cloned, you can simply create a worktree from it:

```
% cd freebsd-current
% git worktree add ../freebsd-stable-12 stable/12
```

this will checkout `stable/12` into a directory named `freebsd-stable-12` that's a peer to the `freebsd-current` directory. Once created, it's updated very similarly to how you might expect:

```
% cd freebsd-current
% git checkout main
% git pull --ff-only
# changes from upstream now local and current tree updated
% cd ../freebsd-stable-12
% git merge --ff-only freebsd/stable/12
# now your stable/12 is up to date too
```

I recommend using `--ff-only` because it's safer and you avoid accidentally getting into a 'merge nightmare' where you have an extra change in your tree, forcing a complicated merge rather than a simple one.

Here's a good writeup that goes into more detail.

5.6.2. Developers

Ooops! I committed to `main`, instead of another branch.

Q: From time to time, I goof up and mistakenly commit to the `main` branch. What do I do?

A: First, don't panic.

Second, don't push. In fact, you can fix almost anything if you haven't pushed. All the answers in this section assume no push has happened.

The following answer assumes you committed to `main` and want to create a branch called `issue`:

```
% git checkout -b issue              # Create the 'issue' branch
% git checkout -B main freebsd/main  # Reset main to upstream
% git checkout issue                 # Back to where you were
```

Ooops! I committed something to the wrong branch!

Q: I was working on feature on the `wilma` branch, but accidentally committed a change relevant to the `fred` branch in 'wilma'. What do I do?

A: The answer is similar to the previous one, but with cherry picking. This assumes there's only one commit on wilma, but will generalize to more complicated situations. It also assumes that it's the last commit on wilma (hence using wilma in the `git cherry-pick` command), but that too can be generalized.

```
# We're on branch wilma
% git checkout fred     # move to fred branch
% git cherry-pick wilma     # copy the misplaced commit
% git checkout wilma        # go back to wilma branch
% git reset --hard HEAD^    # move what wilma refers to back 1 commit
```

If it is not the last commit, you can cherry-pick that one change from wilma onto fred, then use `git rebase -i` to remove the change from wilma.

```
# We're on branch wilma
% git checkout fred         # move to fred branch
% git cherry-pick HASH_OF_CHANGE    # copy the misplaced commit
% git rebase -i main wilma      # drop the cherry-picked change
```

Q: But what if I want to commit a few changes to `main`, but keep the rest in `wilma` for some reason?

A: The same technique above also works if you are wanting to 'land' parts of the branch you are working on into `main` before the rest of the branch is ready (say you noticed an unrelated typo, or fixed an incidental bug). You can cherry pick those changes into `main`, then push to the parent repository. Once you've done that, cleanup couldn't be simpler: just `git rebase -i`. Git will notice you've done this and skip the common changes automatically (even if you had to change the commit message or tweak the commit slightly). There's no need to switch back to wilma to adjust it:

just rebase!

Q: I want to split off some changes from branch `wilma` into branch `fred`

A: The more general answer would be the same as the previous. You'd checkout/create the `fred` branch, cherry pick the changes you want from `wilma` one at a time, then rebase `wilma` to remove those changes you cherry picked. `git rebase -i main wilma` will toss you into an editor, and remove the `pick` lines that correspond to the commits you copied to `fred`. If all goes well, and there are no conflicts, you're done. If not, you'll need to resolve the conflicts as you go.

The other way to do this would be to checkout `wilma` and then create the branch `fred` to point to the same point in the tree. You can then `git rebase -i` both these branches, selecting the changes you want in `fred` or `wilma` by retaining the pick likes, and deleting the rest from the editor. Some people would create a tag/branch called `pre-split` before starting in case something goes wrong in the split. You can undo it with the following sequence:

```
% git checkout pre-split    # Go back
% git branch -D fred        # delete the fred branch
% git checkout -B wilma     # reset the wilma branch
% git branch -d pre-split   # Pretend it didn't happen
```

The last step is optional. If you are going to try again to split, you'd omit it.

Q: But I did things as I read along and didn't see your advice at the end to create a branch, and now `fred` and `wilma` are all screwed up. How do I find what `wilma` was before I started. I don't know how many times I moved things around.

A: All is not lost. You can figure out it, so long as it hasn't been too long, or too many commits (hundreds).

So I created a wilma branch and committed a couple of things to it, then decided I wanted to split it into fred and wilma. Nothing weird happened when I did that, but let's say it did. The way to look at what you've done is with the `git reflog`:

```
% git reflog
6ff9c25 (HEAD -> wilma) HEAD@{0}: rebase -i (finish): returning to refs/heads/wilma
6ff9c25 (HEAD -> wilma) HEAD@{1}: rebase -i (start): checkout main
869cbd3 HEAD@{2}: rebase -i (start): checkout wilma
a6a5094 (fred) HEAD@{3}: rebase -i (finish): returning to refs/heads/fred
a6a5094 (fred) HEAD@{4}: rebase -i (pick): Encourage contributions
1ccd109 (freebsd/main, main) HEAD@{5}: rebase -i (start): checkout main
869cbd3 HEAD@{6}: rebase -i (start): checkout fred
869cbd3 HEAD@{7}: checkout: moving from wilma to fred
869cbd3 HEAD@{8}: commit: Encourage contributions
...
%
```

Here we see the changes I've made. You can use it to figure out where things went wrong. I'll just

point out a few things here. The first one is that HEAD@{X} is a 'commitish' thing, so you can use that as an argument to a command. Although if that command commits anything to the repository, the X numbers change. You can also use the hash (first column).

Next, 'Encourage contributions' was the last commit I made to `wilma` before I decided to split things up. You can also see the same hash is there when I created the `fred` branch to do that. I started by rebasing `fred` and you see the 'start', each step, and the 'finish' for that process. While we don't need it here, you can figure out exactly what happened. Fortunately, to fix this, you can follow the prior answer's steps, but with the hash `869cbd3` instead of `pre-split`. While that seems a bit verbose, it's easy to remember since you're doing one thing at a time. You can also stack:

```
% git checkout -B wilma 869cbd3
% git branch -D fred
```

and you are ready to try again. The `checkout -B` with the hash combines checking out and creating a branch for it. The `-B` instead of `-b` forces the movement of a pre-existing branch. Either way works, which is what's great (and awful) about Git. One reason I tend to use `git checkout -B xxxx hash` instead of checking out the hash, and then creating / moving the branch is purely to avoid the slightly distressing message about detached heads:

```
% git checkout 869cbd3
M   faq.md
Note: checking out '869cbd3'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 869cbd3 Encourage contributions
% git checkout -B wilma
```

this produces the same effect, but I have to read a lot more and severed heads aren't an image I like to contemplate.

Ooops! I did a `git pull` and it created a merge commit, what do I do?

Q: I was on autopilot and did a `git pull` for my development tree and that created a merge commit on `main`. How do I recover?

A: This can happen when you invoke the pull with your development branch checked out.

Many developers use `git pull --rebase` to avoid this situation.

Right after the pull, you will have the new merge commit checked out. Git supports a `HEAD^#` syntax to examine the parents of a merge commit:

```
git log --oneline HEAD^1   # Look at the first parent's commits
git log --oneline HEAD^2   # Look at the second parent's commits
```

From those logs, you can easily identify which commit is your development work. Then you simply reset your branch to the corresponding `HEAD^#`:

```
git reset --hard HEAD^1
```

In addition, a `git pull --rebase` at this stage will rebase your changes to 'main' to the latest 'freebsd/main'.

Q: But I also need to fix my `main` branch. How do I do that?

A: Git keeps track of the remote repository branches in a `freebsd/` namespace. To fix your `main` branch, just make it point to the remote's `main`:

```
git branch -f main freebsd/main
```

There's nothing magical about branches in Git: they are just labels on a graph that are automatically moved forward by making commits. So the above works because you're just moving a label. There's no metadata about the branch that needs to be preserved due to this.

Mixing and matching branches

Q: So I have two branches `worker` and `async` that I'd like to combine into one branch called `feature` while maintaining the commits in both.

A: This is a job for cherry pick.

```
% git checkout worker
% git checkout -b feature   # create a new branch
% git cherry-pick main..async   # bring in the changes
```

You now have a new branch called `feature`. This branch combines commits from both branches. You can further curate it with `git rebase`.

Q: I have a branch called `driver` and I'd like to break it up into `kernel` and `userland` so I can evolve them separately and commit each branch as it becomes ready.

A: This takes a little bit of prep work, but `git rebase` will do the heavy lifting here.

```
% git checkout driver       # Checkout the driver
% git checkout -b kernel    # Create kernel branch
```

```
% git checkout -b userland  # Create userland branch
```

Now you have two identical branches. So, it's time to separate out the commits. We'll assume first that all the commits in `driver` go into either the `kernel` or the `userland` branch, but not both.

```
% git rebase -i main kernel
```

and just include the changes you want (with a 'p' or 'pick' line) and just delete the commits you don't (this sounds scary, but if worse comes to worse, you can throw this all away and start over with the `driver` branch since you've not yet moved it).

```
% git rebase -i main userland
```

and do the same thing you did with the `kernel` branch.

Q: Oh great! I followed the above and forgot a commit in the `kernel` branch. How do I recover?

A: You can use the `driver` branch to find the hash of the commit is missing and cherry pick it.

```
% git checkout kernel
% git log driver
% git cherry-pick $HASH
```

Q: OK. I have the same situation as the above, but my commits are all mixed up. I need parts of one commit to go to one branch and the rest to go to the other. In fact, I have several. Your rebase method to select sounds tricky.

A: In this situation, you'd be better off to curate the original branch to separate out the commits, and then use the above method to split the branch.

So let's assume that there's just one commit with a clean tree. You can either use `git rebase` with an `edit` line, or you can use this with the commit on the tip. The steps are the same either way. The first thing we need to do is to back up one commit while leaving the changes uncommitted in the tree:

```
% git reset HEAD^
```

Note: Do not, repeat do not, add `--hard` here since that also removes the changes from your tree.

Now, if you are lucky, the change needing to be split up falls entirely along file lines. In that case you can just do the usual `git add` for the files in each group than do a `git commit`. Note: when you do this, you'll lose the commit message when you do the reset, so if you need it for some reason, you should save a copy (though `git log $HASH` can recover it).

If you are not lucky, you'll need to split apart files. There's another tool to do that which you can apply one file at a time.

```
git add -i foo/bar.c
```

will step through the diffs, prompting you, one at time, whether to include or exclude the hunk. Once you're done, `git commit` and you'll have the remainder in your tree. You can run it multiple times as well, and even over multiple files (though I find it easier to do one file at a time and use the `git rebase -i` to fold the related commits together).

Joining the FreeBSD GitHub organization.

Q: How do I join the FreeBSD GitHub organization?

A: Please see our GitHub Wiki Info page for details. Briefly, all FreeBSD committers may join. Those who are not committers who request joining will be considered on a case by case basis.

5.6.3. Cloning and Mirroring

Q: I'd like to mirror the entire Git repository, how do I do that?

A: If all you want to do is mirror, then

```
% git clone --mirror $URL
```

will do the trick. However, there are two disadvantages to this if you want to use it for anything other than a mirror you'll reclone.

First, this is a 'bare repository' which has the repository database, but no checked out worktree. This is great for mirroring, but terrible for day to day work. There's a number of ways around this with `git worktree`:

```
% git clone --mirror https://git.freebsd.org/ports.git ports.git
% cd ports.git
% git worktree add ../ports main
% git worktree add ../quarterly branches/2020Q4
% cd ../ports
```

But if you aren't using your mirror for further local clones, then it's a poor match.

The second disadvantage is that Git normally rewrites the refs (branch name, tags, etc) from upstream so that your local refs can evolve independently of upstream. This means that you'll lose changes if you are committing to this repository on anything other than private project branches.

Q: So what can I do instead?

A: Well, you can stuff all of the upstream repository's refs into a private namespace in your local repository. Git clones everything via a 'refspec' and the default refspec is:

```
        fetch = +refs/heads/*:refs/remotes/freebsd/*
```

which says just fetch the branch refs.

However, the FreeBSD repository has a number of other things in it. To see those, you can add explicit refspecs for each ref namespace, or you can fetch everything. To setup your repository to do that:

```
git config --add remote.freebsd.fetch '+refs/*:refs/freebsd/*'
```

which will put everything in the upstream repository into your local repository's `refs/freebsd/` namespace. Please note, that this also grabs all the unconverted vendor branches and the number of refs associated with them is quite large.

