Magit User Manual

Jonas Bernoulli

Table of Contents

1 Introduction

Magit is an interface to the version control system Git, implemented as an Emacs package. Magit aspires to be a complete Git porcelain. While we cannot (yet) claim that Magit wraps and improves upon each and every Git command, it is complete enough to allow even experienced Git users to perform almost all of their daily version control tasks directly from within Emacs. While many fine Git clients exist, only Magit and Git itself deserve to be called porcelains.

Staging and otherwise applying changes is one of the most important features in a Git porcelain and here Magit outshines anything else, including Git itself. Git's own staging interface (`git add --patch`) is so cumbersome that many users only use it in exceptional cases. In Magit staging a hunk or even just part of a hunk is as trivial as staging all changes made to a file.

The most visible part of Magit's interface is the status buffer, which displays information about the current repository. Its content is created by running several Git commands and making their output actionable. Among other things, it displays information about the current branch, lists unpulled and unpushed changes and contains sections displaying the staged and unstaged changes. That might sound noisy, but, since sections are collapsible, it's not.

To stage or unstage a change one places the cursor on the change and then types `s` or `u`. The change can be a file or a hunk, or when the region is active (i.e. when there is a selection) several files or hunks, or even just part of a hunk. The change or changes that these commands - and many others - would act on are highlighted.

Magit also implements several other "apply variants" in addition to staging and unstaging. One can discard or reverse a change, or apply it to the working tree. Git's own porcelain only supports this for staging and unstaging and you would have to do something like `git diff ... | ??? | git apply ...` to discard, revert, or apply a single hunk on the command line. In fact that's exactly what Magit does internally (which is what lead to the term "apply variants").

Magit isn't just for Git experts, but it does assume some prior experience with Git as well as Emacs. That being said, many users have reported that using Magit was what finally taught them what Git is capable of and how to use it to its fullest. Other users wished they had switched to Emacs sooner so that they would have gotten their hands on Magit earlier.

While one has to know the basic features of Emacs to be able to make full use of Magit, acquiring just enough Emacs skills doesn't take long and is worth it, even for users who prefer other editors. Vim users are advised to give Evil (`https://bitbucket.org/lyro/evil/wiki/Home`), the "Extensible VI Layer for Emacs", and Spacemacs (`https://github.com/syl20bnr/spacemacs`), an "Emacs starter-kit focused on Evil" a try.

Magit provides a consistent and efficient Git porcelain. After a short learning period, you will be able to perform most of your daily version control tasks faster than you would on the command line. You will likely also start using features that seemed too daunting in the past.

Magit fully embraces Git. It exposes many advanced features using a simple but flexible interface instead of only wrapping the trivial ones like many GUI clients do. Of course

Magit supports logging, cloning, pushing, and other commands that usually don't fail in spectacular ways; but it also supports tasks that often cannot be completed in a single step. Magit fully supports tasks such as merging, rebasing, cherry-picking, reverting, and blaming by not only providing a command to initiate these tasks but also by displaying context sensitive information along the way and providing commands that are useful for resolving conflicts and resuming the sequence after doing so.

Magit wraps and in many cases improves upon at least the following Git porcelain commands: `add`, `am`, `bisect`, `blame`, `branch`, `checkout`, `cherry`, `cherry-pick`, `clean`, `clone`, `commit`, `config`, `describe`, `diff`, `fetch`, `format-patch`, `init`, `log`, `merge`, `merge-tree`, `mv`, `notes`, `pull`, `rebase`, `reflog`, `remote`, `request-pull`, `reset`, `revert`, `rm`, `show`, `stash`, `submodule`, `subtree`, `tag`, and `worktree`. Many more Magit porcelain commands are implemented on top of Git plumbing commands.

2 Installation

Magit can be installed using Emacs' package manager or manually from its development repository.

2.1 Installing from Melpa

Magit is available from Melpa and Melpa-Stable. If you haven't used Emacs' package manager before, then it is high time you familiarize yourself with it by reading the documentation in the Emacs manual, see Section "Packages" in `emacs`. Then add one of the archives to `package-archives`:

- To use Melpa:

  ```
  (require 'package)
  (add-to-list 'package-archives
               '("melpa" . "http://melpa.org/packages/") t)
  ```

- To use Melpa-Stable:

  ```
  (require 'package)
  (add-to-list 'package-archives
               '("melpa-stable" . "http://stable.melpa.org/packages/") t)
  ```

Once you have added your preferred archive, you need to update the local package list using:

```
M-x package-refresh-contents RET
```

Once you have done that, you can install Magit and its dependencies using:

```
M-x package-install RET magit RET
```

Now see Section 2.3 [Post-Installation Tasks], page 4.

2.2 Installing from the Git Repository

Magit depends on the `dash`, `transient` and `with-editor` libraries which are available from Melpa and Melpa-Stable. Install them using `M-x package-install RET <package> RET`. Of course you may also install them manually from their repository.

Then clone the Magit repository:

```
$ git clone https://github.com/magit/magit.git ~/.emacs.d/site-lisp/magit
$ cd ~/.emacs.d/site-lisp/magit
```

Then compile the libraries and generate the info manuals:

```
$ make
```

If you haven't installed `dash`, `transient` and `with-editor` from Melpa or at `/path/to/magit/../<package>`, then you have to tell `make` where to find them. To do so create the file `/path/to/magit/config.mk` with the following content before running `make`:

```
LOAD_PATH  = -L /path/to/magit/lisp
LOAD_PATH += -L /path/to/dash
LOAD_PATH += -L /path/to/transient
LOAD_PATH += -L /path/to/with-editor
```

Finally add this to your init file:

```
(add-to-list 'load-path "~/.emacs.d/site-lisp/magit/lisp")
(require 'magit)

(with-eval-after-load 'info
  (info-initialize)
  (add-to-list 'Info-directory-list
               "~/.emacs.d/site-lisp/magit/Documentation/"))
```

Note that you have to add the `lisp` subdirectory to the `load-path`, not the top-level of the repository, and that elements of `load-path` should not end with a slash, while those of `Info-directory-list` should.

Instead of requiring the feature `magit`, you could load just the autoload definitions, by loading the file `magit-autoloads.el`.

```
(load "/path/to/magit/lisp/magit-autoloads")
```

Instead of running Magit directly from the repository by adding that to the `load-path`, you might want to instead install it in some other directory using `sudo make install` and setting `load-path` accordingly.

To update Magit use:

```
$ git pull
$ make
```

At times it might be necessary to run `make clean all` instead.

To view all available targets use `make help`.

Now see Section 2.3 [Post-Installation Tasks], page 4.

2.3 Post-Installation Tasks

After installing Magit you should verify that you are indeed using the Magit, Git, and Emacs releases you think you are using. It's best to restart Emacs before doing so, to make sure you are not using an outdated value for `load-path`.

```
M-x magit-version RET
```

should display something like

```
Magit 2.8.0, Git 2.10.2, Emacs 25.1.1, gnu/linux
```

Then you might also want to read about options that many users likely want to customize. See Section 9.2 [Essential Settings], page 113.

To be able to follow cross references to Git manpages found in this manual, you might also have to manually install the `gitman` info manual, or advice `Info-follow-nearest-node` to instead open the actual manpage. See Section A.1.2 [How to install the gitman info manual?], page 128.

If you are completely new to Magit then see Chapter 3 [Getting Started], page 5.

If you run into problems, then please see the Appendix A [FAQ], page 128. Also see the Chapter 11 [Debugging Tools], page 133.

And last but not least please consider making a donation, to ensure that I can keep working on Magit. See `https://magit.vc/donations`. for various donation options.

3 Getting Started

This short tutorial describes the most essential features that many Magitians use on a daily basis. It only scratches the surface but should be enough to get you started.

IMPORTANT: It is safest if you clone some repository just for this tutorial. Alternatively you can use an existing local repository, but if you do that, then you should commit all uncommitted changes before proceeding.

To display information about the current Git repository, type `M-x magit-status RET`. You will be using this command a lot, and should therefore give it a global key binding. This is what we recommend:

```
(global-set-key (kbd "C-x g") 'magit-status)
```

Most Magit commands are commonly invoked from the status buffer. It can be considered the primary interface for interacting with Git using Magit. Many other Magit buffers may exist at a given time, but they are often created from this buffer.

Depending on what state your repository is in, this buffer may contain sections titled "Staged changes", "Unstaged changes", "Unmerged into origin/master", "Unpushed to origin/master", and many others.

Since we are starting from a safe state, which you can easily return to (by doing a `git reset --hard PRE-MAGIT-STATE`), there currently are no staged or unstaged changes. Edit some files and save the changes. Then go back to the status buffer, while at the same time refreshing it, by typing `C-x g`. (When the status buffer, or any Magit buffer for that matter, is the current buffer, then you can also use just `g` to refresh it).

Move between sections using `p` and `n`. Note that the bodies of some sections are hidden. Type `TAB` to expand or collapse the section at point. You can also use `C-tab` to cycle the visibility of the current section and its children. Move to a file section inside the section named "Unstaged changes" and type `s` to stage the changes you have made to that file. That file now appears under "Staged changes".

Magit can stage and unstage individual hunks, not just complete files. Move to the file you have just staged, expand it using `TAB`, move to one of the hunks using `n`, and unstage just that by typing `u`. Note how the staging (`s`) and unstaging (`u`) commands operate on the change at point. Many other commands behave the same way.

You can also un-/stage just part of a hunk. Inside the body of a hunk section (move there using `C-n`), set the mark using `C-SPC` and move down until some added and/or removed lines fall inside the region but not all of them. Again type `s` to stage.

It is also possible to un-/stage multiple files at once. Move to a file section, type `C-SPC`, move to the next file using `n`, and then `s` to stage both files. Note that both the mark and point have to be on the headings of sibling sections for this to work. If the region looks like it does in other buffers, then it doesn't select Magit sections that can be acted on as a unit.

And then of course you want to commit your changes. Type `c`. This shows the available commit commands and arguments in a buffer at the bottom of the frame. Each command and argument is prefixed with the key that invokes/sets it. Do not worry about this for now. We want to create a "normal" commit, which is done by typing `c` again.

Now two new buffers appear. One is for writing the commit message, the other shows a diff with the changes that you are about to committed. Write a message and then type `C-c C-c` to actually create the commit.

You probably don't want to push the commit you just created because you just committed some random changes, but if that is not the case you could push it by typing `P` to show all the available push commands and arguments and then `p` to push to a branch with the same name as the local branch onto the remote configured as the push-remote. (If the push-remote is not configured yet, then you would first be prompted for the remote to push to.)

So far we have mentioned the commit, push, and log transient prefix commands. These are probably among the transients you will be using the most, but many others exist. To show a transient that lists all other transients (as well as the various apply commands and some other essential commands), type `h`. Try a few.

The key bindings in that transient correspond to the bindings in Magit buffers, including but not limited to the status buffer. So you could type `h d` to bring up the diff transient, but once you remember that "d" stands for "diff", you would usually do so by just typing `d`. But this "prefix of prefixes" is useful even once you have memorized all the bindings, as it can provide easy access to Magit commands from non-Magit buffers. You should create a global key binding for this command too:

```
(global-set-key (kbd "C-x M-g") 'magit-dispatch)
```

In the same vein, you might also want to enable `global-magit-file-mode` to get some more Magit key bindings in regular file-visiting buffers (see Section 8.8 [Minor Mode for Buffers Visiting Files], page 109).

It is not necessary that you do so now, but if you stick with Magit, then it is highly recommended that you read the next section too.

4 Interface Concepts

4.1 Modes and Buffers

Magit provides several major-modes. For each of these modes there usually exists only one buffer per repository. Separate modes and thus buffers exist for commits, diffs, logs, and some other things.

Besides these special purpose buffers, there also exists an overview buffer, called the **status buffer**. It's usually from this buffer that the user invokes Git commands, or creates or visits other buffers.

In this manual we often speak about "Magit buffers". By that we mean buffers whose major-modes derive from `magit-mode`.

M-x magit-toggle-buffer-lock   (magit-toggle-buffer-lock)
>This command locks the current buffer to its value or if the buffer is already locked, then it unlocks it.
>
>Locking a buffer to its value prevents it from being reused to display another value. The name of a locked buffer contains its value, which allows telling it apart from other locked buffers and the unlocked buffer.
>
>Not all Magit buffers can be locked to their values; for example, it wouldn't make sense to lock a status buffer.
>
>There can only be a single unlocked buffer using a certain major-mode per repository. So when a buffer is being unlocked and another unlocked buffer already exists for that mode and repository, then the former buffer is instead deleted and the latter is displayed in its place.

