Participating in gawk Development

Edition 0.74
July, 2022

Arnold D. Robbins

Published by:

Free Software Foundation
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301 USA
Phone: +1-617-542-5942
Fax: +1-617-542-2652
Email: gnu@gnu.org
URL: http://www.gnu.org/

This is Edition 0.74 of *Participating in* `gawk` *Development*.

Table of Contents

Preface

This booklet describes how to participate in development of GNU Awk (`gawk`). GNU Awk is a Free Software project belonging to the Free Software Foundation's GNU project.

Intended Audience

This booklet is aimed at software developers who wish to participate in `gawk` development.

You should be comfortable working with traditional Unix-style command-line tools, and with the C language and standard library facilities.

You should also have some prior experience using distributed source code control systems, such as the Concurrent Versions System (CVS) or Subversion (SVN). Experience with a more modern system such as Mercurial or Git will be even more helpful.

The booklet focuses on participation in the project (that is, how to work most effectively if you wish to contribute to it) and also describes how to make use of the Git distributed source code management system for `gawk` development.

Using This Book

This booklet has the following chapters and appendices:

Typographical Conventions

This booklet is written in Texinfo, the GNU documentation formatting language. A single Texinfo source file is used to produce both the printed and online versions of the documentation. Because of this, the typographical conventions are slightly different than in other books you may have read.

Examples you would type at the command line are preceded by the common shell primary and secondary prompts, '`$`' and '`>`'. Input that you type is shown *`like this`*. Output from the command is preceded by the glyph "⊣". This typically represents the command's standard output. Error messages and other output on the command's standard error are preceded by the glyph " error ". For example:

```
$ echo hi on stdout
⊣ hi on stdout
$ echo hello on stderr 1>&2
error  hello on stderr
```

In the text, almost anything related to programming, such as command names, variable and function names, and string, numeric and regexp constants appear in `this font`. Code fragments appear in the same font and quoted, '`like this`'. Things that are replaced by the user or programmer appear in *this font*. Options look like this: `-f`. File names are indicated like this: `/path/to/ourfile`. Some things are emphasized *like this*, and if a point needs to be made strongly, it is done **like this**. The first occurrence of a new term is usually its *definition* and appears in the same font as the previous occurrence of "definition" in this sentence.

Characters that you type at the keyboard look *`like this`*. In particular, there are special characters called "control characters." These are characters that you type by holding down both the *`CONTROL`* key and another key, at the same time. For example, a *`Ctrl-d`* is typed by first pressing and holding the *`CONTROL`* key, next pressing the *`d`* key, and finally releasing both keys.

NOTE: Notes of interest look like this.

CAUTION: Cautionary or warning notes look like this.

Acknowledgments

Thanks to Jürgen Kahrs for his initial efforts to write a document like this. Although his prose has not survived, his material was helpful in preparing this booklet.

Thanks to Yehezkel Bernat for reviewing this document and in general for his good intentions.

FIXME: YOUR NAME HERE...

Notes to Reviewers

Please let me know if anything is missing, or unclear. Real errors with respect Git commands and usage are very important as well.

Spelling errors and typo fixes welcome, but not as important.

1 How to Start Contributing

`gawk` development is distributed. It's done using electronic mail (*email*) and via branches in the Git repository (or *repo*) on Savannah, the GNU project's source code management site.

In this chapter we use some Git terminology. If you're not at all familiar with Git, then skim this chapter and come back after reading the rest of the booklet.

`gawk` is similar to many other Free Software projects. To begin contributing, simply start! Take a look at the `TODO` file in the distribution, see if there is something of interest to you, and ask on the `bug-gawk@gnu.org` mailing list if anyone else is working on it. If not, then go for it! (See Chapter 8 [Development Stuff], page 23, for a discussion of some of the technical things you'll need to do. Here we describe the process in general.)

Your contribution can be almost anything that is relevant for `gawk`, such as code fixes, documentation fixes, and/or new features.

> **NOTE:** If possible, new features should be done using `gawk`'s extension mechanism. If you want to add a user-visible language change to the `gawk` core, you're going to have to convince the maintainer and the other developers that it's really worthwhile to do so.
>
> Changes that improve performance or portability, or that fix bugs, or that enable more things in extensions, will require less convincing, of course.

As you complete a task, submit patches for review to the `bug-gawk@gnu.org` mailing list, where you'll be given feedback about your work. Once your changes are acceptable, the maintainer will commit them to the Git repository.

Over time, as the maintainer and development team gain confidence in your ability to contribute, you may be asked to join the private `gawk` developers' mailing list, and/or be granted commit access to the Git repository on Savannah. This has happened to more than one person who just "came out of the woodwork."

Until that happens, or if you don't want to join the list, you should continue to work with private branches and submission of patches to the mailing list.

Once you have commit access, if you want to make a major change or add a major feature, where the patch(es) would be very large, it has become the practice to create a separate branch, based off of `master`, to host the feature. This way the maintainer can review it, and you can continue to improve it, until it's ready for integration into `master`.

> **NOTE:** Because of the GNU project's requirements for signed paperwork for contributions, the `gawk` project will **not** work with pull requests from GitHub or any other Git-based software hosting service. You must submit patches to the mailing list, and be willing to sign paperwork for large patches (see Section 8.2 [Assigning Copyrights to the FSF], page 23).

The `bug-gawk@gnu.org` mailing list is not private. Anyone may send mail to it, and anyone may subscribe to it. To subscribe, go to the list's web page and follow the instructions there. If you plan to be involved long-term with `gawk` development, then you probably should subscribe to the list.