You'll need to refer to these 'refs' with their full name because they aren't in and of Git's regular namespaces.

```
git log refs/freebsd/vendor/zlib/1.2.10
```

would look at the log for the vendor branch for zlib starting at 1.2.10.

5.7. Collaborating with others

One of the keys to good software development on a project as large as FreeBSD is the ability to collaborate with others before you push your changes to the tree. The FreeBSD project's Git repositories do not, yet, allow user-created branches to be pushed to the repository, and therefore if you wish to share your changes with others you must use another mechanism, such as a hosted GitLab or GitHub, to share changes in a user-generated branch.

The following instructions show how to set up a user-generated branch, based on the FreeBSD `main` branch, and push it to GitHub.

Before you begin, make sure that your local Git repo is up to date and has the correct origins set as shown above.

```
% git remote -v
freebsd https://git.freebsd.org/src.git (fetch)
freebsd ssh://git@gitrepo.freebsd.org/src.git (push)
```

The first step is to create a fork of FreeBSD on GitHub following these guidelines. The destination of the fork should be your own, personal, GitHub account (gvnn3 in my case).

Now add a remote on your local system that points to your fork:

```
% git remote add github git@github.com:gvnn3/freebsd-src.git
```

```
% git remote -v
github  git@github.com:gvnn3/freebsd-src.git (fetch)
github  git@github.com:gvnn3/freebsd-src.git (push)
freebsd https://git.freebsd.org/src.git (fetch)
freebsd ssh://git@gitrepo.freebsd.org/src.git (push)
```

With this in place you can create a branch as shown above.

```
% git checkout -b gnn-pr2001-fix
```

Make whatever modifications you wish in your branch. Build, test, and once you're ready to collaborate with others it's time to push your changes into your hosted branch. Before you can push you'll have to set the appropriate upstream, as Git will tell you the first time you try to push to your github remote:

```
% git push github
fatal: The current branch gnn-pr2001-fix has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream github gnn-pr2001-fix
```

Setting the push as git advises allows it to succeed:

```
% git push --set-upstream github gnn-feature
Enumerating objects: 20486, done.
Counting objects: 100% (20486/20486), done.
Delta compression using up to 8 threads
Compressing objects: 100% (12202/12202), done.
Writing objects: 100% (20180/20180), 56.25 MiB | 13.15 MiB/s, done.
Total 20180 (delta 11316), reused 12972 (delta 7770), pack-reused 0
remote: Resolving deltas: 100% (11316/11316), completed with 247 local objects.
remote:
remote: Create a pull request for 'gnn-feature' on GitHub by visiting:
remote:      https://github.com/gvnn3/freebsd-src/pull/new/gnn-feature
remote:
To github.com:gvnn3/freebsd-src.git
 * [new branch]                gnn-feature -> gnn-feature
Branch 'gnn-feature' set up to track remote branch 'gnn-feature' from 'github'.
```

Subsequent changes to the same branch will push correctly by default:

```
% git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
```

```
Writing objects: 100% (3/3), 314 bytes | 1024 bytes/s, done.
Total 3 (delta 1), reused 1 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:gvnn3/freebsd-src.git
   9e5243d7b659..cf6aeb8d7dda  gnn-feature -> gnn-feature
```

At this point your work is now in your branch on GitHub and you can share the link with other collaborators.

5.8. Landing a github pull request

This section documents how to land a GitHub pull request that's submitted against the FreeBSD Git mirrors at GitHub. While this is not an official way to submit patches at this time, sometimes good fixes come in this way and it is easiest just to bring them into a committer's tree and have them pushed into the FreeBSD's tree from there. Similar steps can be used to pull branches from other repositories and land those. When committing pull requests from others, one should take extra care to examine all the changes to ensure they are exactly as represented.

Before beginning, make sure that the local Git repo is up to date and has the correct origins set as shown above. In addition, make sure to have the following origins:

```
% git remote -v
freebsd https://git.freebsd.org/src.git (fetch)
freebsd ssh://git@gitrepo.freebsd.org/src.git (push)
github https://github.com/freebsd/freebsd-src (fetch)
github https://github.com/freebsd/freebsd-src (fetch)
```

Often pull requests are simple: requests that contain only a single commit. In this case, a streamlined approach may be used, though the approach in the prior section will also work. Here, a branch is created, the change is cherry picked, the commit message adjusted, and sanity-checked before being pushed. The branch `staging` is used in this example but it can be any name. This technique works for any number of commits in the pull request, especially when the changes apply cleanly to the FreeBSD tree. However, when there's multiple commits, especially when minor adjustments are needed, `git rebase -i` works better than `git cherry-pick`. Briefly, these commands create a branch; cherry-picks the changes from the pull request; tests it; adjusts the commit messages; and fast forward merges it back to `main`. The PR number is `$PR` below. When adjusting the message, add `Pull Request: https://github.com/freebsd-src/pull/$PR`. All pull requests committed to the FreeBSD repository should be reviewed by at least one person. This need not be the person committing it, but in that case the person committing it should trust the other reviewers competence to review the commit. Committers that do a code review of pull requests before pushing them into the repo should add a `Reviewed by:` line to the commit, because in this case it is not implicit. Add anybody that reviews and approves the commit on github to `Reviewed by:` as well. As always, care should be taken to ensure the change does what it is supposed to, and that no malicious code is present.



In addition, please check to make sure that the pull request author name is not anonymous. Github's web editing interface generates names like:

```
Author:    github-user <38923459+github-user@users.noreply.github.com>
```

A polite request to the author for a better name and/or email should be made. Extra care should be taken to ensure no style issue or malicious code is introduced.

```
% git fetch github pull/$PR/head:staging
% git rebase -i main staging    # to move the staging branch forward, adjust commit
message here
<do testing here, as needed>
% git checkout main
% git pull --ff-only        # to get the latest if time has passed
% git checkout main
% git merge --ff-only staging
<test again if needed>
% git push freebsd --push-option=confirm-author
```

For complicated pull requests that have multiple commits with conflicts, follow the following outline.

1. checkout the pull request `git checkout github/pull/XXX`
2. create a branch to rebase `git checkout -b staging`
3. rebase the `staging` branch to the latest `main` with `git rebase -i main staging`
4. resolve conflicts and do whatever testing is needed
5. fast forward the `staging` branch into `main` as above
6. final sanity check of changes to make sure all is well
7. push to FreeBSD's Git repository.

This will also work when bringing branches developed elsewhere into the local tree for committing.

Once finished with the pull request, close it using GitHub's web interface. It is worth noting that if your `github` origin uses `https://`, the only step you'll need a GitHub account for is closing the pull request.

6. Version Control History

The project has moved to git.

The FreeBSD source repository switched from CVS to Subversion on May 31st, 2008. The first real SVN commit is *r179447*. The source repository switched from Subversion to Git on December 23rd, 2020. The last real svn commit is *r368820*. The first real git commit hash is *5ef5f51d2bef80b0ede9b10ad5b0e9440b60518c*.

The FreeBSD doc/www repository switched from CVS to Subversion on May 19th, 2012. The first real SVN commit is *r38821*. The documentation repository switched from Subversion to Git on December 8th, 2020. The last SVN commit is *r54737*. The first real git commit hash is *3be01a475855e7511ad755b2defd2e0da5d58bbe*.

The FreeBSD ports repository switched from CVS to Subversion on July 14th, 2012. The first real SVN commit is *r300894*. The ports repository switched from Subversion to Git on April 6, 2021. The last SVN commit is *r569609* The first real git commit hash is *ed8d3eda309dd863fb66e04bccaa513eee255cbf*.

7. Setup, Conventions, and Traditions

There are a number of things to do as a new developer. The first set of steps is specific to committers only. These steps must be done by a mentor for those who are not committers.

7.1. For New Committers

Those who have been given commit rights to the FreeBSD repositories must follow these steps.

- Get mentor approval before committing each of these changes!
- All src commits go to FreeBSD-CURRENT first before being merged to FreeBSD-STABLE. The FreeBSD-STABLE branch must maintain ABI and API compatibility with earlier versions of that branch. Do not merge changes that break this compatibility.

Steps for New Committers

1. Add an Author Entity

   doc/shared/authors.adoc - Add an author entity. Later steps depend on this entity, and missing this step will cause the doc/ build to fail. This is a relatively easy task, but remains a good first test of version control skills.

2. Update the List of Developers and Contributors

   doc/shared/contrib-committers.adoc - Add an entry, which will then appear in the "Developers" section of the Contributors List. Entries are sorted by last name.

   doc/shared/contrib-additional.adoc - *Remove* the entry. Entries are sorted by first name.

3. Add a News Item

   doc/website/data/en/news/news.toml - Add an entry. Look for the other entries that announce new committers and follow the format. Use the date from the commit bit approval email.

4. Add a PGP Key

   Dag-Erling Smørgrav <des@FreeBSD.org> has written a shell script

(doc/documentation/tools/addkey.sh) to make this easier. See the README file for more information.

Use doc/documentation/tools/checkkey.sh to verify that keys meet minimal best-practices standards.

After adding and checking a key, add both updated files to source control and then commit them. Entries in this file are sorted by last name.



It is very important to have a current PGP/GnuPG key in the repository. The key may be required for positive identification of a committer. For example, the `FreeBSD Administrators <admins@FreeBSD.org>` might need it for account recovery. A complete keyring of `FreeBSD.org` users is available for download from https://docs.FreeBSD.org/pgpkeys/pgpkeys.txt.

5. Update Mentor and Mentee Information

   src/share/misc/committers-<repository>.dot - Add an entry to the current committers section, where *repository* is `doc`, `ports`, or `src`, depending on the commit privileges granted.

   Add an entry for each additional mentor/mentee relationship in the bottom section.

6. Update git mailmap file

   src/.mailmap, doc/.mailmap, and ports/.mailmap - Add an entry for commits you created prior to becoming a FreeBSD committer.

   Mapping to your FreeBSD address allows us to track external committers who may be ready for a commit bit more easily. You can also use this to correct old names, misspelled names, etc in the default `git log` output.

7. Generate a Kerberos Password

   See Kerberos and LDAP web Password for FreeBSD Cluster to generate or set a Kerberos account for use with other FreeBSD services like the bug-tracking database (you get a bug-tracking account as part of that step).

8. Optional: Enable Wiki Account

   FreeBSD Wiki Account - A wiki account allows sharing projects and ideas. Those who do not yet have an account can follow instructions on the Wiki/About page to obtain one. Contact wiki-admin@FreeBSD.org if you need help with your Wiki account.

9. Optional: Update Wiki Information

   Wiki Information - After gaining access to the wiki, some people add entries to the How We Got Here, IRC Nicks, Dogs of FreeBSD, and or Cats of FreeBSD pages.

10. Optional: Update Ports with Personal Information

    ports/astro/xearth/files/freebsd.committers.markers and

src/usr.bin/calendar/calendars/calendar.freebsd - Some people add entries for themselves to these files to show where they are located or the date of their birthday.

11. Optional: Prevent Duplicate Mailings

    Subscribers to Commit messages for all branches of the doc repository, Commit messages for all branches of the ports repository or Commit messages for all branches of the src repository might wish to unsubscribe to avoid receiving duplicate copies of commit messages and followups.

7.2. For Everyone

1. Introduce yourself to the other developers, otherwise no one will have any idea who you are or what you are working on. The introduction need not be a comprehensive biography, just write a paragraph or two about who you are, what you plan to be working on as a developer in FreeBSD, and who will be your mentor. Email this to the FreeBSD developers mailing list and you will be on your way!
2. Log into `freefall.FreeBSD.org` and create a /var/forward/user (where *user* is your username) file containing the e-mail address where you want mail addressed to *yourusername*@FreeBSD.org to be forwarded. This includes all of the commit messages as well as any other mail addressed to the FreeBSD committer's mailing list and the FreeBSD developers mailing list. Really large mailboxes which have taken up permanent residence on `freefall` may get truncated without warning if space needs to be freed, so forward it or save it elsewhere.

   

   If your e-mail system uses SPF with strict rules, you should exclude `mx2.FreeBSD.org` from SPF checks.