4.1.1 Switching Buffers

`magit-display-buffer` *buffer* **&optional** *display-function*   [Function]
>This function is a wrapper around `display-buffer` and is used to display any Magit buffer. It displays BUFFER in some window and, unlike `display-buffer`, also selects that window, provided `magit-display-buffer-noselect` is `nil`. It also runs the hooks mentioned below.
>
>If optional DISPLAY-FUNCTION is non-nil, then that is used to display the buffer. Usually that is `nil` and the function specified by `magit-display-buffer-function` is used.

`magit-display-buffer-noselect`   [Variable]
>When this is non-nil, then `magit-display-buffer` only displays the buffer but forgoes also selecting the window. This variable should not be set globally, it is only intended to be let-bound, by code that automatically updates "the other window". This is used for example when the revision buffer is updated when you move inside the log buffer.

`magit-display-buffer-function`   [User Option]
>The function specified here is called by `magit-display-buffer` with one argument, a buffer, to actually display that buffer. This function should call `display-buffer` with that buffer as first and a list of display actions as second argument.

Magit provides several functions, listed below, that are suitable values for this option. If you want to use different rules, then a good way of doing that is to start with a copy of one of these functions and then adjust it to your needs.

Instead of using a wrapper around `display-buffer`, that function itself can be used here, in which case the display actions have to be specified by adding them to `display-buffer-alist` instead.

To learn about display actions, see Section "Choosing a Window for Display" in `elisp`.

`magit-display-buffer-traditional` *buffer*   [Function]
This function is the current default value of the option `magit-display-buffer-function`. Before that option and this function were added, the behavior was hard-coded in many places all over the code base but now all the rules are contained in this one function (except for the "noselect" special case mentioned above).

`magit-display-buffer-same-window-except-diff-v1`   [Function]
This function displays most buffers in the currently selected window. If a buffer's mode derives from `magit-diff-mode` or `magit-process-mode`, it is displayed in another window.

`magit-display-buffer-fullframe-status-v1`   [Function]
This function fills the entire frame when displaying a status buffer. Otherwise, it behaves like `magit-display-buffer-traditional`.

`magit-display-buffer-fullframe-status-topleft-v1`   [Function]
This function fills the entire frame when displaying a status buffer. It behaves like `magit-display-buffer-fullframe-status-v1` except that it displays buffers that derive from `magit-diff-mode` or `magit-process-mode` to the top or left of the current buffer rather than to the bottom or right. As a result, Magit buffers tend to pop up on the same side as they would if `magit-display-buffer-traditional` were in use.

`magit-display-buffer-fullcolumn-most-v1`   [Function]
This function displays most buffers so that they fill the entire height of the frame. However, the buffer is displayed in another window if (1) the buffer's mode derives from `magit-process-mode`, or (2) the buffer's mode derives from `magit-diff-mode`, provided that the mode of the current buffer derives from `magit-log-mode` or `magit-cherry-mode`.

`magit-pre-display-buffer-hook`   [User Option]
This hook is run by `magit-display-buffer` before displaying the buffer.

`magit-save-window-configuration`   [Function]
This function saves the current window configuration. Later when the buffer is buried, it may be restored by `magit-restore-window-configuration`.

`magit-post-display-buffer-hook`   [User Option]
This hook is run by `magit-display-buffer` after displaying the buffer.

`magit-maybe-set-dedicated`   [Function]
> This function remembers if a new window had to be created to display the buffer,
> or whether an existing window was reused. This information is later used by
> `magit-mode-quit-window`, to determine whether the window should be deleted
> when its last Magit buffer is buried.

4.1.2 Naming Buffers

`magit-generate-buffer-name-function`   [User Option]
> The function used to generate the names of Magit buffers.
>
> Such a function should take the options `magit-uniquify-buffer-names` as well as
> `magit-buffer-name-format` into account. If it doesn't, then should be clearly stated
> in the doc-string. And if it supports %-sequences beyond those mentioned in the
> doc-string of the option `magit-buffer-name-format`, then its own doc-string should
> describe the additions.

`magit-generate-buffer-name-default-function` *mode*   [Function]
> This function returns a buffer name suitable for a buffer whose major-mode is MODE
> and which shows information about the repository in which `default-directory` is
> located.
>
> This function uses `magit-buffer-name-format` and supporting all of the
> %-sequences mentioned the documentation of that option. It also respects the option
> `magit-uniquify-buffer-names`.

`magit-buffer-name-format`   [User Option]
> The format string used to name Magit buffers.
>
> At least the following %-sequences are supported:
>
> - `%m`
>
>   The name of the major-mode, but with the `-mode` suffix removed.
>
> - `%M`
>
>   Like `%m` but abbreviate `magit-status-mode` as `magit`.
>
> - `%v`
>
>   The value the buffer is locked to, in parentheses, or an empty string if the buffer
>   is not locked to a value.
>
> - `%V`
>
>   Like `%v`, but the string is prefixed with a space, unless it is an empty string.
>
> - `%t`
>
>   The top-level directory of the working tree of the repository, or if
>   `magit-uniquify-buffer-names` is non-nil an abbreviation of that.
>
> - `%x`
>
>   If `magit-uniquify-buffer-names` is nil `"*"`, otherwise the empty string. Due
>   to limitations of the `uniquify` package, buffer names must end with the path.
>
> - `%T`
>
>   Obsolete, use `"%t%x"` instead. Like `%t`, but append an asterisk if and only if
>   `magit-uniquify-buffer-names` is nil.

The value should always contain `%m` or `%M`, `%v` or `%V`, and `%t` (or the obsolete `%T`). If `magit-uniquify-buffer-names` is non-nil, then the value must end with `%t` or `%t%x` (or the obsolete `%T`). See issue #2841.

`magit-uniquify-buffer-names`   [User Option]
This option controls whether the names of Magit buffers are uniquified. If the names are not being uniquified, then they contain the full path of the top-level of the working tree of the corresponding repository. If they are being uniquified, then they end with the basename of the top-level, or if that would conflict with the name used for other buffers, then the names of all these buffers are adjusted until they no longer conflict.

This is done using the `uniquify` package; customize its options to control how buffer names are uniquified.

4.1.3 Quitting Windows

`q`   (`magit-mode-bury-buffer`)
This command buries the current Magit buffer.

With a prefix argument, it instead kills the buffer. With a double prefix argument, also kills all other Magit buffers associated with the current project.

`magit-bury-buffer-function`   [User Option]
The function used to actually bury or kill the current buffer.

`magit-mode-bury-buffer` calls this function with one argument. If the argument is non-nil, then the function has to kill the current buffer. Otherwise it has to bury it alive. The default value currently is `magit-restore-window-configuration`.

`magit-restore-window-configuration` *kill-buffer*   [Function]
Bury or kill the current buffer using `quit-window`, which is called with KILL-BUFFER as first and the selected window as second argument.

Then restore the window configuration that existed right before the current buffer was displayed in the selected frame. Unfortunately that also means that point gets adjusted in all the buffers, which are being displayed in the selected frame.

`magit-mode-quit-window` *kill-buffer*   [Function]
Bury or kill the current buffer using `quit-window`, which is called with KILL-BUFFER as first and the selected window as second argument.

Then, if the window was originally created to display a Magit buffer and the buried buffer was the last remaining Magit buffer that was ever displayed in the window, then that is deleted.

4.1.4 Automatic Refreshing of Magit Buffers

After running a command which may change the state of the current repository, the current Magit buffer and the corresponding status buffer are refreshed. The status buffer can be automatically refreshed whenever a buffer is saved to a file inside the respective repository by adding a hook, like so:

```
(add-hook 'after-save-hook 'magit-after-save-refresh-status t)
```

Automatically refreshing Magit buffers ensures that the displayed information is up-to-date most of the time but can lead to a noticeable delay in big repositories. Other Magit

buffers are not refreshed to keep the delay to a minimum and also because doing so can sometimes be undesirable.

Buffers can also be refreshed explicitly, which is useful in buffers that weren't current during the last refresh and after changes were made to the repository outside of Magit.

g   (`magit-refresh`)

        This command refreshes the current buffer if its major mode derives from `magit-mode` as well as the corresponding status buffer.

        If the option `magit-revert-buffers` calls for it, then it also reverts all unmodified buffers that visit files being tracked in the current repository.

G   (`magit-refresh-all`)

        This command refreshes all Magit buffers belonging to the current repository and also reverts all unmodified buffers that visit files being tracked in the current repository.

        The file-visiting buffers are always reverted, even if `magit-revert-buffers` is nil.

`magit-refresh-buffer-hook`   [User Option]

    This hook is run in each Magit buffer that was refreshed during the current refresh - normally the current buffer and the status buffer.

`magit-refresh-status-buffer`   [User Option]

    When this option is non-nil, then the status buffer is automatically refreshed after running git for side-effects, in addition to the current Magit buffer, which is always refreshed automatically.

    Only set this to nil after exhausting all other options to improve performance.

`magit-after-save-refresh-status`   [Function]

    This function is intended to be added to `after-save-hook`. After doing that the corresponding status buffer is refreshed whenever a buffer is saved to a file inside a repository.

    Note that refreshing a Magit buffer is done by re-creating its contents from scratch, which can be slow in large repositories. If you are not satisfied with Magit's performance, then you should obviously not add this function to that hook.

4.1.5 Automatic Saving of File-Visiting Buffers

File-visiting buffers are by default saved at certain points in time. This doesn't guarantee that Magit buffers are always up-to-date, but, provided one only edits files by editing them in Emacs and uses only Magit to interact with Git, one can be fairly confident. When in doubt or after outside changes, type g (`magit-refresh`) to save and refresh explicitly.

`magit-save-repository-buffers`   [User Option]

    This option controls whether file-visiting buffers are saved before certain events.

    If this is non-nil then all modified file-visiting buffers belonging to the current repository may be saved before running commands, before creating new Magit buffers, and before explicitly refreshing such buffers. If this is `dontask` then this is done without user intervention. If it is `t` then the user has to confirm each save.

4.1.6 Automatic Reverting of File-Visiting Buffers

By default Magit automatically reverts buffers that are visiting files that are being tracked in a Git repository, after they have changed on disk. When using Magit one often changes files on disk by running Git, i.e. "outside Emacs", making this a rather important feature.

For example, if you discard a change in the status buffer, then that is done by running `git apply --reverse ...`, and Emacs considers the file to have "changed on disk". If Magit did not automatically revert the buffer, then you would have to type `M-x revert-buffer RET RET` in the visiting buffer before you could continue making changes.

`magit-auto-revert-mode`   [User Option]
> When this mode is enabled, then buffers that visit tracked files are automatically reverted after the visited files change on disk.

`global-auto-revert-mode`   [User Option]
> When this mode is enabled, then any file-visiting buffer is automatically reverted after the visited file changes on disk.
>
> If you like buffers that visit tracked files to be automatically reverted, then you might also like any buffer to be reverted, not just those visiting tracked files. If that is the case, then enable this mode *instead of* `magit-auto-revert-mode`.

`magit-auto-revert-immediately`   [User Option]
> This option controls whether Magit reverts buffers immediately.
>
> If this is non-nil and either `global-auto-revert-mode` or `magit-auto-revert-mode` is enabled, then Magit immediately reverts buffers by explicitly calling `auto-revert-buffers` after running Git for side-effects.
>
> If `auto-revert-use-notify` is non-nil (and file notifications are actually supported), then `magit-auto-revert-immediately` does not have to be non-nil, because the reverts happen immediately anyway.
>
> If `magit-auto-revert-immediately` and `auto-revert-use-notify` are both `nil`, then reverts happen after `auto-revert-interval` seconds of user inactivity. That is not desirable.

`auto-revert-use-notify`   [User Option]
> This option controls whether file notification functions should be used. Note that this variable unfortunately defaults to `t` even on systems on which file notifications cannot be used.

`magit-auto-revert-tracked-only`   [User Option]
> This option controls whether `magit-auto-revert-mode` only reverts tracked files or all files that are located inside Git repositories, including untracked files and files located inside Git's control directory.

`auto-revert-mode`   [User Option]
> The global mode `magit-auto-revert-mode` works by turning on this local mode in the appropriate buffers (but `global-auto-revert-mode` is implemented differently). You can also turn it on or off manually, which might be necessary if Magit does not notice that a previously untracked file now is being tracked or vice-versa.

`auto-revert-stop-on-user-input`   [User Option]

> This option controls whether the arrival of user input suspends the automatic reverts
> for `auto-revert-interval` seconds.

`auto-revert-interval`   [User Option]

> This option controls how many seconds Emacs waits for before resuming suspended
> reverts.