2 Using Git

This chapter provides an introduction to using Git. Our point is *not* to rave about how wonderful Git is, nor to go into painful detail about how it works. Rather we want to give you enough background to understand how to use Git effectively for bug fix and feature development and to interact ("play nicely") with the development team.

2.1 The "Push/Pull" Model of Software Development

Git is a powerful, distributed source code management system. However, the way it's used for `gawk` development purposely does not take advantage of all its features.

Instead, the model is rather simple, and in many ways much like more traditional distributed systems such as the Concurrent Versions System (CVS) or Subversion (SVN).

The central idea can be termed "push/pull." You *pull* updates down from the central repository to your local copy, and if you have commit rights, you *push* your changes or updates up to the central repository.

Other developers work this way; pushing their changes up to the central repository and pulling your changes down into theirs.

Where Git does stand out is in its management of multiple branches of development. Git makes it very easy to set up a separate branch for use in fixing a bug or developing a feature. You can then easily keep that branch up to date with respect to the main development branch(es), and eventually merge the changes from your branch into the main branch.

Almost always Git does these merges for you without problem. When there is a problem (a *merge conflict*), usually it is very easy for you to *resolve* them and then complete the merge. We talk about this in more detail later (see Section 4.6.2 [Dealing With Merge Conflicts], page 13).

2.2 How Git Stores Branches and Their Copies

So how does Git work?[1]

A repository consists of a collection of *branches*. Each branch represents the history of a collection of files and directories (a file *tree*). Each combined set of changes to this collection (files and directories added or deleted, and/or file contents changed) is termed a *commit*.

When you first create a local copy of a remote repository ("clone the repo"), Git copies all of the original repository's branches to your local system. The original remote repository is referred to as being *upstream*, and your local repo is *downstream* from it. Git distinguishes branches from the upstream repo by prefixing their names with '`origin/`'. Let's draw some pictures. Figure 2.1 represents the state of the repo on Savannah:

[1] The following description is greatly simplified.

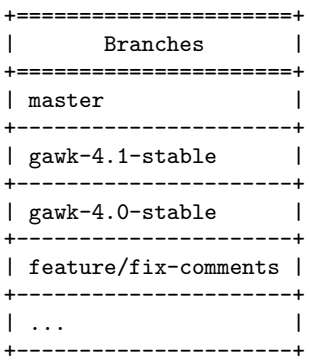

Figure 2.1: The Savannah `gawk` Repository

After you clone the repo, on your local system you will have a single branch named `master` that's visible when you use '`git branch`' to see your branches.

```
$ git clone http://git.savannah.gnu.org/r/gawk.git   Clone the repo
$ cd gawk                                            Change to local copy
$ git branch                                         See branch information
⊣ * master
```

The current branch is always indicated with a leading asterisk ('`*`').

Pictorially, the local repo looks like Figure 2.2 (you can ignore the '`T`' column for the moment):

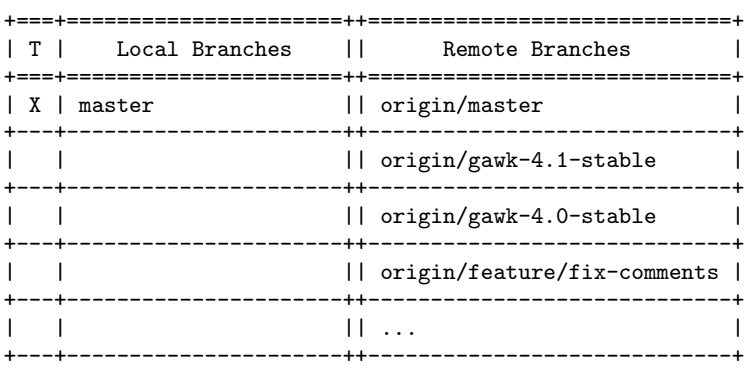

Figure 2.2: Your Local `gawk` Repository

Note that what is simply `gawk-4.1-stable` in the upstream repo is now referred to as `origin/gawk-4.1-stable`. The '`origin/`' branches are a snapshot of the state of the upstream repo. This is how Git allows you to see what changes you've made with respect to the upstream repo, without having to actually communicate with the upstream repo over the Internet. (When files are identical, Git is smart enough to not have two separate physical copies on your local disk.)

If you're working on a simple bug fix or change, you can do so directly in your local `master` branch. You can then commit your changes, and if you have access rights, push them upstream to the Savannah repo. (However, there is a process to follow. Please read the rest of this booklet.)

2.3 Local Branches

Let's talk about local branches in more detail. (The terminology used here is my own, not official Git jargon.) There are two kinds of local branches:

Tracking Branches

Tracking branches track branches from the upstream repository. You first create a tracking branch simply by checking out a branch from the upstream. You use the branch name without the leading '`origin/`' prefix. For example, '`git checkout gawk-4.1-stable`'.

You can then work on this branch, making commits to it as you wish. Once things are ready to move upstream, you simply use '`git push`', and your changes will be pushed up to the main repo.[1]

You should **never** checkout a branch using the '`origin/`' prefix. Things will get very confused. Always work on local tracking branches.

Purely Local Branches

A *purely local branch* exists only on your system. You may be developing some large new feature, or fixing a very difficult bug, or have a change for which paperwork has not yet been completed.

In such a case, you would keep your changes on a local branch, and periodically synchronize it with `master` (or whichever upstream branch you started from).

This may seem somewhat abstract so far. We demonstrate with commands and branches in Chapter 4 [Development Without Commit Access], page 11, later in this booklet.