   Due to the severe load dealing with SPAM places on the central mail servers that do the mailing list processing, the front-end server does do some basic checks and will drop some messages based on these checks. At the moment proper DNS information for the connecting host is the only check in place but that may change. Some people blame these checks for bouncing valid email. To have these checks turned off for your email, create a file named ~/.spam_lover on `freefall.FreeBSD.org`.

   

   Those who are developers but not committers will not be subscribed to the committers or developers mailing lists. The subscriptions are derived from the access rights.

7.2.1. SMTP Access Setup

For those willing to send e-mail messages through the FreeBSD.org infrastructure, follow the instructions below:

1. Point your mail client at `smtp.FreeBSD.org:587`.
2. Enable STARTTLS.
3. Ensure your `From:` address is set to `yourusername@FreeBSD.org`.
4. For authentication, you can use your FreeBSD Kerberos username and password (see Kerberos and LDAP web Password for FreeBSD Cluster). The `yourusername/mail` principal is preferred, as it is only valid for authenticating to mail resources.



Do not include `@FreeBSD.org` when entering in your username.



Additional Notes

- Will only accept mail from `yourusername@FreeBSD.org`. If you are authenticated as one user, you are not permitted to send mail from another.
- A header will be appended with the SASL username: (`Authenticated sender: username`).
- Host has various rate limits in place to cut down on brute force attempts.

Using a Local MTA to Forward Emails to the FreeBSD.org SMTP Service

It is also possible to use a local MTA to forward locally sent emails to the FreeBSD.org SMTP servers.

Example 1. Using Postfix

To tell a local Postfix instance that anything from `yourusername@FreeBSD.org` should be forwarded to the FreeBSD.org servers, add this to your main.cf:

```
sender_dependent_relayhost_maps = hash:/usr/local/etc/postfix/relayhost_maps
smtp_sasl_auth_enable = yes
smtp_sasl_security_options = noanonymous
smtp_sasl_password_maps = hash:/usr/local/etc/postfix/sasl_passwd
smtp_use_tls = yes
```

Create /usr/local/etc/postfix/relayhost_maps with the following content:

```
yourusername@FreeBSD.org  [smtp.freebsd.org]:587
```

Create /usr/local/etc/postfix/sasl_passwd with the following content:

```
[smtp.freebsd.org]:587          yourusername:yourpassword
```

If the email server is used by other people, you may want to prevent them from sending e-

mails from your address. To achieve this, add this to your main.cf:

```
smtpd_sender_login_maps = hash:/usr/local/etc/postfix/sender_login_maps
smtpd_sender_restrictions = reject_known_sender_login_mismatch
```

Create /usr/local/etc/postfix/sender_login_maps with the following content:

```
yourusername@FreeBSD.org yourlocalusername
```

Where *yourlocalusername* is the SASL username used to connect to the local instance of Postfix.

Example 2. Using OpenSMTPD

To tell a local OpenSMTPD instance that anything from `yourusername@FreeBSD.org` should be forwarded to the FreeBSD.org servers, add this to your smtpd.conf:

```
action "freebsd" relay host smtp+tls://freebsd@smtp.freebsd.org:587 auth <secrets>
match from any auth yourlocalusername mail-from "_yourusername_@freebsd.org" for
any action "freebsd"
```

Where *yourlocalusername* is the SASL username used to connect to the local instance of OpenSMTPD.

Create /usr/local/etc/mail/secrets with the following content:

```
freebsd yourusername:yourpassword
```

Example 3. Using Exim

To direct a local Exim instance to forward all mail from `example@FreeBSD.org` to FreeBSD.org servers, add this to Exim configuration:

```
Routers section: (at the top of the list):
freebsd_send:
   driver = manualroute
   domains = !+local_domains
   transport = freebsd_smtp
   route_data = ${lookup {${lc:$sender_address}} lsearch
{/usr/local/etc/exim/freebsd_send}}

Transport Section:
freebsd_smtp:
        driver = smtp
```

```
  tls_certificate=<local certificate>
  tls_privatekey=<local certificate private key>
  tls_require_ciphers =
EECDH+ECDSA+AESGCM:EECDH+aRSA+AESGCM:EECDH+ECDSA+SHA384:EECDH+ECDSA+SHA256:EECDH+a
RSA+SHA384:EECDH+aRSA+SHA256:EECDH+AESGCM:EECDH:EDH+AESGCM:EDH+aRSA:HIGH:!MEDIUM:!
LOW:!aNULL:!eNULL:!LOW:!RC4:!MD5:!EXP:!PSK:!SRP:!DSS
  dkim_domain = <local DKIM domain>
  dkim_selector = <local DKIM selector>
  dkim_private_key= <local DKIM private key>
  dnssec_request_domains = *
  hosts_require_auth = smtp.freebsd.org

Authenticators:
freebsd_plain:
  driver = plaintext
  public_name = PLAIN
  client_send = ^example/mail^examplePassword
  client_condition = ${if eq{$host}{smtp.freebsd.org}}
```

Create /usr/local/etc/exim/freebsd_send with the following content:

```
example@freebsd.org:smtp.freebsd.org::587
```

7.3. Mentors

All new developers have a mentor assigned to them for the first few months. A mentor is responsible for teaching the mentee the rules and conventions of the project and guiding their first steps in the developer community. The mentor is also personally responsible for the mentee's actions during this initial period.

For committers: do not commit anything without first getting mentor approval. Document that approval with an `Approved by:` line in the commit message.

When the mentor decides that a mentee has learned the ropes and is ready to commit on their own, the mentor announces it with a commit to mentors. This file is in the admin orphan branch of each repository. Detailed information on how to access these branches can be found in admin.

8. Pre-Commit Review

Code review is one way to increase the quality of software. The following guidelines apply to commits to the `main` (-CURRENT) branch of the `src` repository. Other branches and the `ports` and `docs` trees have their own review policies, but these guidelines generally apply to commits requiring review:

- All non-trivial changes should be reviewed before they are committed to the repository.
- Reviews may be conducted by email, in Bugzilla, in Phabricator, or by another mechanism.

Where possible, reviews should be public.

- The developer responsible for a code change is also responsible for making all necessary review-related changes.
- Code review can be an iterative process, which continues until the patch is ready to be committed. Specifically, once a patch is sent out for review, it should receive an explicit "looks good" before it is committed. So long as it is explicit, this can take whatever form makes sense for the review method.
- Timeouts are not a substitute for review.

Sometimes code reviews will take longer than you would hope for, especially for larger features. Accepted ways to speed up review times for your patches are:

- Review other people's patches. If you help out, everybody will be more willing to do the same for you; goodwill is our currency.
- Ping the patch. If it is urgent, provide reasons why it is important to you to get this patch landed and ping it every couple of days. If it is not urgent, the common courtesy ping rate is one week. Remember that you are asking for valuable time from other professional developers.
- Ask for help on mailing lists, IRC, etc. Others may be able to either help you directly, or suggest a reviewer.
- Split your patch into multiple smaller patches that build on each other. The smaller your patch, the higher the probability that somebody will take a quick look at it.

  When making large changes, it is helpful to keep this in mind from the beginning of the effort as breaking large changes into smaller ones is often difficult after the fact.

Developers should participate in code reviews as both reviewers and reviewees. If someone is kind enough to review your code, you should return the favor for someone else. Note that while anyone is welcome to review and give feedback on a patch, only an appropriate subject-matter expert can approve a change. This will usually be a committer who works with the code in question on a regular basis.

In some cases, no subject-matter expert may be available. In those cases, a review by an experienced developer is sufficient when coupled with appropriate testing.

9. Commit Log Messages

This section contains some suggestions and traditions for how commit logs are formatted.

9.1. Why are commit messages important?

When you commit a change in Git, Subversion, or another version control system (VCS), you're prompted to write some text describing the commit — a commit message. How important is this commit message? Should you spend some significant effort writing it? Does it really matter if you write simply `fixed a bug`?

Most projects have more than one developer and last for some length of time. Commit messages are

a very important method of communicating with other developers, in the present and for the future.

FreeBSD has hundreds of active developers and hundreds of thousands of commits spanning decades of history. Over that time the developer community has learned how valuable good commit messages are; sometimes these are hard-learned lessons.

Commit messages serve at least three purposes:

- Communicating with other developers

  FreeBSD commits generate email to various mailing lists. These include the commit message along with a copy of the patch itself. Commit messages are also viewed through commands like git log. These serve to make other developers aware of changes that are ongoing; that other developer may want to test the change, may have an interest in the topic and will want to review in more detail, or may have their own projects underway that would benefit from interaction.

- Making Changes Discoverable

  In a large project with a long history it may be difficult to find changes of interest when investigating an issue or change in behaviour. Verbose, detailed commit messages allow searches for changes that might be relevant. For example, `git log --since 1year --grep 'USB timeout'`.

- Providing historical documentation

  Commit messages serve to document changes for future developers, perhaps years or decades later. This future developer may even be you, the original author. A change that seems obvious today may be decidedly not so much later on.

The `git blame` command annotates each line of a source file with the change (hash and subject line) that brought it in.

Having established the importance, here are elements of a good FreeBSD commit message:

9.2. Start with a subject line

Commit messages should start with a single-line subject that briefly summarizes the change. The subject should, by itself, allow the reader to quickly determine if the change is of interest or not.

9.3. Keep subject lines short

The subject line should be as short as possible while still retaining the required information. This is to make browsing Git log more efficient, and so that git log --oneline can display the short hash and subject on a single 80-column line. A good rule of thumb is to stay below 67 characters, and aim for about 50 or fewer if possible.

9.4. Prefix the subject line with a component, if applicable

If the change relates to a specific component the subject line may be prefixed with that component name and a colon (:). If applicable, try to use the same prefix used in previous commits to the same files.

✓ `foo: Add -k option to keep temporary data`

Include the prefix in the 67-character limit suggested above, so that `git log --oneline` avoids wrapping.

9.5. Capitalize the first letter of the subject

Capitalize the first letter of the subject itself. The prefix, if any, is not capitalized unless necessary (e.g., `USB:` is capitalized).

9.6. Do not end the subject line with punctuation

Do not end with a period or other punctuation. In this regard the subject line is like a newspaper headline.

9.7. Separate the subject and body with a blank line

Separate the body from the subject with a blank line.

Some trivial commits do not require a body, and will have only a subject.

✓ `ls: Fix typo in usage text`

9.8. Limit messages to 72 columns

`git log` and `git format-patch` indent the commit message by four spaces. Wrapping at 72 columns provides a matching margin on the right edge. Limiting messages to 72 characters also keeps the commit message in formatted patches below RFC 2822's suggested email line length limit of 78 characters. This limit works well with a variety of tools that may render commit messages; line wrapping might be inconsistent with longer line length.

9.9. Use the present tense, imperative mood

This facilitates short subject lines and provides consistency, including with automatically generated commit messages (e.g., as generated by git revert). This is important when reading a list of commit subjects. Think of the subject as finishing the sentence "when applied, this change will ...".

✓ `foo: Implement the -k (keep) option`
 `foo: Implemented the -k option`

```
 This change implements the -k option in foo
 -k option added
```

9.10. Focus on what and why, not how

Explain what the change accomplishes and why it is being done, rather than how.

Do not assume that the reader is familiar with the issue. Explain the background and motivation for the change. Include benchmark data if you have it.

If there are limitations or incomplete aspects of the change, describe them in the commit message.

9.11. Consider whether parts of the commit message could be code comments instead

Sometimes while writing a commit message you may find yourself writing a sentence or two explaining some tricky or confusing aspect of the change. When this happens consider whether it would be valuable to have that explanation as a comment in the code itself.

9.12. Write commit messages for your future self

While writing the commit message for a change you have all of the context in mind - what prompted the change, alternate approaches that were considered and rejected, limitations of the change, and so on. Imagine yourself revisiting the change a year or two in the future, and write the commit message in a way that would provide that necessary context.

9.13. Commit messages should stand alone

You may include references to mailing list postings, benchmark result web sites, or code review links. However, the commit message should contain all of the relevant information in case these references are no longer available in the future.

Similarly, a commit may refer to a previous commit, for example in the case of a bug fix or revert. In addition to the commit identifier (revision or hash), include the subject line from the referenced commit (or another suitable brief reference). With each VCS migration (from CVS to Subversion to Git) revision identifiers from previous systems may become difficult to follow.

9.14. Include appropriate metadata in a footer

As well as including an informative message with each commit, some additional information may be needed.

This information consists of one or more lines containing the key word or phrase, a colon, tabs for formatting, and then the additional information.