`auto-revert-buffer-list-filter`   [User Option]

> This option specifies an additional filter used by `auto-revert-buffers` to determine
> whether a buffer should be reverted or not.
>
> This option is provided by Magit, which also advises `auto-revert-buffers` to respect
> it. Magit users who do not turn on the local mode `auto-revert-mode` themselves,
> are best served by setting the value to `magit-auto-revert-repository-buffer-p`.
>
> However the default is nil, so as not to disturb users who do use the local mode
> directly. If you experience delays when running Magit commands, then you should
> consider using one of the predicates provided by Magit - especially if you also use
> Tramp.
>
> Users who do turn on `auto-revert-mode` in buffers in which Magit doesn't do that
> for them, should likely not use any filter. Users who turn on `global-auto-revert-`
> `mode`, do not have to worry about this option, because it is disregarded if the global
> mode is enabled.

`auto-revert-verbose`   [User Option]

> This option controls whether Emacs reports when a buffer has been reverted.

The options with the `auto-revert-` prefix are located in the Custom group named
`auto-revert`. The other, Magit-specific, options are located in the `magit` group.

Risk of Reverting Automatically

For the vast majority of users, automatically reverting file-visiting buffers after they have
changed on disk is harmless.

If a buffer is modified (i.e. it contains changes that haven't been saved yet), then Emacs
will refuse to automatically revert it. If you save a previously modified buffer, then that
results in what is seen by Git as an uncommitted change. Git will then refuse to carry
out any commands that would cause these changes to be lost. In other words, if there is
anything that could be lost, then either Git or Emacs will refuse to discard the changes.

However, if you use file-visiting buffers as a sort of ad hoc "staging area", then the
automatic reverts could potentially cause data loss. So far I have heard from only one user
who uses such a workflow.

An example: You visit some file in a buffer, edit it, and save the changes. Then, outside
of Emacs (or at least not using Magit or by saving the buffer) you change the file on disk
again. At this point the buffer is the only place where the intermediate version still exists.
You have saved the changes to disk, but that has since been overwritten. Meanwhile Emacs
considers the buffer to be unmodified (because you have not made any changes to it since
you last saved it to the visited file) and therefore would not object to it being automatically
reverted. At this point an Auto-Revert mode would kick in. It would check whether the

buffer is modified and since that is not the case it would revert it. The intermediate version would be lost. (Actually you could still get it back using the `undo` command.)

If your workflow depends on Emacs preserving the intermediate version in the buffer, then you have to disable all Auto-Revert modes. But please consider that such a workflow would be dangerous even without using an Auto-Revert mode, and should therefore be avoided. If Emacs crashes or if you quit Emacs by mistake, then you would also lose the buffer content. There would be no autosave file still containing the intermediate version (because that was deleted when you saved the buffer) and you would not be asked whether you want to save the buffer (because it isn't modified).

4.2 Sections

Magit buffers are organized into nested sections, which can be collapsed and expanded, similar to how sections are handled in Org mode. Each section also has a type, and some sections also have a value. For each section type there can also be a local keymap, shared by all sections of that type.

Taking advantage of the section value and type, many commands operate on the current section, or when the region is active and selects sections of the same type, all of the selected sections. Commands that only make sense for a particular section type (as opposed to just behaving differently depending on the type) are usually bound in section type keymaps.

4.2.1 Section Movement

To move within a section use the usual keys (`C-p`, `C-n`, `C-b`, `C-f` etc), whose global bindings are not shadowed. To move to another section use the following commands.

p   (`magit-section-backward`)
          When not at the beginning of a section, then move to the beginning of the current section. At the beginning of a section, instead move to the beginning of the previous visible section.

n   (`magit-section-forward`)
          Move to the beginning of the next visible section.

M-p   (`magit-section-backward-siblings`)
          Move to the beginning of the previous sibling section. If there is no previous sibling section, then move to the parent section instead.

M-n   (`magit-section-forward-siblings`)
          Move to the beginning of the next sibling section. If there is no next sibling section, then move to the parent section instead.

^   (`magit-section-up`)
          Move to the beginning of the parent of the current section.

The above commands all call the hook `magit-section-movement-hook`. Any of the functions listed below can be used as members of this hook.

`magit-section-movement-hook`   [Variable]
       This hook is run by all of the above movement commands, after arriving at the destination.

`magit-hunk-set-window-start`   [Function]
> This hook function ensures that the beginning of the current section is visible, pro-
> vided it is a `hunk` section. Otherwise, it does nothing. This function is a member of
> the hook's default value.

`magit-section-set-window-start`   [Function]
> This hook function ensures that the beginning of the current section is visible, regard-
> less of the section's type. If you add this to `magit-section-movement-hook`, then
> you must remove the hunk-only variant in turn.

`magit-log-maybe-show-more-commits`   [Function]
> This hook function only has an effect in log buffers, and `point` is on the "show more"
> section. If that is the case, then it doubles the number of commits that are being
> shown. This function is a member of the hook's default value.

`magit-log-maybe-update-revision-buffer`   [Function]
> When moving inside a log buffer, then this function updates the revision buffer,
> provided it is already being displayed in another window of the same frame. This
> function is a member of the hook's default value.

`magit-log-maybe-update-blob-buffer`   [Function]
> When moving inside a log buffer and another window of the same frame displays a
> blob buffer, then this function instead displays the blob buffer for the commit at point
> in that window.

`magit-status-maybe-update-revision-buffer`   [Function]
> When moving inside a status buffer, then this function updates the revision buffer,
> provided it is already being displayed in another window of the same frame.

`magit-status-maybe-update-stash-buffer`   [Function]
> When moving inside a status buffer, then this function updates the stash buffer,
> provided it is already being displayed in another window of the same frame.

`magit-status-maybe-update-blob-buffer`   [Function]
> When moving inside a status buffer and another window of the same frame displays
> a blob buffer, then this function instead displays the blob buffer for the commit at
> point in that window.

`magit-stashes-maybe-update-stash-buffer`   [Function]
> When moving inside a buffer listing stashes, then this function updates the stash
> buffer, provided it is already being displayed in another window of the same frame.

`magit-update-other-window-delay`   [User Option]
> Delay before automatically updating the other window.
>
> When moving around in certain buffers, then certain other buffers, which are being
> displayed in another window, may optionally be updated to display information about
> the section at point.
>
> When holding down a key to move by more than just one section, then that would
> update that buffer for each section on the way. To prevent that, updating the revision

buffer is delayed, and this option controls for how long. For optimal experience you might have to adjust this delay and/or the keyboard repeat rate and delay of your graphical environment or operating system.

4.2.2 Section Visibility

Magit provides many commands for changing the visibility of sections, but all you need to get started are the next two.

TAB   (`magit-section-toggle`)
> Toggle the visibility of the body of the current section.

C-<tab>   (`magit-section-cycle`)
> Cycle the visibility of current section and its children.

M-<tab>   (`magit-section-cycle-diffs`)
> Cycle the visibility of diff-related sections in the current buffer.

S-<tab>   (`magit-section-cycle-global`)
> Cycle the visibility of all sections in the current buffer.

1   (`magit-section-show-level-1`)
2   (`magit-section-show-level-2`)
3   (`magit-section-show-level-3`)
4   (`magit-section-show-level-4`)
> Show sections surrounding the current section up to level N.

M-1   (`magit-section-show-level-1-all`)
M-2   (`magit-section-show-level-2-all`)
M-3   (`magit-section-show-level-3-all`)
M-4   (`magit-section-show-level-4-all`)
> Show all sections up to level N.

Some functions, which are used to implement the above commands, are also exposed as commands themselves. By default no keys are bound to these commands, as they are generally perceived to be much less useful. But your mileage may vary.

`magit-section-show`   [Command]
> Show the body of the current section.

`magit-section-hide`   [Command]
> Hide the body of the current section.

`magit-section-show-headings`   [Command]
> Recursively show headings of children of the current section. Only show the headings. Previously shown text-only bodies are hidden.

`magit-section-show-children`   [Command]
> Recursively show the bodies of children of the current section. With a prefix argument show children down to the level of the current section, and hide deeper children.

`magit-section-hide-children`   [Command]
> Recursively hide the bodies of children of the current section.

magit-section-toggle-children   [Command]
     Toggle visibility of bodies of children of the current section.

When a buffer is first created then some sections are shown expanded while others
are not. This is hard coded. When a buffer is refreshed then the previous visibility is
preserved. The initial visibility of certain sections can also be overwritten using the hook
`magit-section-set-visibility-hook`.

magit-section-initial-visibility-alist   [User Option]
     This options can be used to override the initial visibility of sections. In the future
it will also be used to define the defaults, but currently a section's default is still
hardcoded.

     The value is an alist. Each element maps a section type or lineage to the initial
visibility state for such sections. The state has to be one of `show` or `hide`, or a
function that returns one of these symbols. A function is called with the section as
the only argument.

     Use the command `magit-describe-section-briefly` to determine a section's lin-
eage or type. The vector in the output is the section lineage and the type is the first
element of that vector. Wildcards can be used, see `magit-section-match`.

magit-section-cache-visibility   [User Option]
     This option controls for which sections the previous visibility state should be restored
if a section disappears and later appears again. The value is a boolean or a list of
section types. If t, then the visibility of all sections is cached. Otherwise this is only
done for sections whose type matches one of the listed types.

     This requires that the function `magit-section-cached-visibility` is a member of
`magit-section-set-visibility-hook`.

magit-section-set-visibility-hook   [Variable]
     This hook is run when first creating a buffer and also when refreshing an existing
buffer, and is used to determine the visibility of the section currently being inserted.

     Each function is called with one argument, the section being inserted. It should return
`hide` or `show`, or to leave the visibility undefined `nil`. If no function decides on the
visibility and the buffer is being refreshed, then the visibility is preserved; or if the
buffer is being created, then the hard coded default is used.

     Usually this should only be used to set the initial visibility but not during refreshes.
If `magit-insert-section--oldroot` is non-nil, then the buffer is being refreshed and
these functions should immediately return `nil`.

magit-section-visibility-indicator   [User Option]
     This option controls whether and how to indicate that a section can be
expanded/collapsed.

     If nil, then no visibility indicators are shown. Otherwise the value has to have one of
these two forms:

- **(EXPANDABLE-BITMAP . COLLAPSIBLE-BITMAP)**

     Both values have to be variables whose values are fringe bitmaps. In this case
every section that can be expanded or collapsed gets an indicator in the left
fringe.

To provide extra padding around the indicator, set `left-fringe-width` in `magit-mode-hook`, e.g.:

```
(add-hook 'magit-mode-hook (lambda ()
                                (setq left-fringe-width 20)))
```

- `(STRING . BOOLEAN)`

  In this case STRING (usually an ellipsis) is shown at the end of the heading of every collapsed section. Expanded sections get no indicator. The cdr controls whether the appearance of these ellipsis take section highlighting into account. Doing so might potentially have an impact on performance, while not doing so is kinda ugly.

4.2.3 Section Hooks

Which sections are inserted into certain buffers is controlled with hooks. This includes the status and the refs buffers. For other buffers, e.g. log and diff buffers, this is not possible. The command `magit-describe-section` can be used to see which hook (if any) was responsible for inserting the section at point.

For buffers whose sections can be customized by the user, a hook variable called `magit-TYPE-sections-hook` exists. This hook should be changed using `magit-add-section-hook`. Avoid using `add-hooks` or the Custom interface.

The various available section hook variables are described later in this manual along with the appropriate "section inserter functions".

`magit-add-section-hook` *hook function* **&optional** *at append local*   [Function]
    Add the function FUNCTION to the value of section hook HOOK.

    Add FUNCTION at the beginning of the hook list unless optional APPEND is non-nil, in which case FUNCTION is added at the end. If FUNCTION already is a member then move it to the new location.

    If optional AT is non-nil and a member of the hook list, then add FUNCTION next to that instead. Add before or after AT, or replace AT with FUNCTION depending on APPEND. If APPEND is the symbol `replace`, then replace AT with FUNCTION. For any other non-nil value place FUNCTION right after AT. If nil, then place FUNCTION right before AT. If FUNCTION already is a member of the list but AT is not, then leave FUNCTION where ever it already is.

    If optional LOCAL is non-nil, then modify the hook's buffer-local value rather than its global value. This makes the hook local by copying the default value. That copy is then modified.

    HOOK should be a symbol. If HOOK is void, it is first set to nil. HOOK's value must not be a single hook function. FUNCTION should be a function that takes no arguments and inserts one or multiple sections at point, moving point forward. FUNCTION may choose not to insert its section(s), when doing so would not make sense. It should not be abused for other side-effects.