Let's say you have checked out a copy of `gawk-4.1-stable` and have created a purely local branch named `better-random`. Then our picture now looks like Figure 2.3, where the '`T`' column indicates a tracking branch.

```
+===+======================++============================+
| T |   Local Branches     ||      Remote Branches       |
+===+======================++============================+
| X | master               || origin/master              |
+---+----------------------++----------------------------+
| X | gawk-4.1-stable      || origin/gawk-4.1-stable     |
+---+----------------------++----------------------------+
|   |                      || origin/gawk-4.0-stable     |
+---+----------------------++----------------------------+
|   |                      || origin/feature/fix-comments |
+---+----------------------++----------------------------+
|   |                      || ...                        |
+---+----------------------++----------------------------+
|   | better-random        ||                            |
+---+----------------------++----------------------------+
```

Figure 2.3: Your Local `gawk` Repository With a Purely Local Branch

[1] Assuming you have permission to do so, of course.

2.4 Branches Represent Development State

Branches represent development state. At any given time, when you checkout a particular branch (or create a new one), you have a copy of the `gawk` source tree that you should be able to build and test.

The following sections describe the different branches in the `gawk` repository and what they are for, as well as how to use your own branches.

2.4.1 Branches in the Savannah Repository

There are several kinds of branches in the Savannah repository.

Dead Branches
: Branches with the prefix '`dead-branches/`' (such as `dead-branches/const`) hold code that was never merged into the main code base. For example, a feature which was started, but later deemed to be unwise to add. These branches keep the code available, but they are not updated.

Stable Branches
: These branches are used for bug fixes to released versions of `gawk`. Sometimes new development (i.e., user-visible changes) also occurs on these branches, although in a perfect world they would be used only for bug fixes.

  These branches have names like `gawk-4.1-stable`, `gawk-4.0-stable`, and so on. Once a release has been made from `master`, the previous stable branch is not updated. For example, once `gawk` 4.1.0 was released, no more work was done on `gawk-4.0-stable`.

The Main Branch
: This is the `master` branch. Here is where most new feature development takes place, and releases of new major versions are based off of this branch.

  Feature branches are typically based off this branch as well, and when the feature is deemed complete, merged back into it.

Feature Branches
: Often, a proposed new feature or code improvement is quite involved. It may take some time to perfect, or the `gawk` development team may not be convinced that the feature should be kept.

  For this purpose, the team uses branches prefixed with '`feature/`'. This prefix is used even for code that simply improves the internals and does not make a user-visible change.

  Having large changes on separate branches makes it easier for members of the team to review the code, and also makes it easier to keep the changes up-to-date with respect to `master`, since Git excels at merging commits from one branch to another.

2.4.2 Branches in Your Local Repository

Purely local branches are where you do your own development. You may use purely local branches because you don't have commit rights to the Savannah repo. You may also use them if you are doing some work that isn't ready for sharing with the rest of the team, or cannot be committed for some other reason.

For example, for around a nine-month period, the maintainer kept a purely local branch for some contributed changes for which paperwork had not yet been completed.

2.4.3 A Closer Look at Branch Naming

Earlier, we said that Git maintains copies of the branches in the upstream repo, as well as manages your local branches. You can see all these branches with '`git branch -a`':

```
$ git branch -a
⊣   gawk-4.1-stable
⊣ * master
⊣   remotes/origin/HEAD -> origin/master
⊣   remotes/origin/dead-branches/async-events
⊣   ...
⊣   remotes/origin/feature/api-mpfr
⊣   remotes/origin/feature/array-iface
⊣   remotes/origin/feature/fix-comments
⊣   ...
```

You'll note that what we've referred to as '`origin/`' branches appear in the output with an additional prefix: '`remotes/`'. Up to this point, we've treated Git as if it allowed only a single upstream repository. But in fact, you can configure it to use more than one. All the known upstream repositories are grouped under the '`remotes/`' prefix, with `remotes/origin` being the one from which you initially cloned your local repository.

The ability to work with multiple upstream repositories is an advanced one; `gawk` development does not make use of it. The intent of this subsection is to explain the output from '`git branch -a`', nothing more.

3 Configuring Global Settings For Git

Before starting to use Git, you should configure it with some important settings that won't change as you use Git. You may configure options both globally, and on a per-repository basis. Here, we discuss only global configuration settings.

You can configure Git using either '`git config`', or by editing the relevant files with your favorite text editor.[1]

The first things to set are your email address and your real name:

```
$ git config --global user.name "J. P. Developer"     Set full name
$ git config --global user.email jpdev@example.com    Set email address
```

Setting these two items are an absolute requirement. **Note**: No aliases are allowed. If you can't supply your real name, you cannot contribute to the project. Other options that the `gawk` maintainer recommends that you use are:

```
$ git config --global push.default simple     Only push the current branch
$ git config --global pager.status true       Use pager for output of git status
```

The global settings are stored in the `.gitconfig` file in your home directory. The file looks like this:

```
[user]
        name = J. P. Developer
        email = jpdev@example.com
[push]
        default = simple
[pager]
        status = true
```

The `push.default=simple` setting ensures that older versions of Git only push the current branch up to the Savannah repo. This is the safest way to operate, and is the default in current Git versions.

There may be other settings in your configuration file as well. Use '`git config`' to see your settings:

```
$ git config --list
⊣ user.name=J. P. Developer
⊣ user.email=jpdev@example.com
⊣ push.default=simple
```

Here are the `gawk` maintainer's settings:

```
$ git config --global --list
⊣ user.name=Arnold D. Robbins
⊣ user.email=arnold@...
⊣ credential.helper=cache --timeout=3600
⊣ push.default=simple
⊣ color.ui=false
⊣ core.autocrlf=input
```

[1] You are required to use either Vim or Emacs, other text editors are not allowed. Of course, reasonable developers wouldn't want to use any other editor anyway.