For key words where multiple values make sense (e.g., `PR:` with a comma-separated list of PRs), it is permitted to use the same keyword multiple times to avoid ambiguity or improve readability.

The key words or phrases are:

`PR:`	The problem report (if any) which is affected (typically, by being closed) by this commit. Multiple PRs may be specified on one line, separated by commas or spaces.
`Reported by:`	The name and e-mail address of the person that reported the issue; for developers, just the username on the FreeBSD cluster. Typically used when there is no PR, for example if the issue was reported on a mailing list.
`Submitted by:` (discouraged)	Name of an author who submitted a change without providing a full valid patch, especially without a valid email. Submitted patches should have the author set by using `git commit --author` with a full name and valid email. Before the migration to git allowed separate author and committer fields, this was used for contributed patches.
`Reviewed by:`	The name and e-mail address of the person or people that reviewed the change; for developers, just the username on the FreeBSD cluster. If a patch was submitted to a mailing list for review, and the review was favorable, then just include the list name. If the reviewer is not a member of the project, provide the name, email, and if ports an external role like maintainer: Reviewed by a developer: [source,shell] .... Reviewed by: username .... Reviewed by a ports maintainer that is not a developer: [source,shell] .... Reviewed by: Full Name <valid@email> (maintainer) ....
`Tested by:`	The name and e-mail address of the person or people that tested the change; for developers, just the username on the FreeBSD cluster.
`Discussed with:`	The name and e-mail address of the person or people that contributed to the patch by providing meaningful feedback; for developers, just the username on the FreeBSD cluster. Typically used to credit those who did not explicitly review, test, or approve the change, but nevertheless contributed to the discussion surrounding the change, which led to improvements and a better understanding of its impact on the FreeBSD project.

`Approved by:`	The name and e-mail address of the person or people that approved the change; for developers, just the username on the FreeBSD cluster.
	
	There are several cases where approval is customary:
	
	• while a new committer is under mentorship
	• commits to an area of the tree covered by the LOCKS file (src)
	• during a release cycle
	• committing non-obvious changes to a repo where you do not hold a commit bit (e.g. src committer committing to docs); for obvious fixes, `Reviewed by:` is sufficient
	• committing to a port maintained by someone else
	
	While under mentorship, get mentor approval before the commit. Enter the mentor's username in this field, and note that they are a mentor:
	
	`Approved by: username-of-mentor (mentor)`
	
	If a team approved these commits then include the team name followed by the username of the approver in parentheses. For example:
	
	`Approved by: re (username)`

`Obtained from:`	The name of the project (if any) from which the code was obtained. Do not use this line for the name of an individual person.
`Fixes:`	The Git short hash and the title line of a commit that is fixed by this change as returned by `git log -n1 --format='%h ("%s")' GIT-COMMIT-HASH`. We include the commit title so that the referenced commit can be located even in the case that a future VCS migration invalidates hash references.
`MFC after:`	To receive an e-mail reminder to MFC at a later date, specify the number of days, weeks, or months after which an MFC is planned.
`MFC to:`	If the commit should be merged to a subset of stable branches, specify the branch names.
`MFH:`	If the commit is to be merged into a ports quarterly branch name, specify the quarterly branch. For example `2021Q2`.
`Relnotes:`	If the change is a candidate for inclusion in the release notes for the next release from the branch, set to `yes`.
Candidates are user-visible changes, new features, compatibility breaks, etc..	If you forget to set this line, or want to provide more details, add an entry to the `RELNOTES` file in the root of the src tree.

The `RELNOTES` file is used to generate release notes for the next release.	Do not use the `Relnotes:` line to describe the change: its only valid value is `yes`.
`Security:`	If the change is related to a security vulnerability or security exposure, include one or more references or a description of the issue. If possible, include a VuXML URL or a CVE ID.
`Event:`	The description for the event where this commit was made. If this is a recurring event, add the year or even the month to it. For example, this could be `FooBSDcon 2019`. The idea behind this line is to put recognition to conferences, gatherings, and other types of meetups and to show that these are useful to have. Please do not use the `Sponsored by:` line for this as that is meant for organizations sponsoring certain features or developers working on them.
`Sponsored by:`	Sponsoring organizations for this change, if any. Separate multiple organizations with commas. If only a portion of the work was sponsored, or different amounts of sponsorship were provided to different authors, please give appropriate credit in parentheses after each sponsor name. For example, `Example.com (alice, code refactoring), Wormulon (bob), Momcorp (cindy)` shows that Alice was sponsored by Example.com to do code refactoring, while Wormulon sponsored Bob's work and Momcorp sponsored Cindy's work. Other authors were either not sponsored or chose not to list sponsorship.
`Pull Request:`	This change was submitted as a pull request or merge request against one of FreeBSD's public read-only Git repositories. It should include the entire URL to the pull request, as these often act as code reviews for the code. For example: `https://github.com/freebsd/freebsd-src/pull/745`
`Closes:`	This change concludes the patch series discussed at the specified Github pull request, and closes that request. It should include the entire URL to the pull request, as these often act as code reviews for the code. For example: `https://github.com/freebsd/freebsd-src/pull/745`
`Co-authored-by:`	The name and email address of an additional author of the commit. GitHub has a detailed description of the Co-authored-by trailer at https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/creating-a-commit-with-multiple-authors.
`Signed-off-by:`	ID certifies compliance with https://developercertificate.org/
`Differential Revision:`	The full URL of the Phabricator review. This line *must be the last line*. For example: `https://reviews.freebsd.org/D1708`.

Example 4. Commit Log for a Commit Based on a PR

The commit is based on a patch from a PR submitted by John Smith. The commit message "PR" field is filled.

```
...
```

```
PR:     12345
```

The committer sets the author of the patch with `git commit --author "John Smith <John.Smith@example.com>"`.

Example 5. Commit Log for a Commit Needing Review

The virtual memory system is being changed. After posting patches to the appropriate mailing list (in this case, `freebsd-arch`) and the changes have been approved.

```
...

Reviewed by:    -arch
```

Example 6. Commit Log for a Commit Needing Approval

Commit a port, after working with the listed MAINTAINER, who said to go ahead and commit.

```
...

Approved by:    abc (maintainer)
```

Where *abc* is the account name of the person who approved.

Example 7. Commit Log for a Commit Bringing in Code from OpenBSD

Committing some code based on work done in the OpenBSD project.

```
...

Obtained from:  OpenBSD
```

Example 8. Commit Log for a Change to FreeBSD-CURRENT with a Planned Commit to FreeBSD-STABLE to Follow at a Later Date.

Committing some code which will be merged from FreeBSD-CURRENT into the FreeBSD-STABLE branch after two weeks.

```
...

MFC after:  2 weeks
```

Where *2* is the number of days, weeks, or months after which an MFC is planned. The *weeks* option may be `day`, `days`, `week`, `weeks`, `month`, `months`.

It is often necessary to combine these.

Consider the situation where a user has submitted a PR containing code from the NetBSD project. Looking at the PR, the developer sees it is not an area of the tree they normally work in, so they have the change reviewed by the `arch` mailing list. Since the change is complex, the developer opts to MFC after one month to allow adequate testing.

The extra information to include in the commit would look something like

Example 9. Example Combined Commit Log

```
PR:     54321
Reviewed by:    -arch
Obtained from:  NetBSD
MFC after:  1 month
Relnotes:   yes
```

10. Preferred License for New Files

The FreeBSD Project's full license policy is documented in the FreeBSD License Policies article. The rest of this section is intended to help you get started. As a rule, when in doubt, ask. It is much easier to give advice than to fix the source tree.

SPDX-only license marking is strongly preferred. The FreeBSD Project uses this text as the preferred license:

```
/*
 * Copyright (c) [year] [your name]
 *
 * SPDX-License-Identifier: BSD-2-Clause
 */
```

The preferred order is the copyright notice first, followed by the SPDX-License-Identifier, but either order is acceptable. New contributions to FreeBSD should use the BSD-2-Clause license. The FreeBSD project does not allow the "advertising clause" in new code. If you have code in the tree with the advertising clause, please consider switching to a license without it.

The FreeBSD project discourages completely new licenses and variations on the standard licenses. New licenses require the approval of core@FreeBSD.org.

Project policy dictates that code under some non-BSD licenses must be placed only in specific sections of the repository, and in some cases, compilation must be conditional or even disabled by

default. For example, the GENERIC kernel must be compiled under only licenses identical to or substantially similar to the BSD license. GPL, APSL, CDDL, etc, licensed software must not be compiled into GENERIC.

Developers are reminded that in open source, getting "open" right is just as important as getting "source" right, as improper handling of intellectual property has serious consequences. Any questions or concerns should immediately be brought to the attention of the core team.

11. Keeping Track of Licenses Granted to the FreeBSD Project

Various software or data exist in the repositories where the FreeBSD project has been granted a special license to be able to use them. A case in point are the Terminus fonts for use with vt(4). Here the author Dimitar Zhekov has allowed us to use the "Terminus BSD Console" font under a 2-clause BSD license rather than the regular Open Font License he normally uses.

It is clearly sensible to keep a record of any such license grants. To that end, the core@FreeBSD.org has decided to keep an archive of them. Whenever the FreeBSD project is granted a special license we require the core@FreeBSD.org to be notified. Any developers involved in arranging such a license grant, please send details to the core@FreeBSD.org including:

- Contact details for people or organizations granting the special license.
- What files, directories etc. in the repositories are covered by the license grant including the revision numbers where any specially licensed material was committed.
- The date the license comes into effect from. Unless otherwise agreed, this will be the date the license was issued by the authors of the software in question.
- The license text.
- A note of any restrictions, limitations or exceptions that apply specifically to FreeBSD's usage of the licensed material.
- Any other relevant information.

Once the core@FreeBSD.org is satisfied that all the necessary details have been gathered and are correct, the secretary will send a PGP-signed acknowledgment of receipt including the license details. This receipt will be persistently archived and serve as our permanent record of the license grant.

The license archive should contain only details of license grants; this is not the place for any discussions around licensing or other subjects. Access to data within the license archive will be available on request to the core@FreeBSD.org.

12. SPDX Tags in the tree

The project uses SPDX tags in our source base. SPDX-only license marking (without reproducing the full license text) is strongly preferred for new files. When a file contains only a copyright notice and an SPDX-License-Identifier tag, the SPDX tag specifies the license for that file. When a file contains

both an SPDX-License-Identifier tag and the full verbatim license text, the SPDX tag is informative and the verbatim license is controlling in the event of a discrepancy. The project draws identifiers from SPDX's list of valid short license identifiers and uses only the *SPDX-License-Identifier* tag. See the FreeBSD License Policies article for the full specification of how SPDX expressions are used in the FreeBSD software collection.

13. Developer Relations

When working directly on your own code or on code which is already well established as your responsibility, then there is probably little need to check with other committers before jumping in with a commit. When working on a bug in an area of the system which is clearly orphaned (and there are a few such areas, to our shame), the same applies. When modifying parts of the system which are maintained, formally or informally, consider asking for a review just as a developer would have before becoming a committer. For ports, contact the listed `MAINTAINER` in the Makefile.

To determine if an area of the tree is maintained, check the MAINTAINERS file at the root of the tree. If nobody is listed, scan the revision history to see who has committed changes in the past. To list the names and email addresses of all commit authors for a given file in the last 2 years and the number of commits each has authored, ordered by descending number of commits, use:

```
% git -C /path/to/repo shortlog -sne --since="2 years" -- relative/path/to/file
```

If queries go unanswered or the committer otherwise indicates a lack of interest in the area affected, go ahead and commit it.



Avoid sending private emails to maintainers. Other people might be interested in the conversation, not just the final output.

If there is any doubt about a commit for any reason at all, have it reviewed before committing. Better to have it flamed then and there rather than when it is part of the repository. If a commit does results in controversy erupting, it may be advisable to consider backing the change out again until the matter is settled. Remember, with a version control system we can always change it back.

Do not impugn the intentions of others. If they see a different solution to a problem, or even a different problem, it is probably not because they are stupid, because they have questionable parentage, or because they are trying to destroy hard work, personal image, or FreeBSD, but basically because they have a different outlook on the world. Different is good.

Disagree honestly. Argue your position from its merits, be honest about any shortcomings it may have, and be open to seeing their solution, or even their vision of the problem, with an open mind.