To remove a function from a section hook, use `remove-hook`.

4.2.4 Section Types and Values

Each section has a type, for example `hunk`, `file`, and `commit`. Instances of certain section types also have a value. The value of a section of type `file`, for example, is a file name.

Users usually do not have to worry about a section's type and value, but knowing them can be handy at times.

M-x magit-describe-section-briefly   (magit-describe-section-briefly)
> Show information about the section at point in the echo area, as "#<magit-section VALUE [TYPE PARENT-TYPE...] BEGINNING-END>".

Many commands behave differently depending on the type of the section at point and/or somehow consume the value of that section. But that is only one of the reasons why the same key may do something different, depending on what section is current.

Additionally for each section type a keymap **might** be defined, named `magit-TYPE-section-map`. That keymap is used as text property keymap of all text belonging to any section of the respective type. If such a map does not exist for a certain type, then you can define it yourself, and it will automatically be used.

4.2.5 Section Options

This section describes options that have an effect on more than just a certain type of sections. As you can see there are not many of those.

`magit-section-show-child-count`   [User Option]
> Whether to append the number of children to section headings. This only affects sections that could benefit from this information.

4.3 Transient Commands

Many Magit commands are implemented as **transient** commands. First the user invokes a **prefix** command, which causes its **infix** arguments and **suffix** commands to be displayed in the echo area. The user then optionally sets some infix arguments and finally invokes one of the suffix commands.

This is implemented in the library `transient`. Earlier Magit releases used the package `magit-popup` and even earlier versions library `magit-key-mode`.

Transient is documented in `transient`.

C-c C-c   (magit-dispatch)
> This transient prefix command binds most of Magit's other prefix commands as suffix commands and displays them in a temporary buffer until one of them is invoked. Invoking such a sub-prefix causes the suffixes of that command to be bound and displayed instead of those of `magit-dispatch`.

This command is also, or especially, useful outside Magit buffers, so you should setup a global binding:

```
(global-set-key (kbd "C-x M-g") 'magit-dispatch)
```

4.4 Transient Arguments and Buffer Variables

The infix arguments of many of Magit's transient prefix commands cease to have an effect once the `git` command that is called with those arguments has returned. Commands that create a commit are a good example for this. If the user changes the arguments, then that only affects the next invocation of a suffix command. If the same transient prefix command is later invoked again, then the arguments are initially reset to the default value. This default value can be set for the current Emacs session or saved permanently, see Section "Saving Values" in `transient`. It is also possible to cycle through previously used sets of arguments using `M-p` and `M-n`, see Section "Using History" in `transient`.

However the infix arguments of many other transient commands continue to have an effect even after the `git` command that was called with those arguments has returned. The most important commands like this are those that display a diff or log in a dedicated buffer. Their arguments obviously continue to have an effect for as long as the respective diff or log is being displayed. Furthermore the used arguments are stored in buffer-local variables for future reference.

For commands in the second group it isn't always desirable to reset their arguments to the global value when the transient prefix command is invoked again.

As mentioned above, it is possible to cycle through previously used sets of arguments while a transient popup is visible. That means that we could always reset the infix arguments to the default because the set of arguments that is active in the existing buffer is only a few `M-p` away. Magit can be configured to behave like that, but because I expect that most users would not find that very convenient, it is not the default.

Also note that it is possible to change the diff and log arguments used in the current buffer (including the status buffer, which contains both diff and log sections) using the respective "refresh" transient prefix commands on `D` and `L`. (`d` and `l` on the other hand are intended to change **what** diff or log is being displayed. It is possible to also change **how** the diff or log is being displayed at the same time, but if you only want to do the latter, then you should use the refresh variants.) Because these secondary diff and log transient prefixes are about **changing** the arguments used in the current buffer, they **always** start out with the set of arguments that are currently in effect in that buffer.

Some commands are usually invoked directly even though they can also be invoked as the suffix of a transient prefix command. Most prominently `magit-show-commit` is usually invoked by typing `RET` while point is on a commit in a log, but it can also be invoked from the `magit-diff` transient prefix.

When such a command is invoked directly, then it is important to reuse the arguments as specified by the respective buffer-local values, instead of using the default arguments. Imagine you press `RET` in a log to display the commit at point in a different buffer and then use `D` to change how the diff is displayed in that buffer. And then you press `RET` on another commit to show that instead and the diff arguments are reset to the default. Not cool; so Magit does not do that by default.

`magit-prefix-use-buffer-arguments`   [User Option]
   This option controls whether the infix arguments initially shown in certain transient prefix commands are based on the arguments that are currently in effect in the buffer that their suffixes update.

The `magit-diff` and `magit-log` transient prefix commands are affected by this option.

`magit-direct-use-buffer-arguments`   [User Option]

This option controls whether certain commands, when invoked directly (i.e. not as the suffix of a transient prefix command), use the arguments that are currently active in the buffer that they are about to update. The alternative is to use the default value for these arguments, which might change the arguments that are used in the buffer.

Valid values for both of the above options are:

- `always`: Always use the set of arguments that is currently active in the respective buffer, provided that buffer exists of course.

- `selected` or `t`: Use the set of arguments from the respective buffer, but only if it is displayed in a window of the current frame. This is the default for both variables.

- `current`: Use the set of arguments from the respective buffer, but only if it is the current buffer.

- `never`: Never use the set of arguments from the respective buffer.

I am afraid it gets more complicated still:

- The global diff and log arguments are set for each support mode individually. The diff arguments for example have different values in `magit-diff-mode`, `magit-revision-mode`, `magit-merge-preview-mode` and `magit-status-mode` buffers. Setting or saving the value for one mode does not change the value for other modes. The history however is shared.

- When `magit-show-commit` is invoked directly from a log buffer, then the file filter is picked up from that buffer, not from the revision buffer and or the mode's global diff arguments.

- Even though they are suffixes of the diff prefix `magit-show-commit` and `magit-stash-show` do not use the diff buffer used by the diff commands, instead they use the dedicated revision and stash buffers.

  At the time you invoke the diff prefix it is unknown to Magit which of the suffix commands you are going to invoke. While not certain, more often than not users invoke one of the commands that use the diff buffer, so the initial infix arguments are those used in that buffer. However if you invoke one of these commands directly, then Magit knows that it should use the arguments from the revision resp. stash buffer.

- The log prefix also features reflog commands, but these commands do not use the log arguments.

- If `magit-show-refs` is invoked from a `magit-refs-mode` buffer, then it acts as a refresh prefix and therefore unconditionally uses the buffer's arguments as initial arguments. If it is invoked elsewhere with a prefix argument, then it acts as regular prefix and therefore respects `magit-prefix-use-buffer-arguments`. If it is invoked elsewhere without a prefix argument, then it acts as a direct command and therefore respects `magit-direct-use-buffer-arguments`.

4.5 Completion, Confirmation and the Selection

4.5.1 Action Confirmation

By default many actions that could potentially lead to data loss have to be confirmed. This includes many very common actions, so this can quickly become annoying. Many of these actions can be undone and if you have thought about how to undo certain mistakes, then it should be safe to disable confirmation for the respective actions.

The option `magit-no-confirm` can be used to tell Magit to perform certain actions without the user having to confirm them. Note that while this option can only be used to disable confirmation for a specific set of actions, the next section explains another way of telling Magit to ask fewer questions.

`magit-no-confirm`   [User Option]

The value of this option is a list of symbols, representing actions that do not have to be confirmed by the user before being carried out.

By default many potentially dangerous commands ask the user for confirmation. Each of the below symbols stands for an action which, when invoked unintentionally or without being fully aware of the consequences, could lead to tears. In many cases there are several commands that perform variations of a certain action, so we don't use the command names but more generic symbols.

- Applying changes:
  - `discard` Discarding one or more changes (i.e. hunks or the complete diff for a file) loses that change, obviously.
  - `reverse` Reverting one or more changes can usually be undone by reverting the reversion.
  - `stage-all-changes`, `unstage-all-changes` When there are both staged and unstaged changes, then un-/staging everything would destroy that distinction. Of course that also applies when un-/staging a single change, but then less is lost and one does that so often that having to confirm every time would be unacceptable.
- Files:
  - `delete` When a file that isn't yet tracked by Git is deleted, then it is completely lost, not just the last changes. Very dangerous.
  - `trash` Instead of deleting a file it can also be move to the system trash. Obviously much less dangerous than deleting it.

    Also see option `magit-delete-by-moving-to-trash`.
  - `resurrect` A deleted file can easily be resurrected by "deleting" the deletion, which is done using the same command that was used to delete the same file in the first place.
  - `untrack` Untracking a file can be undone by tracking it again.
  - `rename` Renaming a file can easily be undone.
- Sequences:
  - `reset-bisect` Aborting (known to Git as "resetting") a bisect operation loses all information collected so far.

- `abort-rebase` Aborting a rebase throws away all already modified commits, but it's possible to restore those from the reflog.
- `abort-merge` Aborting a merge throws away all conflict resolutions which have already been carried out by the user.
- `merge-dirty` Merging with a dirty worktree can make it hard to go back to the state before the merge was initiated.

- References:
  - `delete-unmerged-branch` Once a branch has been deleted, it can only be restored using low-level recovery tools provided by Git. And even then the reflog is gone. The user always has to confirm the deletion of a branch by accepting the default choice (or selecting another branch), but when a branch has not been merged yet, also make sure the user is aware of that.
  - `delete-pr-remote` When deleting a branch that was created from a pull-request and if no other branches still exist on that remote, then 'magit-branch-delete' offers to delete the remote as well. This should be safe because it only happens if no other refs exist in the remotes namespace, and you can recreate the remote if necessary.
  - `drop-stashes` Dropping a stash is dangerous because Git stores stashes in the reflog. Once a stash is removed, there is no going back without using low-level recovery tools provided by Git. When a single stash is dropped, then the user always has to confirm by accepting the default (or selecting another). This action only concerns the deletion of multiple stashes at once.

- Edit published history:

  Without adding these symbols here, you will be warned before editing commits that have already been pushed to one of the branches listed in `magit-published-branches`.
  - `amend-published` Affects most commands that amend to "HEAD".
  - `rebase-published` Affects commands that perform interactive rebases. This includes commands from the commit transient that modify a commit other than "HEAD", namely the various fixup and squash variants.
  - `edit-published` Affects the commands `magit-edit-line-commit` and `magit-diff-edit-hunk-commit`. These two commands make it quite easy to accidentally edit a published commit, so you should think twice before configuring them not to ask for confirmation.

  To disable confirmation completely, add all three symbols here or set `magit-published-branches` to `nil`.

- Various:
  - `kill-process` There seldom is a reason to kill a process.

- Global settings:

  Instead of adding all of the above symbols to the value of this option, you can also set it to the atom 't', which has the same effect as adding all of the above symbols. Doing that most certainly is a bad idea, especially because other symbols might be added in the future. So even if you don't want to be asked for confirmation for

any of these actions, you are still better of adding all of the respective symbols individually.

When `magit-wip-before-change-mode` is enabled, then the following actions can be undone fairly easily: `discard`, `reverse`, `stage-all-changes`, and `unstage-all-changes`. If and only if this mode is enabled, then `safe-with-wip` has the same effect as adding all of these symbols individually.

4.5.2 Completion and Confirmation

Many Magit commands ask the user to select from a list of possible things to act on, while offering the most likely choice as the default. For many of these commands the default is the thing at point, provided that it actually is a valid thing to act on. For many commands that act on a branch, the current branch serves as the default if there is no branch at point.

These commands combine asking for confirmation and asking for a target to act on into a single action. The user can confirm the default target using `RET` or abort using `C-g`. This is similar to a `y-or-n-p` prompt, but the keys to confirm or abort differ.

At the same time the user is also given the opportunity to select another target, which is useful because for some commands and/or in some situations you might want to select the action before selecting the target by moving to it.

However you might find that for some commands you always want to use the default target, if any, or even that you want the command to act on the default without requiring any confirmation at all. The option `magit-dwim-selection` can be used to configure certain commands to that effect.

Note that when the region is active then many commands act on the things that are selected using a mechanism based on the region, in many cases after asking for confirmation. This region-based mechanism is called the "selection" and is described in detail in the next section. When a selection exists that is valid for the invoked command, then that command never offers to act on something else, and whether it asks for confirmation is not controlled by this option.

Also note that Magit asks for confirmation of certain actions that are not coupled with completion (or the selection). Such dialogs are also not affected by this option and are described in the previous section.

`magit-dwim-selection`   [User Option]

This option can be used to tell certain commands to use the thing at point instead of asking the user to select a candidate to act on, with or without confirmation.