- `pager.status=true`
- `log.decorate=auto`

Additional, per-project ("local") settings are stored in each repo's `.git/config` file.

4 Development Without Commit Access

In this chapter we present step-by-step recipes for checking out and working with a local copy of the Savannah Git repo for `gawk`. The presentation is for when you do not have commit access to the Git repo, and so you cannot push your changes directly.

4.1 Cloning The Repo

Clone the Savannah repo using '`git clone`'. You should do so using using the HTTPS protocol; HTTPS is considered to be more secure than the native Git protocol and is preferred.[1]

To choose which method, you supply a *URL* for the repo when you clone it, as follows.

```
$ git clone https://git.savannah.gnu.org/r/gawk.git  Clone the repo
⊣ ...
$ cd gawk                                            Start working
```

You only need to clone the repo once. From then on, you update its contents using other Git commands. For example, after coming back from your vacation in the Bahamas:

```
$ cd gawk                Move to the repo
$ make distclean         A good idea before updating
⊣ ...
$ git pull               Update it
```

To build, you should generally follow this recipe:

```
$ ./bootstrap.sh && ./configure && make -j && make check
```

NOTE: Unless you have installed all the tools described in Section 8.3.1 [GNU Tools], page 23, you *must* run `./bootstrap.sh` every time you clone a repo, do a '`git pull`' or checkout a different branch. (In the latter case, do '`make distclean`' first.) Otherwise things will get messy very quickly. The `bootstrap.sh` script ensures that all of the file time stamps are up to date so that it's not necessary to run the various configuration tools.

4.2 Switching Branches

So far, we've been working in the default `master` branch. Let's check what's happening in the `gawk-4.1-stable` branch:

```
$ make distclean                          Clean up
$ git checkout gawk-4.1-stable            Checkout a different branch
⊣ ...
$ git pull                                Get up to date
⊣ ...
$ ./bootstrap.sh && ./configure &&        Start working
> make -j && make check
```

[1] The native Git protocol is supported, but not recommended.

4.3 Starting A New Branch

Let's say you want to work on a new feature. For example, you might decide to add Python syntax support.[2] You should create a new branch on which to work. First, switch back to `master`:

```
$ make distclean
$ git checkout master
```

Now, create a new branch. The easiest way to do that is with the `-b` option to '`git checkout`':

```
$ git checkout -b feature/python
⊣ ...
```

You now do massive amounts of work in order to add Python syntax support. As you do each defined chunk of work, you update the `ChangeLog` file with your changes before *committing* them to the repo.

Let's say you've added a new file `python.c` and updated several others. Use '`git status`' to see what's changed:

```
$ git status
⊣ ...
```

Before committing the current set of changes, you can use '`git diff`' to view the changes. You may also use '`git difftool`'[3] to run an external `diff` command, such as `meld` on GNU/Linux:

```
$ git diff                          Regular built-in tool for standard diffs
$ git difftool --tool=meld          GUI diff tool
```

When you're happy with the changes, use '`git add`' to tell Git which of the changed and/or new files you wish to have ready to be committed:

```
$ git add ...
```

Use '`git status`' to see that your changes are scheduled for committing:

```
$ git status
⊣
```

Now you can commit your changes to your branch:

```
$ git commit
```

Running '`git commit`' causes Git to invoke an editor (typically from the `$EDITOR` environment variable) in which you can compose a commit message. Please supply a short message summarizing the commit. This message will be visible via '`git log`'.

4.4 Undoing A Change

Should you need to undo a change that you have not yet committed (so that you can start over), you can do so on per-file basis by simply checking out the file again:

```
$ git checkout awkgram.y        Undo changes to awkgram.y. There is no output
```

To start over completely, use '`git reset --hard`'. Note that this will *throw away* all your changes, with no chance for recovery, so be sure you really want to do it.

[2] Just joking. Please don't attempt this for real.

[3] Don't run '`git difftool`' in the background; it works interactively.

4.5 Moving Changes Aside

Sometimes, you may be in the middle of a set of changes that are not yet completed, when you need to stop what you're doing and work on something else. For example, you might be updating the documentation when a bug report comes in and you want to work on the bug. But you can't just switch branches, since you haven't finished your current changes.

The way to work around this problem is with '`git stash`'. This command saves your changes in a special place within Git from which they may be restored later. After executing '`git stash`', your current branch is restored to its original, pristine state.

The workflow might go something like this:

```
$ git checkout my-local-branch      Checkout a work branch
...                                 Do some work
$ git stash                         Save the work aside
$ git checkout gawk-4.1-stable      Work on a bug fix
...                                 Now we're done
$ git checkout my-local-branch      Go back to our local work
$ git stash pop                     Restore the earlier changes
```

The stash is maintained as a stack. Sets of changes are pushed onto the stack by '`git stash`' and popped off of it with '`git stash pop`'. You may use '`git stash list`' to see the list of saved changes.

4.6 Updating and Merging

As you work on your branch, you will occasionally want to bring it up to date with respect to `master`. This section discusses updating local branches and handling merge conflicts.

4.6.1 Rebasing A Local Branch

For purely local branches, bringing your branch up to date is called *rebasing*, which causes the branch to look *as if* you had started from the latest version of `master`. The steps are as follows:

```
$ git checkout master               Checkout master
$ git pull                          Update it
$ git checkout feature/python       Move back to new, purely local branch
$ git rebase master                 "Start over" from current master
```

4.6.2 Dealing With Merge Conflicts

Sometimes, when merging from `master` into your branch, or from a branch into `master`, there will be *merge conflicts*. These are one or more areas within a file where there are conflicting sets of changes, and Git could not do the merge for you. In this case, the conflicted area will be delimited by the traditional conflict markers, '`<<<`', '`===`' and '`>>>`'.