Accept correction. We are all fallible. When you have made a mistake, apologize and get on with life. Do not beat up yourself, and certainly do not beat up others for your mistake. Do not waste time on embarrassment or recrimination, just fix the problem and move on.

Ask for help. Seek out (and give) peer reviews. One of the ways open source software is supposed to excel is in the number of eyeballs applied to it; this does not apply if nobody will review code.

14. If in Doubt...

When unsure about something, whether it be a technical issue or a project convention be sure to ask. If you stay silent you will never make progress.

If it relates to a technical issue ask on the public mailing lists. Avoid the temptation to email the individual person that knows the answer. This way everyone will be able to learn from the question and the answer.

For project specific or administrative questions ask, in order:

- Your mentor or former mentor.
- An experienced committer on IRC, email, etc.
- Any team with a "hat", as they can give you a definitive answer.
- If still not sure, ask on FreeBSD developers mailing list.

Once your question is answered, if no one pointed you to documentation that spelled out the answer to your question, document it, as others will have the same question.

15. Bugzilla

The FreeBSD Project utilizes Bugzilla for tracking bugs and change requests. If you commit a fix or suggestion found in the PR database, be sure to close the PR. It is also considered nice if you take time to close any other PRs associated with your commits.

Committers with non-`FreeBSD.org` Bugzilla accounts can have the old account merged with the `FreeBSD.org` account by following these steps:

1. Log in using your old account.
2. Open new bug. Choose `Services` as the Product, and `Bug Tracker` as the Component. In bug description list accounts you wish to be merged.
3. Log in using `FreeBSD.org` account and post comment to newly opened bug to confirm ownership. See Kerberos and LDAP web Password for FreeBSD Cluster for more details on how to generate or set a password for your `FreeBSD.org` account.
4. If there are more than two accounts to merge, post comments from each of them.

You can find out more about Bugzilla at:

- FreeBSD Problem Report Handling Guidelines
- https://www.FreeBSD.org/support

16. Phabricator

The FreeBSD Project utilizes Phabricator for code review requests. See the Phabricator wiki page for details.

Please use the `git arc` command provided by `devel/freebsd-git-arc` (install the port or package, then type `git help arc` for documentation) to create and update Phabricator reviews. This will make it easier for others to review and test your patches.

Committers with non-`FreeBSD.org` Phabricator accounts can have the old account renamed to the `FreeBSD.org` account by following these steps:

1. Change your Phabricator account email to your `FreeBSD.org` email.
2. Open new bug on our bug tracker using your `FreeBSD.org` account, see Bugzilla for more information. Choose `Services` as the Product, and `Code Review` as the Component. In bug description request that your Phabricator account be renamed, and provide a link to your Phabricator user. For example, `https://reviews.freebsd.org/p/bob_example.com/`



Phabricator accounts cannot be merged, please do not open a new account.

17. Who's Who

Besides the repository meisters, there are other FreeBSD project members and teams whom you will probably get to know in your role as a committer. Briefly, and by no means all-inclusively, these are:

`Documentation Engineering Team <doceng@FreeBSD.org>`

doceng is the group responsible for the documentation build infrastructure, approving new documentation committers, and ensuring that the FreeBSD website and documentation on the FTP site is up to date with respect to the Subversion tree. It is not a conflict resolution body. The vast majority of documentation related discussion takes place on the FreeBSD documentation project mailing list. More details regarding the doceng team can be found in its charter. Committers interested in contributing to the documentation should familiarize themselves with the Documentation Project Primer.

`Sergio Carlavilla Delgado <carlavilla@FreeBSD.org>, Dave Cottlehuber <dch@FreeBSD.org>, Marc Fonvieille <blackend@FreeBSD.org>, Craig Leres <leres@FreeBSD.org>, Xin Li <delphij@FreeBSD.org>, Ed Maste <emaste@FreeBSD.org>, Colin Percival <cperciva@FreeBSD.org>, Muhammad Moinur Rahman <bofh@FreeBSD.org>, Vladlen Popolitov <vladlen@FreeBSD.org>, Lexi Winter <ivy@FreeBSD.org>, Alexander Ziaee <ziaee@FreeBSD.org>`

These are the members of the `Release Engineering Team <re@FreeBSD.org>`. This team is responsible for setting release deadlines and controlling the release process. During code freezes, the release engineers have final authority on all changes to the system for whichever branch is pending release status. If there is something you want merged from FreeBSD-CURRENT to FreeBSD-STABLE (whatever values those may have at any given time), these are the

people to talk to about it.

`Gordon Tetlow <gordon@FreeBSD.org>`

`Gordon Tetlow` is the FreeBSD Security Officer and oversees the `Security Officer Team <security-officer@FreeBSD.org>`.

FreeBSD committer's mailing list

{dev-src-all}, {dev-ports-all} and {dev-doc-all} are the mailing lists that the version control system uses to send commit messages to. *Never* send email directly to these lists. Only send replies to this list when they are short and are directly related to a commit.

FreeBSD developers mailing list

All committers are subscribed to -developers. This list was created to be a forum for the committers "community" issues. Examples are Core voting, announcements, etc.

The FreeBSD developers mailing list is for the exclusive use of FreeBSD committers. To develop FreeBSD, committers must have the ability to openly discuss matters that will be resolved before they are publicly announced. Frank discussions of work in progress are not suitable for open publication and may harm FreeBSD.

All FreeBSD committers are expected not to not publish or forward messages from the FreeBSD developers mailing list outside the list membership without permission of all of the authors. Violators will be removed from the FreeBSD developers mailing list, resulting in a suspension of commit privileges. Repeated or flagrant violations may result in permanent revocation of commit privileges.

This list is *not* intended as a place for code reviews or for any technical discussion. In fact using it as such hurts the FreeBSD Project as it gives a sense of a closed list where general decisions affecting all of the FreeBSD using community are made without being "open". Last, but not least *never, never ever, email the FreeBSD developers mailing list and CC:/BCC: another FreeBSD list.* Never, ever email another FreeBSD email list and CC:/BCC: the FreeBSD developers mailing list. Doing so can greatly diminish the benefits of this list.

18. SSH Quick-Start Guide

1. If you do not wish to type your password in every time you use ssh(1), and you use keys to authenticate, ssh-agent(1) is there for your convenience. If you want to use ssh-agent(1), make sure that you run it before running other applications. X users, for example, usually do this from their .xsession or .xinitrc. See ssh-agent(1) for details.
2. Generate a key pair using ssh-keygen(1). The key pair will wind up in your $HOME/.ssh/ directory.

   

   Only ECDSA, Ed25519 or RSA keys are supported.

3. Send your public key ($HOME/.ssh/id_ecdsa.pub, $HOME/.ssh/id_ed25519.pub, or $HOME/.ssh/id_rsa.pub) to the person setting you up as a committer so it can be put into yourlogin in /etc/ssh-keys/ on `freefall`.

Now ssh-add(1) can be used for authentication once per session. It prompts for the private key's pass phrase, and then stores it in the authentication agent (ssh-agent(1)). Use `ssh-add -d` to remove keys stored in the agent.

Test with a simple remote command: `ssh freefall.FreeBSD.org ls /usr`.

For more information, see security/openssh-portable, ssh(1), ssh-add(1), ssh-agent(1), ssh-keygen(1), and scp(1).

For information on adding, changing, or removing ssh(1) keys, see this article.

19. Coverity® Availability for FreeBSD Committers

All FreeBSD developers can obtain access to Coverity analysis results of all FreeBSD Project software. All who are interested in obtaining access to the analysis results of the automated Coverity runs, can sign up at Coverity Scan.

The FreeBSD wiki includes a mini-guide for developers who are interested in working with the Coverity® analysis reports: https://wiki.freebsd.org/CoverityPrevent. Please note that this mini-guide is only readable by FreeBSD developers, so if you cannot access this page, you will have to ask someone to add you to the appropriate Wiki access list.

Finally, all FreeBSD developers who are going to use Coverity® are always encouraged to ask for more details and usage information, by posting any questions to the mailing list of the FreeBSD developers.

20. The FreeBSD Committers' Big List of Rules

Everyone involved with the FreeBSD project is expected to abide by the *Code of Conduct* available from https://www.FreeBSD.org/internal/code-of-conduct. As committers, you form the public face of the project, and how you behave has a vital impact on the public perception of it. This guide expands on the parts of the *Code of Conduct* specific to committers.

1. Respect other committers.
2. Respect other contributors.
3. Discuss any significant change *before* committing.
4. Respect existing maintainers (if listed in the `MAINTAINER` field in Makefile or in MAINTAINER in the top-level directory).
5. Any disputed change must be backed out pending resolution of the dispute if requested by a maintainer. Security related changes may override a maintainer's wishes at the Security Officer's discretion.
6. Changes go to FreeBSD-CURRENT before FreeBSD-STABLE unless specifically permitted by the

release engineer or unless they are not applicable to FreeBSD-CURRENT. Any non-trivial or non-urgent change which is applicable should also be allowed to sit in FreeBSD-CURRENT for at least 3 days before merging so that it can be given sufficient testing. The release engineer has the same authority over the FreeBSD-STABLE branch as outlined for the maintainer in rule #5.

7. Do not fight in public with other committers; it looks bad.
8. Respect all code freezes and read the `committers` and `developers` mailing lists in a timely manner so you know when a code freeze is in effect.
9. When in doubt on any procedure, ask first!
10. Test your changes before committing them.
11. Do not commit to contributed software without *explicit* approval from the respective maintainers.

As noted, breaking some of these rules can be grounds for suspension or, upon repeated offense, permanent removal of commit privileges. Individual members of core have the power to temporarily suspend commit privileges until core as a whole has the chance to review the issue. In case of an "emergency" (a committer doing damage to the repository), a temporary suspension may also be done by the repository meisters. Only a 2/3 majority of core has the authority to suspend commit privileges for longer than a week or to remove them permanently. This rule does not exist to set core up as a bunch of cruel dictators who can dispose of committers as casually as empty soda cans, but to give the project a kind of safety fuse. If someone is out of control, it is important to be able to deal with this immediately rather than be paralyzed by debate. In all cases, a committer whose privileges are suspended or revoked is entitled to a "hearing" by core, the total duration of the suspension being determined at that time. A committer whose privileges are suspended may also request a review of the decision after 30 days and every 30 days thereafter (unless the total suspension period is less than 30 days). A committer whose privileges have been revoked entirely may request a review after a period of 6 months has elapsed. This review policy is *strictly informal* and, in all cases, core reserves the right to either act on or disregard requests for review if they feel their original decision to be the right one.

In all other aspects of project operation, core is a subset of committers and is bound by the *same rules*. Just because someone is in core this does not mean that they have special dispensation to step outside any of the lines painted here; core's "special powers" only kick in when it acts as a group, not on an individual basis. As individuals, the core team members are all committers first and core second.

20.1. Details

1. Respect other committers.

   This means that you need to treat other committers as the peer-group developers that they are. Despite our occasional attempts to prove the contrary, one does not get to be a committer by being stupid and nothing rankles more than being treated that way by one of your peers. Whether we always feel respect for one another or not (and everyone has off days), we still have to *treat* other committers with respect at all times, on public forums and in private email.

   Being able to work together long term is this project's greatest asset, one far more important

than any set of changes to the code, and turning arguments about code into issues that affect our long-term ability to work harmoniously together is just not worth the trade-off by any conceivable stretch of the imagination.

To comply with this rule, do not send email when you are angry or otherwise behave in a manner which is likely to strike others as needlessly confrontational. First calm down, then think about how to communicate in the most effective fashion for convincing the other persons that your side of the argument is correct, do not just blow off some steam so you can feel better in the short term at the cost of a long-term flame war. Not only is this very bad "energy economics", but repeated displays of public aggression which impair our ability to work well together will be dealt with severely by the project leadership and may result in suspension or termination of your commit privileges. The project leadership will take into account both public and private communications brought before it. It will not seek the disclosure of private communications, but it will take it into account if it is volunteered by the committers involved in the complaint.

All of this is never an option which the project's leadership enjoys in the slightest, but unity comes first. No amount of code or good advice is worth trading that away.

2. Respect other contributors.

   You were not always a committer. At one time you were a contributor. Remember that at all times. Remember what it was like trying to get help and attention. Do not forget that your work as a contributor was very important to you. Remember what it was like. Do not discourage, belittle, or demean contributors. Treat them with respect. They are our committers in waiting. They are every bit as important to the project as committers. Their contributions are as valid and as important as your own. After all, you made many contributions before you became a committer. Always remember that.