The value has the form `((COMMAND nil|PROMPT DEFAULT)...)`.

- COMMAND is the command that should not prompt for a choice. To have an effect, the command has to use the function `magit-completing-read` or a utility function which in turn uses that function.
- If the command uses `magit-completing-read` multiple times, then PROMPT can be used to only affect one of these uses. PROMPT, if non-nil, is a regular expression that is used to match against the PROMPT argument passed to `magit-completing-read`.
- DEFAULT specifies how to use the default. If it is `t`, then the DEFAULT argument passed to `magit-completing-read` is used without confirmation. If it is `ask`, then the user is given a chance to abort. DEFAULT can also be `nil`, in which case the entry has no effect.

4.5.3 The Selection

If the region is active, then many Magit commands act on the things that are selected using a mechanism based on the region instead of one single thing. When the region is not active, then these commands act on the thing at point or read a single thing to act on. This is described in the previous section — this section only covers how multiple things are selected, how that is visualized, and how certain commands behave when that is the case.

Magit's mechanism for selecting multiple things, or rather sections that represent these things, is based on the Emacs region, but the area that Magit considers to be selected is typically larger than the region and additional restrictions apply.

Magit makes a distinction between a region that qualifies as forming a valid Magit selection and a region that does not. If the region does not qualify, then it is displayed as it is in other Emacs buffers. If the region does qualify as a Magit selection, then the selection is always visualized, while the region itself is only visualized if it begins and ends on the same line.

For a region to qualify as a Magit selection, it must begin in the heading of one section and end in the heading of a sibling section. Note that if the end of the region is at the very beginning of section heading (i.e. at the very beginning of a line) then that section is considered to be **inside** the selection.

This is not consistent with how the region is normally treated in Emacs — if the region ends at the beginning of a line, then that line is outside the region. Due to how Magit visualizes the selection, it should be obvious that this difference exists.

Not every command acts on every valid selection. Some commands do not even consider the location of point, others may act on the section at point but not support acting on the selection, and even commands that do support the selection of course only do so if it selects things that they can act on.

This is the main reason why the selection must include the section at point. Even if a selection exists, the invoked command may disregard it, in which case it may act on the current section only. It is much safer to only act on the current section but not the other selected sections than it is to act on the current section **instead** of the selected sections. The latter would be much more surprising and if the current section always is part of the selection, then that cannot happen.

`magit-keep-region-overlay`   [Variable]
>   This variable controls whether the region is visualized as usual even when a valid Magit selection or a hunk-internal region exists. See the doc-string for more information.

4.5.4 The hunk-internal region

Somewhat related to the Magit selection described in the previous section is the hunk-internal region.

Like the selection, the hunk-internal region is based on the Emacs region but causes that region to not be visualized as it would in other Emacs buffers, and includes the line on which the region ends even if it ends at the very beginning of that line.

Unlike the selection, which is based on a region that must begin in the heading of one section and ends in the section of a sibling section, the hunk-internal region must begin inside the **body** of a hunk section and end in the body of the **same** section.

The hunk-internal region is honored by "apply" commands, which can, among other targets, act on a hunk. If the hunk-internal region is active, then such commands act only on the marked part of the hunk instead of on the complete hunk.

4.5.5 Support for Completion Frameworks

The built-in option `completing-read-function` specifies the low-level function used by `completing-read` to ask a user to select from a list of choices. Its default value is `completing-read-default`. Alternative completion frameworks typically activate themselves by substituting their own implementation.

Mostly for historic reasons Magit provides a similar option named `magit-completing-read-function`, which only controls the low-level function used by `magit-completing-read`. This option also makes it possible to use a different completing mechanism for Magit than for the rest of Emacs, but doing that is not recommend.

You most likely don't have to customize the magit-specific option to use an alternative completion framework. For example, if you enable `ivy-mode`, then Magit will respect that, and if you enable `helm-mode`, then you are done too.

However if you want to use Ido, then `ido-mode` won't do the trick. You will also have to install the `ido-completing-read+` package and use `magit-ido-completing-read` as `magit-completing-read-function`.

`magit-completing-read-function`   [User Option]
    The value of this variable is the low-level function used to perform completion by code that uses `magit-completing-read` (as opposed to the built-in `completing-read`).

    The default value, `magit-builtin-completing-read`, is suitable for the standard completion mechanism, `ivy-mode`, and `helm-mode` at least.

    The built-in `completing-read` and `completing-read-default` are **not** suitable to be used here. `magit-builtin-completing-read` performs some additional work, and any function used in its place has to do the same.

`magit-builtin-completing-read` *prompt choices* **&optional**   [Function]
    predicate require-match initial-input hist def
    This function performs completion using the built-in `completing-read` and does some additional magit-specific work.

`magit-ido-completing-read` *prompt choices* **&optional** *predicate*   [Function]
    require-match initial-input hist def
    This function performs completion using `ido-completing-read+` from the package by the same name (which you have to explicitly install) and does some additional magit-specific work.

    We have to use `ido-completing-read+` instead of the `ido-completing-read` that comes with Ido itself, because the latter, while intended as a drop-in replacement, cannot serve that purpose because it violates too many of the implicit conventions.

`magit-completing-read` *prompt choices* **&optional** *predicate*   [Function]
        require-match initial-input hist def fallback

>   This is the function that Magit commands use when they need the user to select a single thing to act on. The arguments have the same meaning as for `completing-read`, except for FALLBACK, which is unique to this function and is described below.
>
>   Instead of asking the user to choose from a list of possible candidates, this function may just return the default specified by DEF, with or without requiring user confirmation. Whether that is the case depends on PROMPT, `this-command` and `magit-dwim-selection`. See the documentation of the latter for more information.
>
>   If it does read a value in the minibuffer, then this function acts similar to `completing-read`, except for the following:
>
>   - COLLECTION must be a list of choices. A function is not supported.
>   - If REQUIRE-MATCH is `nil` and the user exits without a choice, then `nil` is returned instead of an empty string.
>   - If REQUIRE-MATCH is non-nil and the users exits without a choice, an user-error is raised.
>   - FALLBACK specifies a secondary default that is only used if the primary default DEF is `nil`. The secondary default is not subject to `magit-dwim-selection` — if DEF is `nil` but FALLBACK is not, then this function always asks the user to choose a candidate, just as if both defaults were `nil`.
>   - ": " is appended to PROMPT.
>   - PROMPT is modified to end with \" (default DEF|FALLBACK): \" provided that DEF or FALLBACK is non-nil, that neither `ivy-mode` nor `helm-mode` is enabled, and that `magit-completing-read-function` is set to its default value of `magit-builtin-completing-read`.

4.5.6 Additional Completion Options

`magit-list-refs-sortby`   [User Option]

>   For many commands that read a ref or refs from the user, the value of this option can be used to control the order of the refs. Valid values include any key accepted by the `--sort` flag of `git for-each-ref`. By default, refs are sorted alphabetically by their full name (e.g., "refs/heads/master").

4.6 Running Git

4.6.1 Viewing Git Output

Magit runs Git either for side-effects (e.g. when pushing) or to get some value (e.g. the name of the current branch).

When Git is run for side-effects, the process output is logged in a per-repository log buffer, which can be consulted using the `magit-process` command when things don't go as expected.

The output/errors for up to 'magit-process-log-max' Git commands are retained.

$ (`magit-process`)

>   This commands displays the process buffer for the current repository.

Inside that buffer, the usual key bindings for navigating and showing sections are available. There is one additional command.

`k`   (`magit-process-kill`)

> This command kills the process represented by the section at point.

`magit-git-debug`   [User Option]

> When this is non-nil then the output of all calls to git are logged in the process buffer. This is useful when debugging, otherwise it just negatively affects performance.

4.6.2 Git Process Status

When a Git process is running for side-effects, Magit displays an indicator in the mode line, using the `magit-mode-line-process` face.

If the Git process exits successfully, the process indicator is removed from the mode line immediately.

In the case of a Git error, the process indicator is not removed, but is instead highlighted with the `magit-mode-line-process-error` face, and the error details from the process buffer are provided as a tooltip for mouse users. This error indicator persists in the mode line until the next magit buffer refresh.

If you do not wish process errors to be indicated in the mode line, customize the `magit-process-display-mode-line-error` user option.

Process errors are additionally indicated at the top of the status buffer.

4.6.3 Running Git Manually

While Magit provides many Emacs commands to interact with Git, it does not cover everything. In those cases your existing Git knowledge will come in handy. Magit provides some commands for running arbitrary Git commands by typing them into the minibuffer, instead of having to switch to a shell.

`!`   (`magit-run`)

> This transient prefix command binds the following suffix commands and displays them in a temporary buffer until a suffix is invoked.

`! !`   (`magit-git-command-topdir`)

> This command reads a command from the user and executes it in the top-level directory of the current working tree.
>
> The string "git " is used as initial input when prompting the user for the command. It can be removed to run another command.

`! p`   (`magit-git-command`)

> This command reads a command from the user and executes it in `default-directory`. With a prefix argument the command is executed in the top-level directory of the current working tree instead.
>
> The string "git " is used as initial input when prompting the user for the command. It can be removed to run another command.

`! s`   (`magit-shell-command-topdir`)

> This command reads a command from the user and executes it in the top-level directory of the current working tree.

! S   (magit-shell-command)
> This command reads a command from the user and executes it in `default-directory`. With a prefix argument the command is executed in the top-level directory of the current working tree instead.

`magit-shell-command-verbose-prompt`   [User Option]
> Whether the prompt, used by the above commands when reading a shell command, shows the directory in which it will be run.

These suffix commands start external gui tools.

! k   (magit-run-gitk)
> This command runs `gitk` in the current repository.

! a   (magit-run-gitk-all)
> This command runs `gitk --all` in the current repository.

! b   (magit-run-gitk-branches)
> This command runs `gitk --branches` in the current repository.

! g   (magit-run-git-gui)
> This command runs `git gui` in the current repository.

4.6.4 Git Executable

Except on MS Windows, Magit defaults to running Git without specifying the path to the git executable. Instead the first executable found by Emacs on `exec-path` is used (whose value in turn is set based on the value of the environment variable `$PATH` when Emacs was started).

This has the advantage that it continues to work even when using Tramp to connect to a remote machine on which the executable is found in a different place. The downside is that if you have multiple versions of Git installed, then you might end up using another version than the one you think you are using.

M-x magit-version   (magit-version)
> This command shows the currently used versions of Magit, Git, and Emacs in the echo area. Non-interactively this just returns the Magit version.

When the `system-type` is `windows-nt`, then `magit-git-executable` is set to an absolute path when Magit is first loaded. This is necessary because Git on that platform comes with several wrapper scripts for the actual git binary, which are also placed on `$PATH`, and using one of these wrappers instead of the binary would degrade performance horribly.

If Magit doesn't find the correct executable then you **can** work around that by setting `magit-git-executable` to an absolute path. But note that doing so is a kludge. It is better to make sure the order in the environment variable `$PATH` is correct, and that Emacs is started with that environment in effect. The command `magit-debug-git-executable` can be useful to find out where Emacs is searching for git. If you have to connect from Windows to a non-Windows machine, then you must change the value to "git".

`magit-git-executable`   [User Option]
> The git executable used by Magit, either the full path to the executable or the string "git" to let Emacs find the executable itself, using the standard mechanism for doing such things.

M-x magit-debug-git-executable   (magit-debug-git-executable)
>   Display a buffer with information about `magit-git-executable`.

4.6.5 Global Git Arguments

`magit-git-global-arguments`   [User Option]
>   The arguments set here are used every time the git executable is run as a subprocess.
>   They are placed right after the executable itself and before the git command - as in
>   `git HERE... COMMAND REST`. For valid arguments see the git(1) manpage.
>
>   Be careful what you add here, especially if you are using Tramp to connect to servers
>   with ancient Git versions. Never remove anything that is part of the default value,
>   unless you really know what you are doing. And think very hard before adding
>   something; it will be used every time Magit runs Git for any purpose.

5 Inspecting

The functionality provided by Magit can be roughly divided into three groups: inspecting existing data, manipulating existing data or adding new data, and transferring data. Of course that is a rather crude distinction that often falls short, but it's more useful than no distinction at all. This section is concerned with inspecting data, the next two with manipulating and transferring it. Then follows a section about miscellaneous functionality, which cannot easily be fit into this distinction.

Of course other distinctions make sense too, e.g. Git's distinction between porcelain and plumbing commands, which for the most part is equivalent to Emacs' distinction between interactive commands and non-interactive functions. All of the sections mentioned before are mainly concerned with the porcelain – Magit's plumbing layer is described later.