Your mission then is to edit the file and *resolve* the conflict by fixing the order of additions (such as in a `ChangeLog` file), or fixing the code to take new changes into account.

Once you have done so, you tell Git that everything is OK using '`git add`' and '`git commit`':

```
$ git checkout feature/python       Move back to new, purely local branch
$ git rebase master                 "Start over" from current master
```

```
⊣ First, rewinding head to replay your work on top of it...
⊣ Applying: Demo change.
⊣ Using index info to reconstruct a base tree...
⊣ M main.c
⊣ Falling back to patching base and 3-way merge...
⊣ Auto-merging main.c
⊣ CONFLICT (content): Merge conflict in main.c
⊣ error: Failed to merge in the changes.
⊣ Patch failed at 0001 Demo change.
⊣ Use 'git am --show-current-patch' to see the failed patch
⊣
⊣ 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 re-
base --abort".
$ gvim main.c                          Edit the file and fix the problem
$ git add main.c                       Tell Git everything is OK now ...
$ git commit                           ... and it's settled
$ git rebase --continue                Continue the rebase
```

The `git rebase --continue` then continues the process of rebasing the current branch that we started in Section 4.6.1 [Rebasing A Local Branch], page 13. It's not necessary if you are using '`git merge`' (see Section 4.9 [Points to Remember], page 15).

4.7 Submitting Your Changes

So now your feature is complete. You've added test cases for it to the test suite[4], you have `ChangeLog` entries that describe all the changes[5], you have documented the new feature[6], and everything works great. You're ready to submit the changes for review, and with any luck, inclusion into `gawk`.

There are two ways to submit your changes for review.

Generate a single large patch

To do this, simply compare your branch to the branch off which it is based:

```
$ git checkout feature/python
$ git diff master > /tmp/python.diff
```

Mail the `python.diff` file to the appropriate mailing list along with a description of what you've changed and why.

Generate a set of patches that in toto comprise your changes

To do this, use '`git format-patch`':

```
$ git checkout feature/python
$ git format-patch
```

[4] You did do this, didn't you?

[5] You remembered this, right?

[6] You wouldn't neglect this, would you?

This creates a set of patch files, one per commit that isn't on the original branch. Mail these patches, either separately, or as a set of attachments, to the appropriate mailing list along with a description of what you've changed and why.

Either way you choose to submit your changes, the `gawk` maintainer and development team will review your changes and provide feedback. If you have signed paperwork with the FSF for `gawk` and the maintainer approves your changes, he will apply the patch(es) and commit the changes.

Which list should you send mail to? If you are just starting to contribute, use `bug-gawk@gnu.org`. After making enough contributions, you may be invited to join the private `gawk` developers' mailing list. If you do so, then submit your changes to that list.

If you make any substantial changes, you will need to assign copyright in those changes to the Free Software Foundation before the maintainer can commit those changes. See Section 8.2 [Assigning Copyrights to the FSF], page 23, for more information.

4.8 Removing Branches

Once the maintainer has integrated your changes, you can get rid of your local branch:

```
$ git checkout master                 Move to upstream branch
$ git pull                            Update
$ gvim ChangeLog ...                  Verify your changes are in
$ git branch -d feature/python        Remove your local branch
```

4.9 Points to Remember

There are some important points to remember:

- Always do a '`make distclean`' before switching between branches. Things will get really confused if you don't.
- For upstream branches, *always* work with tracking branches. *Never* use '`git checkout origin/whatever`'. Git will happily let you do something like that, but it's just plain asking for trouble.
- Make sure your tracking branches are up-to-date before doing anything with them, particularly using them as the basis for a rebase or merge. This typically means a three-step process:

  ```
  $ git checkout master               Get to local copy
  $ git pull                          Bring it up to date
  $ git checkout feature/python       Go back to your branch
  ```

  You can then do the actual rebase:

  ```
  $ git rebase master                 Now rebase your feature off of master
  ```

- Git always treats the currently checked-out branch as the object of operations. For example, when comparing files with the regular `diff` command, the usage is '`diff oldfile newfile`'. For '`git diff`', the current branch takes the place of *newfile*, thus:

  ```
  $ git checkout feature/python
  $ git diff master                   Compare master to current branch
  ```

or if merging:

```
$ git checkout master          Checkout master
$ git pull                     Update tracking branch
$ git merge feature/python     Merge changes into master
```

5 Development With Commit Access

This chapter describes how to do development when you *do* have commit access to the `gawk` repo on Savannah.

5.1 Initial Setup

Congratulations! After becoming a quality contributor to `gawk` development, you've been invited to join the private development list and to accept having commit access to the repo.

The first thing to do is to create an account on Savannah, choosing a unique user name. To do so, go to the Savannah home page and click on the "New User" link. The setup will include uploading of your `ssh` key, as per the instructions on the Savannah web page.

After you've done all this, send email to the maintainer with your Savannah user name, and he will add you to the list of users who have commit access to the repo.

5.2 Cloning The Repo With An `ssh` URL

In order to be able to commit changes to the repo, you must clone it using an '`ssh://`' URL. Cloning the repo with `ssh` is similar to cloning with HTTPS, but the URL is different:

```
$ git clone ssh://yourname@git.sv.gnu.org/srv/git/gawk.git
⊣ ...
```

Here, you should replace '`yourname`' in the command with the user name you chose for use on Savannah.