   Consider the points raised under Respect other committers and apply them also to contributors.

3. Discuss any significant change *before* committing.

   The repository is not where changes are initially submitted for correctness or argued over, that happens first in the mailing lists or by use of the Phabricator service. The commit will only happen once something resembling consensus has been reached. This does not mean that permission is required before correcting every obvious syntax error or manual page misspelling, just that it is good to develop a feel for when a proposed change is not quite such a no-brainer and requires some feedback first. People really do not mind sweeping changes if the result is something clearly better than what they had before, they just do not like being *surprised* by those changes. The very best way of making sure that things are on the right track is to have code reviewed by one or more other committers.

   When in doubt, ask for review!

4. Respect existing maintainers if listed.

   Many parts of FreeBSD are not "owned" in the sense that any specific individual will jump up and yell if you commit a change to "their" area, but it still pays to check first. One convention we use is to put a maintainer line in the Makefile for any package or subtree which is being actively

maintained by one or more people; see Source Tree Guidelines and Policies for documentation on this. Where sections of code have several maintainers, commits to affected areas by one maintainer need to be reviewed by at least one other maintainer. In cases where the "maintainer-ship" of something is not clear, look at the repository logs for the files in question and see if someone has been working recently or predominantly in that area.

5. Any disputed change must be backed out pending resolution of the dispute if requested by a maintainer. Security related changes may override a maintainer's wishes at the Security Officer's discretion.

   This may be hard to swallow in times of conflict (when each side is convinced that they are in the right, of course) but a version control system makes it unnecessary to have an ongoing dispute raging when it is far easier to simply reverse the disputed change, get everyone calmed down again and then try to figure out what is the best way to proceed. If the change turns out to be the best thing after all, it can be easily brought back. If it turns out not to be, then the users did not have to live with the bogus change in the tree while everyone was busily debating its merits. People *very* rarely call for back-outs in the repository since discussion generally exposes bad or controversial changes before the commit even happens, but on such rare occasions the back-out should be done without argument so that we can get immediately on to the topic of figuring out whether it was bogus or not.

6. Changes go to FreeBSD-CURRENT before FreeBSD-STABLE unless specifically permitted by the release engineer or unless they are not applicable to FreeBSD-CURRENT. Any non-trivial or non-urgent change which is applicable should also be allowed to sit in FreeBSD-CURRENT for at least 3 days before merging so that it can be given sufficient testing. The release engineer has the same authority over the FreeBSD-STABLE branch as outlined in rule #5.

   This is another "do not argue about it" issue since it is the release engineer who is ultimately responsible (and gets beaten up) if a change turns out to be bad. Please respect this and give the release engineer your full cooperation when it comes to the FreeBSD-STABLE branch. The management of FreeBSD-STABLE may frequently seem to be overly conservative to the casual observer, but also bear in mind the fact that conservatism is supposed to be the hallmark of FreeBSD-STABLE and different rules apply there than in FreeBSD-CURRENT. There is also really no point in having FreeBSD-CURRENT be a testing ground if changes are merged over to FreeBSD-STABLE immediately. Changes need a chance to be tested by the FreeBSD-CURRENT developers, so allow some time to elapse before merging unless the FreeBSD-STABLE fix is critical, time sensitive or so obvious as to make further testing unnecessary (spelling fixes to manual pages, obvious bug/typo fixes, etc.) In other words, apply common sense.

   Changes to the security branches (for example, `releng/9.3`) must be approved by a member of the `Security Officer Team <security-officer@FreeBSD.org>`, or in some cases, by a member of the `Release Engineering Team <re@FreeBSD.org>`.

7. Do not fight in public with other committers; it looks bad.

   This project has a public image to uphold and that image is very important to all of us, especially if we are to continue to attract new members. There will be occasions when, despite everyone's very best attempts at self-control, tempers are lost and angry words are exchanged. The best thing that can be done in such cases is to minimize the effects of this until everyone

has cooled back down. Do not air angry words in public and do not forward private correspondence or other private communications to public mailing lists, mail aliases, instant messaging channels or social media sites. What people say one-to-one is often much less sugar-coated than what they would say in public, and such communications therefore have no place there - they only serve to inflame an already bad situation. If the person sending a flame-o-gram at least had the grace to send it privately, then have the grace to keep it private yourself. If you feel you are being unfairly treated by another developer, and it is causing you anguish, bring the matter up with core rather than taking it public. Core will do its best to play peace makers and get things back to sanity. In cases where the dispute involves a change to the codebase and the participants do not appear to be reaching an amicable agreement, core may appoint a mutually-agreeable third party to resolve the dispute. All parties involved must then agree to be bound by the decision reached by this third party.

8. Respect all code freezes and read the `committers` and `developers` mailing list on a timely basis so you know when a code freeze is in effect.

   Committing unapproved changes during a code freeze is a really big mistake and committers are expected to keep up-to-date on what is going on before jumping in after a long absence and committing 10 megabytes worth of accumulated stuff. People who abuse this on a regular basis will have their commit privileges suspended until they get back from the FreeBSD Happy Reeducation Camp we run in Greenland.

9. When in doubt on any procedure, ask first!

   Many mistakes are made because someone is in a hurry and just assumes they know the right way of doing something. If you have not done it before, chances are good that you do not actually know the way we do things and really need to ask first or you are going to completely embarrass yourself in public. There is no shame in asking "how in the heck do I do this?" We already know you are an intelligent person; otherwise, you would not be a committer.

10. Test your changes before committing them.

    If your changes are to the kernel, make sure you can still compile both GENERIC and LINT. If your changes are anywhere else, make sure you can still compile userspace via `make buildworld`. If your changes are to a branch, make sure your testing occurs with a machine which is running that code. If you have a change which also may break another architecture, be sure and test on all supported architectures. Please ensure your change works for supported toolchains. Please refer to the FreeBSD Internal Page for a list of available resources. As other architectures are added to the FreeBSD supported platforms list, the appropriate shared testing resources will be made available.

11. Do not commit to contributed software without *explicit* approval from the respective maintainers.

    Contributed software is anything under the src/contrib, src/crypto, or src/sys/contrib trees.

    The trees mentioned above are for contributed software usually imported onto a vendor branch. Committing something there may cause unnecessary headaches when importing newer versions of the software. As a general consider sending patches upstream to the vendor. Patches may be committed to FreeBSD first with permission of the maintainer.

Reasons for modifying upstream software range from wanting strict control over a tightly coupled dependency to lack of portability in the canonical repository's distribution of their code. Regardless of the reason, effort to minimize the maintenance burden of fork is helpful to fellow maintainers. Avoid committing trivial or cosmetic changes to files since it makes every merge thereafter more difficult: such patches need to be manually re-verified every import.

If a particular piece of software lacks a maintainer, you are encouraged to take up ownership. If you are unsure of the current maintainership email FreeBSD architecture and design mailing list and ask.

20.2. Policy on Multiple Architectures

In an effort to make it easier to keep FreeBSD portable across the platforms we support, core has developed this mandate:

Major design work (including major API and ABI changes) must prove itself on at least one Tier 1 platform before it may be committed to the source tree.

Developers should also be aware of our Tier Policy for the long term support of hardware architectures. The rules here are intended to provide guidance during the development process, and are distinct from the requirements for features and architectures listed in that section. The Tier rules for feature support on architectures at release-time are more strict than the rules for changes during the development process.

20.3. Policy on Multiple Compilers

The FreeBSD base system builds with both Clang and GCC. The project does this in a careful and controlled way to maximize benefits from this extra work, while keeping the extra work to a minimum. Supporting both Clang and GCC improves the flexibility our users have. These compilers have different strengths and weaknesses, and supporting both allows users to pick the best one for their needs. Clang and GCC support similar dialects of C and C++, necessitating a relatively small amount of conditional code. The project gains increased code coverage and improves the code quality by using features from both compilers. The project is able to build in more user environments and leverage more CI environments by supporting this range, increasing convenience for users and giving them more tools to test with. By carefully constraining the range of versions supported to modern versions of these compilers, the project avoids unduly increasing the testing matrix. Older and obscure compilers, as well as older dialects of the languages, have extremely limited support that allow user programs to build with them, but without constraining the base system to being built with them. The exact balance continues to evolve to ensure the benefits of extra work remain greater than the burdens it imposes. The project used to support really old Intel compilers or old GCC versions, but we traded supporting those obsolete compilers for a carefully selected range of modern compilers. This section documents where we use different compilers, and the expectations around that.

The FreeBSD base system includes an in-tree Clang compiler. Due to being in the tree, this compiler is the most supported compiler. All changes must compile with it, prior to commit. Complete testing, as appropriate for the change, should be done with this compiler.

The FreeBSD base system also supports various versions of Clang and GCC as out-of-tree compilers. For large or risky changes, committers should do a test build with a supported version of GCC. Out of tree compilers are available as packages. GCC compilers are available as `${TARGET_ARCH}-gcc${VERSION}` packages, such as devel/freebsd-gcc14@aarch64. Clang compilers are available as `llvm${VERSION}` packages, such as devel/llvm18. The project runs automated CI jobs to build everything with these compilers. Committers are expected to fix the jobs they break with their changes. Committers may test builds of userspace or individual kernels by setting `CROSS_TOOLCHAIN` to the package name, for example `CROSS_TOOLCHAIN=aarch64-gcc14` or `CROSS_TOOLCHAIN=llvm18`. For universe or tinderbox builds, `USE_GCC_TOOLCHAINS=gcc${VERSION}` builds all architectures using the appropriate GCC compiler packages. For universe or tinderbox builds using an out-of-tree Clang, pass `CROSS_TOOLCHAIN=llvm${VERSION}`. Note that while all architectures in the base system can be compiled by Clang, only a few architectures can be fully built by GCC.

The FreeBSD project also has some CI pipelines on github. For pull requests on github and some branches pushed to github forks, a number of cross compilation jobs run. These test FreeBSD building using versions of Clang that lag the in-tree compiler by one or more major versions.

The FreeBSD project is also upgrading compilers. Both Clang and GCC are fast moving targets. Some work to change things in the tree, for example removing the old-style K&R function declarations and definitions, will land in the tree prior to the compiler landing. Committers should try to be mindful about this and be receptive to looking into problems with their code or changes with these new compilers. Also, just after a new compiler version hits the tree, people may need to compile things with the old version if there was an undetected regression suspected.

In addition to the compiler, LLVM's LLD and GNU's binutils are used indirectly by the compiler. Committers should be mindful of variations in assembler syntax and features of the linkers and ensure both variants work. These components will be tested as part of FreeBSD's CI jobs for Clang or GCC.

The FreeBSD project provides headers and libraries that allow other compilers to be used to build software not in the base system. These headers have support for making the environment as strict as the standard, supporting prior dialects of ANSI-C back to C89, and other edge cases our large ports collection has uncovered. This support constrains retirement of older standards in places like header files, but does not constrain updating the base system to newer dialects. Nor does it require the base system to compile with these older standards as a whole. Breaking this support will cause packages in the ports collection to fail, so should be avoided where possible, and promptly fixed when it is easy to do so.

The FreeBSD build system currently accommodates these different environments. As new warnings are added to compilers, the project tries to fix them. However, sometimes these warnings require extensive rework, so are suppressed in some way by using make variables that evaluate to the proper thing depending on the compiler version. Developers should be mindful of this, and ensure any compiler specific flags are properly conditionalized.

20.3.1. Current Compiler Versions

The versions of supported compilers for a given branch such as `main` or `stable/X` varies over time. The authoritative source for supported compiler versions are automated CI jobs tested in GitHub's cross-build actions and Jenkins.

Branch	In-tree Compiler	llvm12	llvm13	llvm14	llvm15	llvm18	amd64-gcc12	amd64-gcc13	amd64-gcc14	amd64-gcc15	riscv64-gcc15
main	llvm 19				Y	Y	Y	Y	Y	Y	Y
stable/15	llvm 19			Y		Y	Y	Y	Y		
stable/14	llvm 19	Y	Y	Y			Y		Y		
stable/13	llvm 19	Y	Y	Y			Y		Y		

GCC toolchains are tested for amd64 and riscv64 via CI jobs in Jenkins. LLVM toolchains are tested for aarch64 and amd64 in GitHub's cross-build actions.