5.1 Status Buffer

While other Magit buffers contain e.g. one particular diff or one particular log, the status buffer contains the diffs for staged and unstaged changes, logs for unpushed and unpulled commits, lists of stashes and untracked files, and information related to the current branch.

During certain incomplete operations – for example when a merge resulted in a conflict – additional information is displayed that helps proceeding with or aborting the operation.

The command `magit-status` displays the status buffer belonging to the current repository in another window. This command is used so often that it should be bound globally. We recommend using `C-x g`:

```
(global-set-key (kbd "C-x g") 'magit-status)
```

C-x g (magit-status)

> When invoked from within an existing Git repository, then this command shows the status of that repository in a buffer.
>
> If the current directory isn't located within a Git repository, then this command prompts for an existing repository or an arbitrary directory, depending on the option `magit-repository-directories`, and the status for the selected repository is shown instead.
>
> - If that option specifies any existing repositories, then the user is asked to select one of them.
> - Otherwise the user is asked to select an arbitrary directory using regular file-name completion. If the selected directory is the top-level directory of an existing working tree, then the status buffer for that is shown.
> - Otherwise the user is offered to initialize the selected directory as a new repository. After creating the repository its status buffer is shown.
>
> These fallback behaviors can also be forced using one or more prefix arguments:
>
> - With two prefix arguments (or more precisely a numeric prefix value of 16 or greater) an arbitrary directory is read, which is then acted on as described above. The same could be accomplished using the command `magit-init`.

- With a single prefix argument an existing repository is read from the user,
  or if no repository can be found based on the value of `magit-repository-directories`, then the behavior is the same as with two prefix arguments.

`magit-repository-directories`   [User Option]

   List of directories that are Git repositories or contain Git repositories.

   Each element has the form (`DIRECTORY . DEPTH`). DIRECTORY has to be a directory
   or a directory file-name, a string. DEPTH, an integer, specifies the maximum depth
   to look for Git repositories. If it is 0, then only add DIRECTORY itself.

   This option controls which repositories are being listed by `magit-list-repositories`. It also affects `magit-status` (which see) in potentially surprising
   ways (see above).

`ido-enter-magit-status`   [Command]

   From an Ido prompt used to open a file, instead drop into `magit-status`. This is
   similar to `ido-magic-delete-char`, which, despite its name, usually causes a Dired
   buffer to be created.

   To make this command available, use something like:

```
(add-hook 'ido-setup-hook
          (lambda ()
            (define-key ido-completion-map
              (kbd \"C-x g\") 'ido-enter-magit-status)))
```

   Starting with Emacs 25.1 the Ido keymaps are defined just once instead of every time
   Ido is invoked, so now you can modify it like pretty much every other keymap:

```
(define-key ido-common-completion-map
  (kbd \"C-x g\") 'ido-enter-magit-status)
```

5.1.1 Status Sections

The contents of status buffers is controlled using the hook `magit-status-sections-hook`.
See Section 4.2.3 [Section Hooks], page 18, to learn about such hooks and how to customize
them.

`magit-status-sections-hook`   [User Option]

   Hook run to insert sections into a status buffer.

   The first function on that hook by default is `magit-insert-status-headers`; it is described in the next section. By default the following functions are also members of that
hook:

`magit-insert-merge-log`   [Function]

   Insert section for the on-going merge. Display the heads that are being merged. If
   no merge is in progress, do nothing.

`magit-insert-rebase-sequence`   [Function]

   Insert section for the on-going rebase sequence. If no such sequence is in progress, do
   nothing.

`magit-insert-am-sequence`   [Function]
    Insert section for the on-going patch applying sequence. If no such sequence is in progress, do nothing.

`magit-insert-sequencer-sequence`   [Function]
    Insert section for the on-going cherry-pick or revert sequence. If no such sequence is in progress, do nothing.

`magit-insert-bisect-output`   [Function]
    While bisecting, insert section with output from `git bisect`.

`magit-insert-bisect-rest`   [Function]
    While bisecting, insert section visualizing the bisect state.

`magit-insert-bisect-log`   [Function]
    While bisecting, insert section logging bisect progress.

`magit-insert-untracked-files`   [Function]
    Maybe insert a list or tree of untracked files.

    Do so depending on the value of `status.showUntrackedFiles`. Note that even if the value is `all`, Magit still initially only shows directories. But the directory sections can then be expanded using `TAB`.

`magit-insert-unstaged-changes`   [Function]
    Insert section showing unstaged changes.

`magit-insert-staged-changes`   [Function]
    Insert section showing staged changes.

`magit-insert-stashes` **&optional** *ref heading*   [Function]
    Insert the `stashes` section showing reflog for "refs/stash". If optional REF is non-nil show reflog for that instead. If optional HEADING is non-nil use that as section heading instead of "Stashes:".

`magit-insert-unpulled-from-upstream`   [Function]
    Insert section showing commits that haven't been pulled from the upstream branch yet.

`magit-insert-unpulled-from-pushremote`   [Function]
    Insert section showing commits that haven't been pulled from the push-remote branch yet.

`magit-insert-unpushed-to-upstream`   [Function]
    Insert section showing commits that haven't been pushed to the upstream yet.

`magit-insert-unpushed-to-pushremote`   [Function]
    Insert section showing commits that haven't been pushed to the push-remote yet.

    The following functions can also be added to the above hook:

`magit-insert-tracked-files`   [Function]
    Insert a tree of tracked files.

`magit-insert-ignored-files`   [Function]
>   Insert a tree of ignored files. Its possible to limit the logs in the current buffer to a
>   certain directory using `D = f <DIRECTORY> RET g`. If you do that, then that that also
>   affects this command.
>
>   The log filter can be used to limit to multiple files. In that case this function only
>   respects the first of the files and only if it is a directory.

`magit-insert-skip-worktree-files`   [Function]
>   Insert a tree of skip-worktree files. If the first element of `magit-buffer-diff-files`
>   is a directory, then limit the list to files below that. The value of that variable can
>   be set using `D -- DIRECTORY RET g`.

`magit-insert-assumed-unchanged-files`   [Function]
>   Insert a tree of files that are assumed to be unchanged. If the first element of
>   `magit-buffer-diff-files` is a directory, then limit the list to files below that. The
>   value of that variable can be set using `D -- DIRECTORY RET g`.

`magit-insert-unpulled-or-recent-commits`   [Function]
>   Insert section showing unpulled or recent commits. If an upstream is configured for
>   the current branch and it is ahead of the current branch, then show the missing
>   commits. Otherwise, show the last `magit-log-section-commit-count` commits.

`magit-insert-recent-commits`   [Function]
>   Insert section showing the last `magit-log-section-commit-count` commits.

`magit-log-section-commit-count`   [User Option]
>   How many recent commits `magit-insert-recent-commits` and `magit-insert-`
>   `unpulled-or-recent-commits` (provided there are no unpulled commits)
>   show.

`magit-insert-unpulled-cherries`   [Function]
>   Insert section showing unpulled commits. Like `magit-insert-unpulled-commits`
>   but prefix each commit that has not been applied yet (i.e. a commit with a patch-id
>   not shared with any local commit) with "+", and all others with "-".

`magit-insert-unpushed-cherries`   [Function]
>   Insert section showing unpushed commits. Like `magit-insert-unpushed-commits`
>   but prefix each commit which has not been applied to upstream yet (i.e. a commit
>   with a patch-id not shared with any upstream commit) with "+" and all others with
>   "-".

   See Section 5.6 [References Buffer], page 52, for some more section inserters, which could
be used here.

5.1.2 Status Header Sections

The contents of status buffers is controlled using the hook `magit-status-sections-hook`
(see Section 5.1.1 [Status Sections], page 32).

   By default `magit-insert-status-headers` is the first member of that hook variable.

`magit-insert-status-headers`   [Function]
    Insert headers sections appropriate for `magit-status-mode` buffers. The sections are inserted by running the functions on the hook `magit-status-headers-hook`.

`magit-status-headers-hook`   [User Option]
    Hook run to insert headers sections into the status buffer.

    This hook is run by `magit-insert-status-headers`, which in turn has to be a member of `magit-status-sections-hook` to be used at all.

By default the following functions are members of the above hook:

`magit-insert-error-header`   [Function]
    Insert a header line showing the message about the Git error that just occurred.

    This function is only aware of the last error that occur when Git was run for side-effects. If, for example, an error occurs while generating a diff, then that error won't be inserted. Refreshing the status buffer causes this section to disappear again.

`magit-insert-diff-filter-header`   [Function]
    Insert a header line showing the effective diff filters.

`magit-insert-head-branch-header`   [Function]
    Insert a header line about the current branch or detached `HEAD`.

`magit-insert-upstream-branch-header`   [Function]
    Insert a header line about the branch that is usually pulled into the current branch.

`magit-insert-push-branch-header`   [Function]
    Insert a header line about the branch that the current branch is usually pushed to.

`magit-insert-tags-header`   [Function]
    Insert a header line about the current and/or next tag, along with the number of commits between the tag and `HEAD`.

The following functions can also be added to the above hook:

`magit-insert-repo-header`   [Function]
    Insert a header line showing the path to the repository top-level.

`magit-insert-remote-header`   [Function]
    Insert a header line about the remote of the current branch.

    If no remote is configured for the current branch, then fall back showing the "origin" remote, or if that does not exist the first remote in alphabetic order.

`magit-insert-user-header`   [Function]
    Insert a header line about the current user.

5.1.3 Status Module Sections

The contents of status buffers is controlled using the hook `magit-status-sections-hook` (see Section 5.1.1 [Status Sections], page 32).

By default `magit-insert-modules` is *not* a member of that hook variable.

`magit-insert-modules`   [Function]
> Insert submodule sections.
>
> Hook `magit-module-sections-hook` controls which module sections are inserted, and option `magit-module-sections-nested` controls whether they are wrapped in an additional section.

`magit-module-sections-hook`   [User Option]
> Hook run by `magit-insert-modules`.

`magit-module-sections-nested`   [User Option]
> This option controls whether `magit-insert-modules` wraps inserted sections in an additional section.
>
> If this is non-nil, then only a single top-level section is inserted. If it is nil, then all sections listed in `magit-module-sections-hook` become top-level sections.

`magit-insert-modules-overview`   [Function]
> Insert sections for all submodules. For each section insert the path, the branch, and the output of `git describe --tags`, or, failing that, the abbreviated HEAD commit hash.
>
> Press RET on such a submodule section to show its own status buffer. Press RET on the "Modules" section to display a list of submodules in a separate buffer. This shows additional information not displayed in the super-repository's status buffer.

`magit-insert-modules-unpulled-from-upstream`   [Function]
> Insert sections for modules that haven't been pulled from the upstream yet. These sections can be expanded to show the respective commits.

`magit-insert-modules-unpulled-from-pushremote`   [Function]
> Insert sections for modules that haven't been pulled from the push-remote yet. These sections can be expanded to show the respective commits.

`magit-insert-modules-unpushed-to-upstream`   [Function]
> Insert sections for modules that haven't been pushed to the upstream yet. These sections can be expanded to show the respective commits.

`magit-insert-modules-unpushed-to-pushremote`   [Function]
> Insert sections for modules that haven't been pushed to the push-remote yet. These sections can be expanded to show the respective commits.

5.1.4 Status Options

`magit-status-refresh-hook`   [User Option]
> Hook run after a status buffer has been refreshed.

`magit-status-margin`   [User Option]

This option specifies whether the margin is initially shown in Magit-Status mode buffers and how it is formatted.

The value has the form (`INIT STYLE WIDTH AUTHOR AUTHOR-WIDTH`).

- If INIT is non-nil, then the margin is shown initially.
- STYLE controls how to format the committer date. It can be one of `age` (to show the age of the commit), `age-abbreviated` (to abbreviate the time unit to a character), or a string (suitable for `format-time-string`) to show the actual date.
- WIDTH controls the width of the margin. This exists for forward compatibility and currently the value should not be changed.
- AUTHOR controls whether the name of the author is also shown by default.
- AUTHOR-WIDTH has to be an integer. When the name of the author is shown, then this specifies how much space is used to do so.

`magit-log-section-args`   [User Option]

Additional Git arguments used when creating log sections. Only `--graph`, `--decorate`, and `--show-signature` are supported. This option is only a temporary kludge and will be removed.

Note that due to an issue in Git the use of `--graph` is very slow with long histories, so you probably don't want to add this here.

Also see the proceeding section for more options concerning status buffers.

5.2 Repository List

`magit-list-repositories`   [Command]

This command displays a list of repositories in a separate buffer.

The options `magit-repository-directories` and `magit-repository-directories-depth` control which repositories are displayed.