5.3 Developing Patches

The first part of developing a patch is the same as for developers without commit access:

1. Develop the code and test it.
2. Update the `ChangeLog`.
3. If necessary, update the documentation: `doc/gawktexi.in` and/or `doc/gawk.1`.
4. Use '`git diff > mychange.diff`' to create a patch file.
5. Send it to the mailing list for discussion.
6. Iterate until the patch is ready to be committed.

However, now that you have commit access, you can commit the fix and push it up to the repo yourself! Let's assume you've made a bug fix directly on `master`. Here's how to commit your changes:

```
$ git diff          Review the patch one more time
$ git add ...       Add any files for committing
$ git commit        Commit the files, with a commit message
$ git push          Push the files up to the repo.  Ta da!
```

The first three steps are the same described earlier (see Section 4.3 [Starting A New Branch], page 12). The '`git push`' is what's new, and it updates the repo on Savannah. Congratulations!

As a courtesy, you should send a note to the mailing list indicating that you have pushed your change.

5.4 Developing New Features

Developing a new feature can be easier once you have commit access to the repo. First, create a new branch to hold your feature:

```
$ git checkout master                  Start from master
$ git pull                             Be sure to be up to date
$ git checkout -b feature/python       Create and switch to a new branch
```

Now, you can develop as normal, adding new files if necessary (such as new tests), modifying code, updating the `ChangeLog` and documentation, and so on.

You can share changes with the mailing list as diffs, as usual. However, especially for a large feature, it would be better to push your branch up to Savannah. Then, everyone else can simply pull it down to their local systems and review your changes at their leisure.

To push your branch up initially:

```
$ git diff                             Review your changes
$ git add ...                          Add any files for committing
$ git commit                           Commit the files with a commit message
$ git push -u origin feature/python    Push the branch up to the repo
```

When you use '`push -u origin`', Git helpfully converts your purely local branch into a tracking branch. It becomes as if the branch had originated from the upstream repo and you checked it out locally.

You only need to do '`git push -u origin`*' once.* As you continue to work on your branch, the workflow simplifies into this:

```
$ git diff                             Review your changes
$ git add ...                          Add any files for committing
$ git commit                           Commit the files
$ git push                             Push your changes to the branch upstream
```

5.5 Developing Fixes

If you want to make a fix on `master` or on the current stable branch, you work the same way, by producing and discussing a diff on the mailing list. Once it's approved, you can commit it yourself:

```
$ git checkout master                  Move to master
$ git pull                             Make sure we're up to date with the maintainer
$ gvim ...                             Make any fixes, compile, test
$ git diff                             Review your changes
$ git add ...                          Add any files for committing
$ git commit                           Commit the files with a commit message.
```

When you're ready to push your changes:

```
$ git pull                             Download latest version; Git will merge
$ gvim ...                             Resolve any merge conflicts with git add and git commit
$ git push                             Now you can push your changes upstream
```

See Section 4.6.2 [Dealing With Merge Conflicts], page 13, for instructions on dealing with merge conflicts.

6 General Development Practices

This chapter discusses general practices for `gawk` development. The discussion here is mainly for developers with commit access to the Savannah repo.

Propagating Fixes
: Usually, bug fixes should be made on the current "stable" branch. Once a fix has been reviewed and approved, you can commit it and push it yourself. Typically, the maintainer then takes care to merge the fix to `master` and from there to any other branches. However, you are welcome to save him the time and do this yourself.

Directory ownership
: Some developers "own" certain parts of the tree, such as the `pc` and `vms` directories. They are allowed to commit changes to those directories without review by the mailing list, but changes that also touch the mainline code should be submitted for review.

New feature development
: Unless you can convince the maintainer (and the other developers!) otherwise, you should *always* start branches for new features from `master`, and not from the current "stable" branch.

  Use '`checkout -b feature/`*`feature_name`*' to create the initial branch. You may then elect to keep it purely local, or to push it up to Savannah for review, even if the feature is not yet totally "ready for prime time."

During development of a new feature, you will most likely wish to keep your feature branch up to date with respect to ongoing improvements in `master`. This is generally easy to do. There are two different mechanisms, and which one you use depends upon the nature of your new feature branch.

As long as your branch is purely local
: You should use '`git rebase`' to the keep the branch synchronized with the original branch from which it was forked:

```
$ git checkout master              Move to master
$ git pull                         Bring it up to date
$ git checkout feature/python      Move to your new feature branch
$ git rebase master                Rebase from master
```

  The rebasing operation may require that you resolve conflicts (see Section 4.6.2 [Dealing With Merge Conflicts], page 13). Edit any conflicted files and resolve the problem(s). Compile and test your changes, then use '`git add`' and '`git commit`' to indicate resolution, and then use '`git rebase --continue`' to continue the rebasing. Git is very good about providing short instructions on how to continue when such conflicts occur.

Once the branch has been pushed up to Savannah
: You *must* use '`git merge`' to bring your feature branch up to date. That flow looks like this:

```
$ git checkout master              Move to master
```

```
$ git pull                          Bring it up to date
$ git checkout feature/python       Move to your new feature branch
$ git merge master                  Merge from master
```

Here too, you may have to resolve any merge conflicts (see Section 4.6.2 [Dealing With Merge Conflicts], page 13). Once that's done, you can push the changes up to Savannah.

When the changes on your branch are complete, usually the maintainer merges the branch to `master`. But there's really no magic involved, the merge is simply done in the other direction:

```
$ git checkout feature/python       Checkout feature branch
$ git pull                          Bring it up to date
$ git checkout master               Checkout master
$ git pull                          Bring it up to date
$ git merge feature/python          Merge from feature/python into master
```

If you've been keeping '`feature/python`' in sync with `master`, then there should be no merge conflicts to resolve, and you can push the result to Savannah:

```
$ git push                          Push up to Savannah
```

Since '`feature/python`' is no longer needed, it can be gotten rid of:

```
$ git branch                                    Still on master
  ...
* master
$ git branch -d feature/python                  Delete feature branch
$ git push -u origin --delete feature/python    Delete on Savannah
```

The '`git push`' command deletes the `feature/python` branch from the Savannah repo.