20.4. Other Suggestions

When committing documentation changes, use a spell checker before committing. For all XML docs, verify that the formatting directives are correct by running `make lint` and textproc/igor.

For manual pages, run sysutils/manck and textproc/igor over the manual page to verify all of the cross references and file references are correct and that the man page has all of the appropriate `MLINKS` installed.

Do not mix style fixes with new functionality. A style fix is any change which does not modify the functionality of the code. Mixing the changes obfuscates the functionality change when asking for differences between revisions, which can hide any new bugs. Do not include whitespace changes with content changes in commits to doc/. The extra clutter in the diffs makes the translators' job much more difficult. Instead, make any style or whitespace changes in separate commits that are clearly labeled as such in the commit message.

20.5. Deprecating Features

When it is necessary to remove functionality from software in the base system, follow these guidelines whenever possible:

1. Mention is made in the manual page and possibly the release notes that the option, utility, or interface is deprecated. Use of the deprecated feature generates a warning.
2. The option, utility, or interface is preserved until the next major (point zero) release.
3. The option, utility, or interface is removed and no longer documented. It is now obsolete. It is also generally a good idea to note its removal in the release notes.

20.6. Privacy and Confidentiality

1. Most FreeBSD business is done in public.

FreeBSD is an *open* project. Which means that not only can anyone use the source code, but that most of the development process is open to public scrutiny.

2. Certain sensitive matters must remain private or held under embargo.

   There unfortunately cannot be complete transparency. As a FreeBSD developer you will have a certain degree of privileged access to information. Consequently you are expected to respect certain requirements for confidentiality. Sometimes the need for confidentiality comes from external collaborators or has a specific time limit. Mostly though, it is a matter of not releasing private communications.

3. The Security Officer has sole control over the release of security advisories.

   Where there are security problems that affect many different operating systems, FreeBSD frequently depends on early access to be able to prepare advisories for coordinated release. Unless FreeBSD developers can be trusted to maintain security, such early access will not be made available. The Security Officer is responsible for controlling pre-release access to information about vulnerabilities, and for timing the release of all advisories. He may request help under condition of confidentiality from any developer with relevant knowledge to prepare security fixes.

4. Communications with Core are kept confidential for as long as necessary.

   Communications to core will initially be treated as confidential. Eventually however, most of Core's business will be summarized into the monthly or quarterly core reports. Care will be taken to avoid publicising any sensitive details. Records of some particularly sensitive subjects may not be reported on at all and will be retained only in Core's private archives.

5. Non-disclosure Agreements may be required for access to certain commercially sensitive data.

   Access to certain commercially sensitive data may only be available under a Non-Disclosure Agreement. The FreeBSD Foundation legal staff must be consulted before any binding agreements are entered into.

6. Private communications must not be made public without permission.

   Beyond the specific requirements above there is a general expectation not to publish private communications between developers without the consent of all parties involved. Ask permission before forwarding a message onto a public mailing list, or posting it to a forum or website that can be accessed by other than the original correspondents.

7. Communications on project-only or restricted access channels must be kept private.

   Similarly to personal communications, certain internal communications channels, including FreeBSD Committer only mailing lists and restricted access IRC channels are considered private communications. Permission is required to publish material from these sources.

8. Core may approve publication.

   Where it is impractical to obtain permission due to the number of correspondents or where permission to publish is unreasonably withheld, Core may approve release of such private

matters that merit more general publication.

21. Support for Multiple Architectures

FreeBSD is a highly portable operating system intended to function on many different types of hardware architectures. Maintaining clean separation of Machine Dependent (MD) and Machine Independent (MI) code, as well as minimizing MD code, is an important part of our strategy to remain agile with regards to current hardware trends. Each new hardware architecture supported by FreeBSD adds substantially to the cost of code maintenance, toolchain support, and release engineering. It also dramatically increases the cost of effective testing of kernel changes. As such, there is strong motivation to differentiate between classes of support for various architectures while remaining strong in a few key architectures that are seen as the FreeBSD "target audience".

21.1. Statement of General Intent

The FreeBSD Project targets "production quality commercial off-the-shelf (COTS) workstation, server, and high-end embedded systems". By retaining a focus on a narrow set of architectures of interest in these environments, the FreeBSD Project is able to maintain high levels of quality, stability, and performance, as well as minimize the load on various support teams on the project, such as the ports team, documentation team, security officer, and release engineering teams. Diversity in hardware support broadens the options for FreeBSD consumers by offering new features and usage opportunities, but these benefits must always be carefully considered in terms of the real-world maintenance cost associated with additional platform support.

The FreeBSD Project differentiates platform targets into four tiers. Each tier includes a list of guarantees consumers may rely on as well as obligations by the Project and developers to fulfill those guarantees. These lists define the minimum guarantees for each tier. The Project and developers may provide additional levels of support beyond the minimum guarantees for a given tier, but such additional support is not guaranteed. Each platform target is assigned to a specific tier for each stable branch. As a result, a platform target might be assigned to different tiers on concurrent stable branches.

21.2. Platform Targets

Support for a hardware platform consists of two components: kernel support and userland Application Binary Interfaces (ABIs). Kernel platform support includes things needed to run a FreeBSD kernel on a hardware platform such as machine-dependent virtual memory management and device drivers. A userland ABI specifies an interface for user processes to interact with a FreeBSD kernel and base system libraries. A userland ABI includes system call interfaces, the layout and semantics of public data structures, and the layout and semantics of arguments passed to subroutines. Some components of an ABI may be defined by specifications such as the layout of C++ exception objects or calling conventions for C functions.

A FreeBSD kernel also uses an ABI (sometimes referred to as the Kernel Binary Interface (KBI)) which includes the semantics and layouts of public data structures and the layout and semantics of arguments to public functions within the kernel itself.

A FreeBSD kernel may support multiple userland ABIs. For example, FreeBSD's amd64 kernel supports FreeBSD amd64 and i386 userland ABIs as well as Linux x86_64 and i386 userland ABIs. A FreeBSD kernel should support a "native" ABI as the default ABI. The native "ABI" generally shares certain properties with the kernel ABI such as the C calling convention, sizes of basic types, etc.

Tiers are defined for both kernels and userland ABIs. In the common case, a platform's kernel and FreeBSD ABIs are assigned to the same tier.

21.2.1. Tier 1: Fully-Supported Architectures

Tier 1 platforms are the most mature FreeBSD platforms. They are supported by the security officer, release engineering, and Ports Management Team. Tier 1 architectures are expected to be Production Quality with respect to all aspects of the FreeBSD operating system, including installation and development environments.

The FreeBSD Project provides the following guarantees to consumers of Tier 1 platforms:

- Official FreeBSD release images will be provided by the release engineering team.
- Binary updates and source patches for Security Advisories and Errata Notices will be provided for supported releases.
- Source patches for Security Advisories will be provided for supported branches.
- Binary updates and source patches for cross-platform Security Advisories will typically be provided at the time of the announcement.
- Changes to userland ABIs will generally include compatibility shims to ensure correct operation of binaries compiled against any stable branch where the platform is Tier 1. These shims might not be enabled in the default install. If compatibility shims are not provided for an ABI change, the lack of shims will be clearly documented in the release notes.
- Changes to certain portions of the kernel ABI will include compatibility shims to ensure correct operation of kernel modules compiled against the oldest supported release on the branch. Note that not all parts of the kernel ABI are protected.
- Official binary packages for third party software will be provided by the ports team. For embedded architectures, these packages may be cross-built from a different architecture.
- Most relevant ports should either build or have the appropriate filters to prevent inappropriate ones from building.
- New features which are not inherently platform-specific will be fully functional on all Tier 1 architectures.
- Features and compatibility shims used by binaries compiled against older stable branches may be removed in newer major versions. Such removals will be clearly documented in the release notes.
- Tier 1 platforms should be fully documented. Basic operations will be documented in the FreeBSD Handbook.
- Tier 1 platforms will be included in the source tree.
- Tier 1 platforms should be self-hosting either via the in-tree toolchain or an external toolchain. If an external toolchain is required, official binary packages for an external toolchain will be

provided.

To maintain maturity of Tier 1 platforms, the FreeBSD Project will maintain the following resources to support development:

- Build and test automation support either in the FreeBSD.org cluster or some other location easily available for all developers. Embedded platforms may substitute an emulator available in the FreeBSD.org cluster for actual hardware.
- Inclusion in the `make universe` and `make tinderbox` targets.
- Dedicated hardware in one of the FreeBSD clusters for package building (either natively or via qemu-user).

Collectively, developers are required to provide the following to maintain the Tier 1 status of a platform:

- Changes to the source tree should not knowingly break the build of a Tier 1 platform.
- Tier 1 architectures must have a mature, healthy ecosystem of users and active developers.
- Developers should be able to build packages on commonly available, non-embedded Tier 1 systems. This can mean either native builds if non-embedded systems are commonly available for the platform in question, or it can mean cross-builds hosted on some other Tier 1 architecture.
- Changes cannot break the userland ABI. If an ABI change is required, ABI compatibility for existing binaries should be provided via use of symbol versioning or shared library version bumps.
- Changes merged to stable branches cannot break the protected portions of the kernel ABI. If a kernel ABI change is required, the change should be modified to preserve functionality of existing kernel modules.

21.2.2. Tier 2: Developmental and Niche Architectures

Tier 2 platforms are functional, but less mature FreeBSD platforms. They are not supported by the security officer, release engineering, and Ports Management Team.

Tier 2 platforms may be Tier 1 platform candidates that are still under active development. Architectures reaching end of life may also be moved from Tier 1 status to Tier 2 status as the availability of resources to continue to maintain the system in a Production Quality state diminishes. Well-supported niche architectures may also be Tier 2.

The FreeBSD Project provides the following guarantees to consumers of Tier 2 platforms:

- The ports infrastructure should include basic support for Tier 2 architectures sufficient to support building ports and packages. This includes support for basic packages such as ports-mgmt/pkg, but there is no guarantee that arbitrary ports will be buildable or functional.
- New features which are not inherently platform-specific should be feasible on all Tier 2 architectures if not implemented.
- Tier 2 platforms will be included in the source tree.

- Tier 2 platforms should be self-hosting either via the in-tree toolchain or an external toolchain. If an external toolchain is required, official binary packages for an external toolchain will be provided.
- Tier 2 platforms should provide functional kernels and userlands even if an official release distribution is not provided.

To maintain maturity of Tier 2 platforms, the FreeBSD Project will maintain the following resources to support development:

- Inclusion in the `make universe` and `make tinderbox` targets.

Collectively, developers are required to provide the following to maintain the Tier 2 status of a platform:

- Changes to the source tree should not knowingly break the build of a Tier 2 platform.
- Tier 2 architectures must have an active ecosystem of users and developers.
- While changes are permitted to break the userland ABI, the ABI should not be broken gratuitously. Significant userland ABI changes should be restricted to major versions.
- New features that are not yet implemented on Tier 2 architectures should provide a means of disabling them on those architectures.

21.2.3. Tier 3: Experimental Architectures

Tier 3 platforms have at least partial FreeBSD support. They are *not* supported by the security officer, release engineering, and Ports Management Team.

Tier 3 platforms are architectures in the early stages of development, for non-mainstream hardware platforms, or which are considered legacy systems unlikely to see broad future use. Initial support for Tier 3 platforms may exist in a separate repository rather than the main source repository.

The FreeBSD Project provides no guarantees to consumers of Tier 3 platforms and is not committed to maintaining resources to support development. Tier 3 platforms may not always be buildable, nor are any kernel or userland ABIs considered stable.

21.2.4. Unsupported Architectures

Other platforms are not supported in any form by the project. The project previously described these as Tier 4 systems.

After a platform transitions to unsupported, all support for the platform is removed from the source, ports and documentation trees. Note that ports support should remain as long as the platform is supported in a branch supported by ports.

21.3. Policy on Changing the Tier of an Architecture

Systems may only be moved from one tier to another by approval of the FreeBSD Core Team, which shall make that decision in collaboration with the Security Officer, Release Engineering, and ports

management teams. For a platform to be promoted to a higher tier, any missing support guarantees must be satisfied before the promotion is completed.