`magit-repolist-columns`   [User Option]

This option controls what columns are displayed by the command `magit-list-repositories` and how they are displayed.

Each element has the form (`HEADER WIDTH FORMAT PROPS`).

HEADER is the string displayed in the header. WIDTH is the width of the column. FORMAT is a function that is called with one argument, the repository identification (usually its basename), and with `default-directory` bound to the toplevel of its working tree. It has to return a string to be inserted or nil. PROPS is an alist that supports the keys `:right-align` and `:pad-right`.

The following functions can be added to the above option:

`magit-repolist-column-ident`   [Function]

This function inserts the identification of the repository. Usually this is just its basename.

`magit-repolist-column-path`   [Function]
    This function inserts the absolute path of the repository.

`magit-repolist-column-version`   [Function]
    This function inserts a description of the repository's `HEAD` revision.

`magit-repolist-column-branch`   [Function]
    This function inserts the name of the current branch.

`magit-repolist-column-upstream`   [Function]
    This function inserts the name of the upstream branch of the current branch.

`magit-repolist-column-branches`   [Function]
    This function inserts the number of branches.

`magit-repolist-column-stashes`   [Function]
    This function inserts the number of stashes.

`magit-repolist-column-flag`   [Function]
    This function inserts a flag as specified by `magit-repolist-column-flag-alist`.

    By default this indicates whether there are uncommitted changes.

       • `N` if there is at least one untracked file.
       • `U` if there is at least one unstaged file.
       • `S` if there is at least one staged file.

    Only the first one of these that applies is shown.

`magit-repolist-column-unpulled-from-upstream`   [Function]
    This function inserts the number of upstream commits not in the current branch.

`magit-repolist-column-unpulled-from-pushremote`   [Function]
    This function inserts the number of commits in the push branch but not the current
    branch.

`magit-repolist-column-unpushed-to-upstream`   [Function]
    This function inserts the number of commits in the current branch but not its up-
    stream.

`magit-repolist-column-unpushed-to-pushremote`   [Function]
    This function inserts the number of commits in the current branch but not its push
    branch.

5.3 Logging

The status buffer contains logs for the unpushed and unpulled commits, but that obvi-
ously isn't enough. The transient prefix command `magit-log`, on `l`, features several suffix
commands, which show a specific log in a separate log buffer.

    Like other transient prefix commands, `magit-log` also features several infix arguments
that can be changed before invoking one of the suffix commands. However, in the case of
the log transient, these arguments may be taken from those currently in use in the current

repository's log buffer, depending on the value of `magit-prefix-use-buffer-arguments`
(see Section 4.4 [Transient Arguments and Buffer Variables], page 20).

For information about the various arguments, see the git-log(1) manpage.

The switch `++order=VALUE` is converted to one of `--author-date-order`, `--date-order`,
or `--topo-order` before being passed to `git log`.

The log transient also features several reflog commands. See Section 5.3.5 [Reflog],
page 43.

`l` (magit-log)
>   This transient prefix command binds the following suffix commands along with
>   the appropriate infix arguments and displays them in a temporary buffer until
>   a suffix is invoked.

`l l` (magit-log-current)
>   Show log for the current branch. When `HEAD` is detached or with a prefix
>   argument, show log for one or more revs read from the minibuffer.

`l o` (magit-log-other)
>   Show log for one or more revs read from the minibuffer. The user can input any
>   revision or revisions separated by a space, or even ranges, but only branches,
>   tags, and a representation of the commit at point are available as completion
>   candidates.

`l h` (magit-log-head)
>   Show log for `HEAD`.

`l L` (magit-log-branches)
>   Show log for all local branches and `HEAD`.

`l b` (magit-log-all-branches)
>   Show log for all local and remote branches and `HEAD`.

`l a` (magit-log-all)
>   Show log for all references and `HEAD`.

Two additional commands that show the log for the file or blob that is being visited in
the current buffer exists, see Section 8.8 [Minor Mode for Buffers Visiting Files], page 109.
The command `magit-cherry` also shows a log, see Section 5.3.6 [Cherries], page 44.

5.3.1 Refreshing Logs

The transient prefix command `magit-log-refresh`, on L, can be used to change the log
arguments used in the current buffer, without changing which log is shown. This works in
dedicated log buffers, but also in the status buffer.

`L` (magit-log-refresh)
>   This transient prefix command binds the following suffix commands along with
>   the appropriate infix arguments and displays them in a temporary buffer until
>   a suffix is invoked.

`L g` (magit-log-refresh)
>   This suffix command sets the local log arguments for the current buffer.

`L s`   (`magit-log-set-default-arguments`)
>   This suffix command sets the default log arguments for buffers of the same type
>   as that of the current buffer. Other existing buffers of the same type are not
>   affected because their local values have already been initialized.

`L w`   (`magit-log-save-default-arguments`)
>   This suffix command sets the default log arguments for buffers of the same
>   type as that of the current buffer, and saves the value for future sessions. Other
>   existing buffers of the same type are not affected because their local values have
>   already been initialized.

`L t`   (`magit-toggle-margin`)
>   Show or hide the margin.

5.3.2 Log Buffer

`L`   (`magit-log-refresh`)
>   This transient prefix command binds the following suffix commands along with
>   the appropriate infix arguments and displays them in a temporary buffer until
>   a suffix is invoked.
>
>   See Section 5.3.1 [Refreshing Logs], page 39.

`q`   (`magit-log-bury-buffer`)
>   Bury the current buffer or the revision buffer in the same frame. Like
>   `magit-mode-bury-buffer` (which see) but with a negative prefix argument
>   instead bury the revision buffer, provided it is displayed in the current frame.

`C-c C-b`   (`magit-go-backward`)
>   Move backward in current buffer's history.

`C-c C-f`   (`magit-go-forward`)
>   Move forward in current buffer's history.

`C-c C-n`   (`magit-log-move-to-parent`)
>   Move to a parent of the current commit. By default, this is the first parent,
>   but a numeric prefix can be used to specify another parent.

`SPC`   (`magit-diff-show-or-scroll-up`)
>   Update the commit or diff buffer for the thing at point.
>
>   Either show the commit or stash at point in the appropriate buffer, or if that
>   buffer is already being displayed in the current frame and contains information
>   about that commit or stash, then instead scroll the buffer up. If there is no
>   commit or stash at point, then prompt for a commit.

`DEL`   (`magit-diff-show-or-scroll-down`)
>   Update the commit or diff buffer for the thing at point.
>
>   Either show the commit or stash at point in the appropriate buffer, or if that
>   buffer is already being displayed in the current frame and contains information
>   about that commit or stash, then instead scroll the buffer down. If there is no
>   commit or stash at point, then prompt for a commit.

=   (magit-log-toggle-commit-limit)
         Toggle the number of commits the current log buffer is limited to. If the number
         of commits is currently limited, then remove that limit. Otherwise set it to 256.

+   (magit-log-double-commit-limit)
         Double the number of commits the current log buffer is limited to.

-    (magit-log-half-commit-limit)
         Half the number of commits the current log buffer is limited to.

magit-log-auto-more   [User Option]
     Insert more log entries automatically when moving past the last entry. Only consid-
     ered when moving past the last entry with `magit-goto-*-section` commands.

magit-log-show-refname-after-summary   [User Option]
     Whether to show the refnames after the commit summaries. This is useful if you use
     really long branch names.

     Magit displays references in logs a bit differently from how Git does it.

     Local branches are blue and remote branches are green. Of course that depends on the
used theme, as do the colors used for other types of references. The current branch has a
box around it, as do remote branches that are their respective remote's `HEAD` branch.

     If a local branch and its push-target point at the same commit, then their names are
combined to preserve space and to make that relationship visible. For example:

```
origin/feature
[green][blue-]

instead of

feature origin/feature
[blue-] [green-------]
```

     Also note that while the transient features the `--show-signature` argument, that won't
actually be used when enabled, because Magit defaults to use just one line per commit.
Instead the commit colorized to indicate the validity of the signed commit object, using the
faces named `magit-signature-*` (which see).

     For a description of `magit-log-margin` see Section 5.3.3 [Log Margin], page 41.

5.3.3 Log Margin

In buffers which show one or more logs, it is possible to show additional information
about each commit in the margin.   The options used to configure the margin are
named `magit-INFIX-margin`, where INFIX is the same as in the respective major-mode
`magit-INFIX-mode`. In regular log buffers that would be `magit-log-margin`.

magit-log-margin   [User Option]
     This option specifies whether the margin is initially shown in Magit-Log mode buffers
     and how it is formatted.

     The value has the form `(INIT STYLE WIDTH AUTHOR AUTHOR-WIDTH)`.

       • If INIT is non-nil, then the margin is shown initially.

- STYLE controls how to format the committer date. It can be one of `age` (to show the age of the commit), `age-abbreviated` (to abbreviate the time unit to a character), or a string (suitable for `format-time-string`) to show the actual date.
- WIDTH controls the width of the margin. This exists for forward compatibility and currently the value should not be changed.
- AUTHOR controls whether the name of the author is also shown by default.
- AUTHOR-WIDTH has to be an integer. When the name of the author is shown, then this specifies how much space is used to do so.

You can change the STYLE and AUTHOR-WIDTH of all `magit-INFIX-margin` options to the same values by customizing `magit-log-margin` **before** `magit` is loaded. If you do that, then the respective values for the other options will default to what you have set for that variable. Likewise if you set INIT in `magit-log-margin` to `nil`, then that is used in the default of all other options. But setting it to `t`, i.e. re-enforcing the default for that option, does not carry to other options.

`L`   (`magit-margin-settings`)
          This transient prefix command binds the following suffix commands, each of which changes the appearance of the margin in some way.

In some buffers that support the margin, `L` is instead bound to `magit-log-refresh`, but that transient features the same commands, and then some other unrelated commands.

`L L`   (`magit-toggle-margin`)
          This command shows or hides the margin.

`L l`   (`magit-cycle-margin-style`)
          This command cycles the style used for the margin.

`L d`   (`magit-toggle-margin-details`)
          This command shows or hides details in the margin.

5.3.4 Select from Log

When the user has to select a recent commit that is reachable from `HEAD`, using regular completion would be inconvenient (because most humans cannot remember hashes or "HEAD~5", at least not without double checking). Instead a log buffer is used to select the commit, which has the advantage that commits are presented in order and with the commit message.

Such selection logs are used when selecting the beginning of a rebase and when selecting the commit to be squashed into.

In addition to the key bindings available in all log buffers, the following additional key bindings are available in selection log buffers:

`C-c C-c`   (`magit-log-select-pick`)
             Select the commit at point and act on it. Call `magit-log-select-pick-function` with the selected commit as argument.

`C-c C-k`   (`magit-log-select-quit`)
             Abort selecting a commit, don't act on any commit.

`magit-log-select-margin`   [User Option]

> This option specifies whether the margin is initially shown in Magit-Log-Select mode buffers and how it is formatted.
>
> The value has the form (`INIT STYLE WIDTH AUTHOR AUTHOR-WIDTH`).
>
> - If INIT is non-nil, then the margin is shown initially.
> - STYLE controls how to format the committer date. It can be one of `age` (to show the age of the commit), `age-abbreviated` (to abbreviate the time unit to a character), or a string (suitable for `format-time-string`) to show the actual date.
> - WIDTH controls the width of the margin. This exists for forward compatibility and currently the value should not be changed.
> - AUTHOR controls whether the name of the author is also shown by default.
> - AUTHOR-WIDTH has to be an integer. When the name of the author is shown, then this specifies how much space is used to do so.

5.3.5 Reflog

Also see the git-reflog(1) manpage.

These reflog commands are available from the log transient. See Section 5.3 [Logging], page 38.

`l r`   (`magit-reflog-current`)
>   Display the reflog of the current branch.

`l O`   (`magit-reflog-other`)
>   Display the reflog of a branch or another ref.

`l H`   (`magit-reflog-head`)
>   Display the `HEAD` reflog.

`magit-reflog-margin`   [User Option]

> This option specifies whether the margin is initially shown in Magit-Reflog mode buffers and how it is formatted.
>
> The value has the form (`INIT STYLE WIDTH AUTHOR AUTHOR-WIDTH`).
>
> - If INIT is non-nil, then the margin is shown initially.
> - STYLE controls how to format the committer date. It can be one of `age` (to show the age of the commit), `age-abbreviated` (to abbreviate the time unit to a character), or a string (suitable for `format-time-string`) to show the actual date.
> - WIDTH controls the width of the margin. This exists for forward compatibility and currently the value should not be changed.
> - AUTHOR controls whether the name of the author is also shown by default.
> - AUTHOR-WIDTH has to be an integer. When the name of the author is shown, then this specifies how much space is used to do so.