Finally, you should send an email to developer's list describing what you've done so that everyone else can delete their copies of the branch and do a '`git fetch --prune`' (see Chapter 7 [Keeping Your Repo Organized], page 21).

To update the other remaining development branches with the latest changes on `master`, use the '`helpers/update-branches.sh`' script in the repo.

7 Keeping Your Repo Organized

There are a few commands you should know about to help keep your local repo clean.

Removing old branches

Developers add branches to the Savannah repo and when development on them is done, they get merged into `master`. Then the branches on Savannah are deleted (as shown in Chapter 6 [General Development Practices], page 19).

However, your local copies of those branches (labelled with the '`origin/`' prefix) remain in your local repo. If you don't need them, then you can clean up your repo as follows.

First, remove any related tracking branch you may have:

```
$ git pull                                    Get up to date
$ git branch -d feature/merged-feature        Remove tracking branch
```

Then, ask Git to clean things up for you:

```
$ git fetch --prune                           Remove unneeded branches
```

Removing cruft

As Git works, occasional "cruft" collects in the repository. Git does occasionally clean this out on its own, but if you're concerned about disk usage, you can do so yourself using '`git gc`' (short for "garbage collect"). For example:

```
$ du -s .                                     Check disk usage
⊣ 99188   .                                   Almost 10 megabytes
$ git gc                                      Collect garbage
⊣ Counting objects: 32114, done.
⊣ Delta compression using up to 4 threads.
⊣ Compressing objects: 100% (6370/6370), done.
⊣ Writing objects: 100% (32114/32114), done.
⊣ Total 32114 (delta 25655), reused 31525 (delta 25231)
$ du -s .                                     Check disk usage again
⊣ 75168   .                                   Down to 7 megabytes
```

Renaming branches

Occasionally you may want to rename a branch.[1] If your branch is local and you are on it, use:

```
$ git branch -m feature/new-name
```

Otherwise, use:

```
$ git branch -m feature/old-name feature/new-name
```

You then need to fix the upstream repo. This command does so, using an older syntax to simultaneously delete the old name and push the new name. You should be on the new branch:

```
$ git push origin :feature/old-name feature/new-name
```

NOTE: It is the leading '`:`' in the first branch name that causes Git to delete the old name in the upstream repo. Don't omit it!

[1] This discussion is adopted from here.

Finally, reset the upstream branch for the local branch with the new name:

```
$ git push -u origin feature/new-name
```

You should also update the mailing list to let the other developers know what's happening.

8 Development Stuff

This chapter discusses other things you need to know and/or do if you're going to participate seriously in `gawk` development.

8.1 Coding Style

You should read the discussion about adding code in the `gawk` documentation. See Section "Making Additions to `gawk`" in *GAWK: Effective awk Programming*, for a discussion of the general procedure. In particular, pay attention to the coding style guidelines in Section "Adding New Features" in *GAWK: Effective awk Programming*.[1] These two sections may also be found online, at `https://www.gnu.org/software/gawk/manual/html_node/Additions.html#Additions`, and `https://www.gnu.org/software/gawk/manual/html_node/Adding-Code.html#Adding-Code`, respectively.

8.2 Assigning Copyrights to the FSF

For any change of more than just a few lines, you will need to assign copyright in (that is, ownership of) those changes to the Free Software Foundation.

This is generally an easy thing to do. In particular, you can choose to use a version of the copyright assignment which assigns all your current *and future* changes to `gawk` to the FSF. This means that you only need to do the paperwork once, and from then on all your changes will automatically belong to the FSF. The maintainer recommends doing this.

The maintainer will help you with this process once you have a contribution that warrants it.

8.3 Software Tools You Will Need

This section discusses additional tools that you may need to install on your system in order to be in sync with what the `gawk` maintainer uses. It also discusses different C compiler options for use during code development, and how to compile `gawk` for debugging.

8.3.1 GNU Tools

If you expect to work with the configuration files and/or the `Makefile` files, you will need to install a number of other GNU tools. In general, you should be using the latest versions of the tools, or least the same ones that the maintainer himself uses. This helps minimize the differences that the maintainer has to resolve when merging changes, and in general avoids confusion and hassle. Similarly, you should install the latest GNU documentation tools as well. The tools are described in the following list:

`autoconf` GNU Autoconf processes the `configure.ac` files in order to generate the `configure` shell script and `config.h.in` input file. See the Autoconf home page for more information.

`automake` GNU Automake processes the `configure.ac` and `Makefile.am` files to produce `Makefile.in` files. See the Automake home page for more information.

[1] Changes that don't follow the coding style guidelines won't be accepted. Period.

`gettext` GNU Gettext processes the `gawk` source code to produce the original `po/gawk.pot` message template file. Normally you should not need need to do this; the maintainer usually manages this task. See the Gettext home page for more information.

`libtool` GNU Libtool works with Autoconf and Automake to produce portable shared libraries. It is used for the extensions that ship with `gawk`, whose code is in the `extensions` directory. See the Libtool home page for more information.

`makeinfo` The `makeinfo` command is used to build the Info versions of the documentation. You need to have the same version as the maintainer uses, so that when you make a change to the documentation, the corresponding change to the generated Info file will be minimal. `makeinfo` is part of GNU Texinfo. See the Texinfo home page for more information.