22. Ports Specific FAQ

22.1. Adding a New Port

22.1.1. How do I add a new port?

Adding a port to the tree is relatively simple. Once the port is ready to be added, as explained later here, you need to add the port's directory entry in the category's Makefile. In this Makefile, ports are listed in alphabetical order and added to the `SUBDIR` variable, like this:

```
    SUBDIR += newport
```

Once the port and its category's Makefile are ready, the new port can be committed:

```
% git add category/Makefile category/newport
% git commit
% git push
```



Don't forget to setup git hooks for the ports tree as explained here; a specific hook has been developed to verify the category's Makefile.

22.1.2. Any other things I need to know when I add a new port?

Check the port, preferably to make sure it compiles and packages correctly.

The Porters Handbook's Testing Chapter contains more detailed instructions. See the Portclippy / Portfmt and the poudriere sections.

You do not necessarily have to eliminate all warnings but make sure you have fixed the simple ones.

If the port came from a submitter who has not contributed to the Project before, add that person's name to the Additional Contributors section of the FreeBSD Contributors List.

Close the PR if the port came in as a PR. To close a PR, change the state to `Issue Resolved` and the resolution as `Fixed`.



If for some reason using poudriere to test the new port is not possible, the bare minimum of testing includes this sequence:

```
# make install
# make package
```

```
# make deinstall
# pkg add package you built above
# make deinstall
# make reinstall
# make package
```

Note that poudriere is the reference for package building, it the port does not build in poudriere, it will be removed.

22.2. Removing an Existing Port

22.2.1. How do I remove an existing port?

First, please read the section about repository copies. Before you remove the port, you have to verify there are no other ports depending on it.

- Make sure there is no dependency on the port in the ports collection:
  - The port's PKGNAME appears in exactly one line in a recent INDEX file.
  - No other ports contains any reference to the port's directory or PKGNAME in their Makefiles



When using Git, consider using git-grep(1), it is much faster than `grep -r`.

- Then, remove the port:
  - Remove the port's files and directory with `git rm`.
  - Remove the `SUBDIR` listing of the port in the parent directory Makefile.
  - Add an entry to ports/MOVED.
  - Remove the port from ports/LEGAL if it is there.

Alternatively, you can use the rmport script, from ports/Tools/scripts. This script was written by Vasil Dimov <vd@FreeBSD.org>. When sending questions about this script to the FreeBSD ports mailing list, please also CC Chris Rees <crees@FreeBSD.org>, the current maintainer.

22.3. How do I move a port to a new location?

1. Perform a thorough check of the ports collection for any dependencies on the old port location/name, and update them. Running `grep` on INDEX is not enough because some ports have dependencies enabled by compile-time options. A full git-grep(1) of the ports collection is recommended.
2. Remove the `SUBDIR` entry from the old category Makefile and add a `SUBDIR` entry to the new category Makefile.
3. Add an entry to ports/MOVED.

4. Search for entries in xml files inside ports/security/vuxml and adjust them accordingly. In particular, check for previous packages with the new name which version could include the new port.
5. Move the port with `git mv`.
6. Commit the changes.

22.4. How do I copy a port to a new location?

1. Copy port with `cp -R old-cat/old-port new-cat/new-port`.
2. Add the new port to the new-cat/Makefile.
3. Change stuff in new-cat/new-port.
4. Commit the changes.

22.5. Ports Freeze

22.5.1. What is a “ports freeze”?

A “ports freeze” was a restricted state the ports tree was put in before a release. It was used to ensure a higher quality for the packages shipped with a release. It usually lasted a couple of weeks. During that time, build problems were fixed, and the release packages were built. This practice is no longer used, as the packages for the releases are built from the current stable, quarterly branch.

For more information on how to merge commits to the quarterly branch, see What is the procedure to request authorization for merging a commit to the quarterly branch?.

22.6. Quarterly Branches

22.6.1. What is the procedure to request authorization for merging a commit to the quarterly branch?

As of November 30, 2020, there is no need to seek explicit approval to commit to the quarterly branch.

22.6.2. What is the procedure for merging commits to the quarterly branch?

Merging commits to the quarterly branch (a process we call MFH for a historical reason) is very similar to MFC’ing a commit in the src repository, so basically:

```
% git checkout 2021Q2
% git cherry-pick -x $HASH
(verify everything is OK, for example by doing a build test)
% git push
```

where $HASH is the hash of the commit you want to copy over to the quarterly branch. The -x parameter ensures the hash $HASH of the main branch is included in the new commit message of the quarterly branch.

22.7. Creating a New Category

22.7.1. What is the procedure for creating a new category?

Please see Proposing a New Category in the Porter's Handbook. Once that procedure has been followed and the PR has been assigned to the Ports Management Team <portmgr@FreeBSD.org>, it is their decision whether or not to approve it. If they do, it is their responsibility to:

1. Perform any needed moves. (This only applies to physical categories.)
2. Update the VALID_CATEGORIES definition in ports/Mk/bsd.port.mk.
3. Assign the PR back to you.

22.7.2. What do I need to do to implement a new physical category?

1. Upgrade each moved port's Makefile. Do not connect the new category to the build yet.

   To do this, you will need to:

   1. Change the port's CATEGORIES (this was the point of the exercise, remember?) The new category is listed first. This will help to ensure that the PKGORIGIN is correct.
   2. Run a make describe. Since the top-level make index that you will be running in a few steps is an iteration of make describe over the entire ports hierarchy, catching any errors here will save you having to re-run that step later on.
   3. If you want to be really thorough, now might be a good time to run portlint(1).

2. Check that the PKGORIGINs are correct. The ports system uses each port's CATEGORIES entry to create its PKGORIGIN, which is used to connect installed packages to the port directory they were built from. If this entry is wrong, common port tools like pkg-version(8) and portupgrade(1) fail.

   To do this, use the chkorigin.sh tool: env PORTSDIR=/path/to/ports sh -e /path/to/ports/Tools/scripts/chkorigin.sh. This will check every port in the ports tree, even those not connected to the build, so you can run it directly after the move operation. Hint: do not forget to look at the PKGORIGINs of any slave ports of the ports you just moved!

3. On your own local system, test the proposed changes: first, comment out the SUBDIR entries in the old ports' categories' Makefiles; then enable building the new category in ports/Makefile. Run make checksubdirs in the affected category directories to check the SUBDIR entries. Next, in the ports/ directory, run make index. This can take over 40

minutes on even modern systems; however, it is a necessary step to prevent problems for other people.

4. Once this is done, you can commit the updated ports/Makefile to connect the new category to the build and also commit the Makefile changes for the old category or categories.
5. Add appropriate entries to ports/MOVED.
6. Update the documentation by modifying:
   - the list of categories in the Porter's Handbook
7. Only once all the above have been done, and no one is any longer reporting problems with the new ports, should the old ports be deleted from their previous locations in the repository.

22.7.3. What do I need to do to implement a new virtual category?

This is much simpler than a physical category. Only a few modifications are needed:

- the list of categories in the Porter's Handbook

22.8. Miscellaneous Questions

22.8.1. Are there changes that can be committed without asking the maintainer for approval?

Blanket approval for most ports applies to these types of fixes:

- Most infrastructure changes to a port (that is, modernizing, but not changing the functionality). For example, the blanket covers converting to new `USES` macros, enabling verbose builds, and switching to new ports system syntaxes.
- Trivial and *tested* build and runtime fixes.
- Documentations or metadata changes to ports, like pkg-descr or `COMMENT`.



Exceptions to this are anything maintained by the Ports Management Team <portmgr@FreeBSD.org>, or the Security Officer Team <security-officer@FreeBSD.org>. No unauthorized commits may ever be made to ports maintained by those groups.

22.8.2. How do I know if my port is building correctly or not?

The packages are built multiple times each week. If a port fails, the maintainer will receive an email from `pkg-fallout@FreeBSD.org`.

Reports for all the package builds (official, experimental, and non-regression) are aggregated at pkg-status.FreeBSD.org.

22.8.3. I added a new port. Do I need to add it to the INDEX?

No. The file can either be generated by running `make index`, or a pre-generated version can be downloaded with `make fetchindex`.

22.8.4. Are there any other files I am not allowed to touch?

Any file directly under ports/, or any file under a subdirectory that starts with an uppercase letter (Mk/, Tools/, etc.). In particular, the Ports Management Team <portmgr@FreeBSD.org> is very protective of ports/Mk/bsd.port*.mk so do not commit changes to those files unless you want to face their wrath.

22.8.5. What is the proper procedure for updating the checksum for a port distfile when the file changes without a version change?

When the checksum for a distribution file is updated due to the author updating the file without changing the port revision, the commit message includes a summary of the relevant diffs between the original and new distfile to ensure that the distfile has not been corrupted or maliciously altered. If the current version of the port has been in the ports tree for a while, a copy of the old distfile will usually be available on the ftp servers; otherwise the author or maintainer should be contacted to find out why the distfile has changed.

22.8.6. How can an experimental test build of the ports tree (exp-run) be requested?

An exp-run must be completed before patches with a significant ports impact are committed. The patch can be against the ports tree or the base system.

Full package builds will be done with the patches provided by the submitter, and the submitter is required to fix detected problems *(fallout)* before commit.

1. Go to the Bugzilla new PR page.
2. Select the product your patch is about.
3. Fill in the bug report as normal. Remember to attach the patch.
4. If at the top it says “Show Advanced Fields” click on it. It will now say “Hide Advanced Fields”. Many new fields will be available. If it already says “Hide Advanced Fields”, no need to do anything.
5. In the “Flags” section, set the “exp-run” one to ?. As for all other fields, hovering the mouse over any field shows more details.
6. Submit. Wait for the build to run.
7. Ports Management Team <portmgr@FreeBSD.org> will reply with a possible fallout.
8. Depending on the fallout:
    - If there is no fallout, the procedure stops here, and the change can be committed, pending any other approval required.

i. If there is fallout, it *must* be fixed, either by fixing the ports directly in the ports tree, or adding to the submitted patch.

ii. When this is done, go back to step 6 saying the fallout was fixed and wait for the exp-run to be run again. Repeat as long as there are broken ports.

23. Issues Specific to Developers Who Are Not Committers

A few people who have access to the FreeBSD machines do not have commit bits. Almost all of this document will apply to these developers as well (except things specific to commits and the mailing list memberships that go with them). In particular, we recommend that you read:

- Administrative Details
- For Everyone



Get your mentor to add you to the "Additional Contributors" (doc/shared/contrib-additional.adoc), if you are not already listed there.

- Developer Relations
- SSH Quick-Start Guide
- The FreeBSD Committers' Big List of Rules

24. Information About Analytics

The Project website uses Plausible Analytics to collect anonymized usage statistics regarding usage of the site since 2022.

Previously, Google Analytics was enabled from December 12, 2012 to March 3, 2022.

25. Miscellaneous Questions

25.1. How do I access people.FreeBSD.org to put up personal or project information?

`people.FreeBSD.org` is the same as `freefall.FreeBSD.org`. Just create a public_html directory. Anything you place in that directory will automatically be visible under https://people.FreeBSD.org/.

25.2. Where are the mailing list archives stored?

The mailing lists are archived under /local/mail on `freefall.FreeBSD.org`.

25.3. I would like to mentor a new committer. What process do I need to follow?

See the New Account Creation Procedure document on the internal pages.

26. Benefits and Perks for FreeBSD Committers

26.1. Recognition

Recognition as a competent software engineer is the longest lasting value. In addition, getting a chance to work with some of the best people that every engineer would dream of meeting is a great perk!

26.2. Gandi.net

Gandi provides website hosting, cloud computing, domain registration, and X.509 certificate services.

Gandi offers an E-rate discount to all FreeBSD developers. To streamline the process of getting the discount first set up a Gandi account, fill in the billing information and select the currency. Then send an email to non-profit@gandi.net using your `@freebsd.org` mail address, and indicate your Gandi handle.

26.3. rsync.net

rsync.net provides cloud storage for offsite backup that is optimized for UNIX users. Their service runs entirely on FreeBSD and ZFS.

rsync.net offers a free-forever 500 GB account to FreeBSD developers. Simply sign up at https://www.rsync.net/freebsd.html using your `@freebsd.org` address to receive this free account.

26.4. Watchgoose

Watchgoose provides hosted monitoring for cron jobs and backups, as well as other scheduled and recurring tasks that can send a start or completion signal.

FreeBSD committers are eligible for a complimentary paid plan matched to their monitoring needs. To request access, create a Watchgoose.com account and email support@watchgoose.com from an `@freebsd.org` address, including the email address used (from the FreeBSD.org domain) for the Watchgoose account.

See the Watchgoose Open Source support program for more information.