8.3.2 Compilers

The default compiler for `gawk` development is GCC, the GNU Compiler Collection. The default version of GCC is whatever is on the maintainer's personal GNU/Linux system, although he does try to build the latest released version if that is newer than what's on his system, and then occasionally test `gawk` with it.

He also attempts to test occasionally with `clang`. However, he uses whatever is the default for his GNU/Linux system, and does *not* make an effort to build the current version for testing.

Both GCC and `clang` are highly optimizing compilers that produce good code, but are very slow. There are two other compilers that are faster, but that may not produce quite as good code. However, they are both reasonable for doing development.

The Tiny C Compiler, `tcc`

This compiler is *very* fast, but it produces only mediocre code. It is capable of compiling `gawk`, and it does so well enough that '`make check`' runs without errors.

However, in the past the quality has varied, and the maintainer has had problems with it. He recommends using it for regular development, where fast compiles are important, but rebuilding with GCC before doing any commits, in case `tcc` has missed something.[2]

See the project's home page for some information. More information can be found in the project's Git repository. The maintainer builds from the `mob` branch for his work, but after updating it you should check that this branch still works to compile `gawk` before installing it.

The (Revived) Portable C Compiler

This is an updated version of the venerable Unix Portable C Compiler, PCC. It accepts ANSI C syntax and supports both older and modern architectures. It produces better code than `tcc` but is slower, although still much faster than GCC and `clang`.

[2] This bit the maintainer once.

See the project's home page for more information. See `http://pcc.ludd.ltu.se/supported-platforms` for instructions about obtaining the code using CVS and building it.

An alternative location for the source is the `gawk` maintainer's Git mirror of the code. If you're using Ubuntu GNU/Linux 18.04 or later, you need to use the `ubuntu-18` branch from this Git mirror.

8.4 Compiling For Debugging

If you wish to compile for debugging, you should use GCC. After running `configure` but before running `make`, edit the `Makefile` and remove the `-O2` flag from the definition of `CFLAGS`. Optionally, do the same for `support/Makefile` and/or `extensions/Makefile`. Then run `make`.

You can enable additional debugging code by creating a file named `.developing` in the `gawk` source code directory *before* running `configure`. Doing so enables additional conditionally-compiled debugging code within `gawk`, and adds additional warning and debugging options if compiling with GCC. It also disables optimization.

Appendix A Git Command Cheat Sheet

This appendix provides an alphabetical list of the Git commands cited in this booklet, along with brief descriptions of what the commands do.

Note that you may always use either '`git help` *`command`*' or '`git` *`command`* `--help`' to get short, man-page style help on how to use any given Git command.

`git add` Add a file to the list of files to be committed.

`git branch`
: View existing branches, or delete a branch. The most useful options are `-a` and `-d`.

`git checkout`
: Checkout an existing branch, create a new branch, or checkout a file to reset it. Use the `-b` option to create and checkout a new branch in one operation.

`git clone` Clone (make a new copy of) an existing repository. You generally only need to do this once.

`git commit`
: Commit changes to files which have been staged for committing with '`git add`'. This makes your changes permanent, *in your local repository only*. To publish your changes to an upstream repo, you must use '`git push`'.

`git config`
: Display and/or change global and/or local configuration settings.

`git diff` Show a unified-format diff of what's changed in the current directory as of the last commit. It helps to have Git configured to use its builtin pager for reviewing diffs (see Chapter 3 [Configuring Global Settings For Git], page 9).

`git difftool`
: Use a "tool" (usually a GUI-based program) to view differences, instead of the standard textual diff as you'd get from '`git diff`'.

`git fetch` Update your local copy of the upstream's branches. That is, update the various '`origin/`' branches. This leaves your local tracking branches unchanged. With the `--prune` option, this removes any copies of stale '`origin/`' branches.

`git format-patch`
: Create a series of patch files, one per commit not on the original branch from which you started.

`git gc` Run a "garbage collection" pass in the current repository. This can often reduce the space used in a large repo. For `gawk` it does not make that much difference.

`git help` Print a man-page–style usage summary for a command.

`git log` Show the current branch's commit log. This includes who made the commit, the date, and the commit message. Commits are shown from newest to oldest.

`git merge` Merge changes from the named branch into the current one.

`git pull` When in your local tracking branch *`xxx`*, run '`git fetch`', and then merge from `origin/`*`xxx`* into *`xxx`*.

`git push` Push commits from your local tracking branch *xxx* through `origin/xxx` and on to branch *xxx* in the upstream repo. Use '`git push -u origin --delete xxx`' to delete an upstream branch. (Do so carefully!)

`git rebase`
: Rebase the changes in the current purely local branch to look as if they had been made relative to the latest commit in the current upstream branch (typically `master`). This is how you keep your local, in-progress changes up-to-date with respect to the original branch from which they were started.

`git reset` Restore the original state of the repo, especially with the `--hard` option. Read up on this command, and use it carefully.

`git stash` Save your current changes in a special place within Git. They can be restored with '`git stash pop`', even on a different branch. Use '`git stash list`' to see the list of stashed changes.

`git status`
: Show the status of files that are scheduled to be committed, and those that have been modified but not yet scheduled for committing. Use '`git add`' to schedule a file for committing. This command also lists untracked files.

Appendix B Git Resources

There are many Git resources available on the Internet. Start at the Git Project home page. In particular, the *Pro Git* book is available online.

See also the Savannah quick introduction to Git.

A nice article on how Git works is *Git From The Bottom Up*, by John Wiegley.

Appendix C Stuff Still To Do In This Document

- Fill out all examples with full output

Index

K

L

M

O

P

R

S

T

U