5.3.6 Cherries

Cherries are commits that haven't been applied upstream (yet), and are usually visualized using a log. Each commit is prefixed with - if it has an equivalent in the upstream and + if it does not, i.e. if it is a cherry.

The command `magit-cherry` shows cherries for a single branch, but the references buffer (see Section 5.6 [References Buffer], page 52) can show cherries for multiple "upstreams" at once.

Also see the git-reflog(1) manpage.

Y   (`magit-cherry`)

> Show commits that are in a certain branch but that have not been merged in the upstream branch.

`magit-cherry-margin`   [User Option]

> This option specifies whether the margin is initially shown in Magit-Cherry mode buffers and how it is formatted.
>
> The value has the form (`INIT STYLE WIDTH AUTHOR AUTHOR-WIDTH`).
>
> - If INIT is non-nil, then the margin is shown initially.
> - STYLE controls how to format the committer date. It can be one of `age` (to show the age of the commit), `age-abbreviated` (to abbreviate the time unit to a character), or a string (suitable for `format-time-string`) to show the actual date.
> - WIDTH controls the width of the margin. This exists for forward compatibility and currently the value should not be changed.
> - AUTHOR controls whether the name of the author is also shown by default.
> - AUTHOR-WIDTH has to be an integer. When the name of the author is shown, then this specifies how much space is used to do so.

5.4 Diffing

The status buffer contains diffs for the staged and unstaged commits, but that obviously isn't enough. The transient prefix command `magit-diff`, on d, features several suffix commands, which show a specific diff in a separate diff buffer.

Like other transient prefix commands, `magit-diff` also features several infix arguments that can be changed before invoking one of the suffix commands. However, in the case of the diff transient, these arguments may be taken from those currently in use in the current repository's diff buffer, depending on the value of `magit-prefix-use-buffer-arguments` (see Section 4.4 [Transient Arguments and Buffer Variables], page 20).

Also see the git-diff(1) manpage.

d   (`magit-diff`)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

d d   (`magit-diff-dwim`)

> Show changes for the thing at point.

d r   (magit-diff-range)

> Show differences between two commits.
>
> RANGE should be a range (A..B or A...B) but can also be a single commit. If one side of the range is omitted, then it defaults to `HEAD`. If just a commit is given, then changes in the working tree relative to that commit are shown.
>
> If the region is active, use the revisions on the first and last line of the region. With a prefix argument, instead of diffing the revisions, choose a revision to view changes along, starting at the common ancestor of both revisions (i.e., use a "..." range).

d w   (magit-diff-working-tree)

> Show changes between the current working tree and the `HEAD` commit. With a prefix argument show changes between the working tree and a commit read from the minibuffer.

d s   (magit-diff-staged)

> Show changes between the index and the `HEAD` commit. With a prefix argument show changes between the index and a commit read from the minibuffer.

d u   (magit-diff-unstaged)

> Show changes between the working tree and the index.

d p   (magit-diff-paths)

> Show changes between any two files on disk.

All of the above suffix commands update the repository's diff buffer. The diff transient also features two commands which show differences in another buffer:

d c   (magit-show-commit)

> Show the commit at point. If there is no commit at point or with a prefix argument, prompt for a commit.

d t   (magit-stash-show)

> Show all diffs of a stash in a buffer.

Two additional commands that show the diff for the file or blob that is being visited in the current buffer exists, see Section 8.8 [Minor Mode for Buffers Visiting Files], page 109.

5.4.1 Refreshing Diffs

The transient prefix command `magit-diff-refresh`, on D, can be used to change the diff arguments used in the current buffer, without changing which diff is shown. This works in dedicated diff buffers, but also in the status buffer.

D   (magit-diff-refresh)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

D g   (magit-diff-refresh)

> This suffix command sets the local diff arguments for the current buffer.

D s   (`magit-diff-set-default-arguments`)
>   This suffix command sets the default diff arguments for buffers of the same type
>   as that of the current buffer. Other existing buffers of the same type are not
>   affected because their local values have already been initialized.

D w   (`magit-diff-save-default-arguments`)
>   This suffix command sets the default diff arguments for buffers of the same
>   type as that of the current buffer, and saves the value for future sessions. Other
>   existing buffers of the same type are not affected because their local values have
>   already been initialized.

D t   (`magit-diff-toggle-refine-hunk`)
>   This command toggles hunk refinement on or off.

D r   (`magit-diff-switch-range-type`)
>   This command converts the diff range type from "revA..revB" to
>   "revB...revA", or vice versa.

D f   (`magit-diff-flip-revs`)
>   This command swaps revisions in the diff range from "revA..revB" to
>   "revB..revA", or vice versa.

D F   (`magit-diff-toggle-file-filter`)
>   This command toggles the file restriction of the diffs in the current buffer,
>   allowing you to quickly switch between viewing all the changes in the commit
>   and the restricted subset. As a special case, when this command is called from
>   a log buffer, it toggles the file restriction in the repository's revision buffer,
>   which is useful when you display a revision from a log buffer that is restricted
>   to a file or files.

In addition to the above transient, which allows changing any of the supported arguments, there also exist some commands that change only a particular argument.

-   (`magit-diff-less-context`)
>   This command decreases the context for diff hunks by COUNT lines.

+   (`magit-diff-more-context`)
>   This command increases the context for diff hunks by COUNT lines.

0   (`magit-diff-default-context`)
>   This command resets the context for diff hunks to the default height.

The following commands quickly change what diff is being displayed without having to using one of the diff transient.

C-c C-d   (`magit-diff-while-committing`)
>   While committing, this command shows the changes that are about to be committed. While amending, invoking the command again toggles between showing
>   just the new changes or all the changes that will be committed.
>
>   This binding is available in the diff buffer as well as the commit message buffer.

C-c C-b   (`magit-go-backward`)
>   This command moves backward in current buffer's history.

C-c C-f   (magit-go-forward)
> This command moves forward in current buffer's history.

5.4.2 Commands Available in Diffs

Some commands are only available if point is inside a diff.

`magit-diff-visit-file` and related commands visit the appropriate version of the file that the diff at point is about. Likewise `magit-diff-visit-worktree-file` and related commands visit the worktree version of the file that the diff at point is about. See Section 5.8.2 [Visiting Files and Blobs from a Diff], page 57, for more information and the key bindings.

C-c C-t   (magit-diff-trace-definition)
> This command shows a log for the definition at point.

`magit-log-trace-definition-function`   [User Option]
> The function specfied by this option is used by `magit-log-trace-definition` to determine the function at point. For major-modes that have special needs, you could set the local value using the mode's hook.

C-c C-e   (magit-diff-edit-hunk-commit)
> From a hunk, this command edits the respective commit and visits the file.
>
> First it visits the file being modified by the hunk at the correct location using `magit-diff-visit-file`. This actually visits a blob. When point is on a diff header, not within an individual hunk, then this visits the blob the first hunk is about.
>
> Then it invokes `magit-edit-line-commit`, which uses an interactive rebase to make the commit editable, or if that is not possible because the commit is not reachable from `HEAD` by checking out that commit directly. This also causes the actual worktree file to be visited.
>
> Neither the blob nor the file buffer are killed when finishing the rebase. If that is undesirable, then it might be better to use `magit-rebase-edit-command` instead of this command.

j   (magit-jump-to-diffstat-or-diff)
> This command jumps to the diffstat or diff. When point is on a file inside the diffstat section, then jump to the respective diff section. Otherwise, jump to the diffstat section or a child thereof.

The next two commands are not specific to Magit-Diff mode (or and Magit buffer for that matter), but it might be worth pointing out that they are available here too.

SPC   (scroll-up)
> This command scrolls text upward.

DEL   (scroll-down)
> This command scrolls text downward.

5.4.3 Diff Options

`magit-diff-refine-hunk`   [User Option]

Whether to show word-granularity differences within diff hunks.

- `nil` Never show fine differences.
- `t` Show fine differences for the current diff hunk only.
- `all` Show fine differences for all displayed diff hunks.

`magit-diff-refine-ignore-whitespace`   [User Option]

Whether to ignore whitespace changes in word-granularity differences.

`magit-diff-adjust-tab-width`   [User Option]

Whether to adjust the width of tabs in diffs.

Determining the correct width can be expensive if it requires opening large and/or many files, so the widths are cached in the variable `magit-diff--tab-width-cache`. Set that to nil to invalidate the cache.

- `nil` Never adjust tab width. Use 'tab-width's value from the Magit buffer itself instead.
- `t` If the corresponding file-visiting buffer exits, then use `tab-width`'s value from that buffer. Doing this is cheap, so this value is used even if a corresponding cache entry exists.
- `always` If there is no such buffer, then temporarily visit the file to determine the value.
- NUMBER Like `always`, but don't visit files larger than NUMBER bytes.

`magit-diff-paint-whitespace`   [User Option]

Specify where to highlight whitespace errors.

See `magit-diff-highlight-trailing`, `magit-diff-highlight-indentation`. The symbol `t` means in all diffs, `status` means only in the status buffer, and nil means nowhere.

- `nil` Never highlight whitespace errors.
- `t` Highlight whitespace errors everywhere.
- `uncommitted` Only highlight whitespace errors in diffs showing uncommitted changes. For backward compatibility `status` is treated as a synonym.

`magit-diff-paint-whitespace-lines`   [User Option]

Specify in what kind of lines to highlight whitespace errors.

- `t` Highlight only in added lines.
- `both` Highlight in added and removed lines.
- `all` Highlight in added, removed and context lines.

`magit-diff-highlight-trailing`   [User Option]

Whether to highlight whitespace at the end of a line in diffs. Used only when `magit-diff-paint-whitespace` is non-nil.

`magit-diff-highlight-indentation`   [User Option]

This option controls whether to highlight the indentaion in case it used the "wrong" indentation style. Indentation is only highlighted if `magit-diff-paint-whitespace` is also non-nil.

The value is an alist of the form `((REGEXP . INDENT)...)`. The path to the current repository is matched against each element in reverse order. Therefore if a REGEXP matches, then earlier elements are not tried.

If the used INDENT is `tabs`, highlight indentation with tabs. If INDENT is an integer, highlight indentation with at least that many spaces. Otherwise, highlight neither.

`magit-diff-hide-trailing-cr-characters`   [User Option]

Whether to hide ^M characters at the end of a line in diffs.

`magit-diff-highlight-hunk-region-functions`   [User Option]

This option specifies the functions used to highlight the hunk-internal region.

`magit-diff-highlight-hunk-region-dim-outside` overlays the outside of the hunk internal selection with a face that causes the added and removed lines to have the same background color as context lines. This function should not be removed from the value of this option.

`magit-diff-highlight-hunk-region-using-overlays` and `magit-diff-highlight-hunk-region-using-underline` emphasize the region by placing delimiting horizontal lines before and after it. Both of these functions have glitches which cannot be fixed due to limitations of Emacs' display engine. For more information see `https://github.com/magit/magit/issues/2758` ff.

Instead of, or in addition to, using delimiting horizontal lines, to emphasize the boundaries, you may which to emphasize the text itself, using `magit-diff-highlight-hunk-region-using-face`.

In terminal frames it's not possible to draw lines as the overlay and underline variants normally do, so there they fall back to calling the face function instead.

`magit-diff-unmarked-lines-keep-foreground`   [User Option]

This option controls whether added and removed lines outside the hunk-internal region only lose their distinct background color or also the foreground color. Whether the outside of the region is dimmed at all depends on `magit-diff-highlight-hunk-region-functions`.

5.4.4 Revision Buffer

`magit-revision-insert-related-refs`   [User Option]

Whether to show related branches in revision buffers.

- `nil` Don't show any related branches.
- `t` Show related local branches.
- `all` Show related local and remote branches.
- `mixed` Show all containing branches and local merged branches.

`magit-revision-show-gravatars`   [User Option]

> Whether to show gravatar images in revision buffers.
>
> If `nil`, then don't insert any gravatar images. If `t`, then insert both images. If `author` or `committer`, then insert only the respective image.
>
> If you have customized the option `magit-revision-headers-format` and want to insert the images then you might also have to specify where to do so. In that case the value has to be a cons-cell of two regular expressions. The car specifies where to insert the author's image. The top half of the image is inserted right after the matched text, the bottom half on the next line in the same column. The cdr specifies where to insert the committer's image, accordingly. Either the car or the cdr may be nil."

`magit-revision-use-hash-sections`   [User Option]

> Whether to turn hashes inside the commit message into sections.
>
> If non-nil, then hashes inside the commit message are turned into `commit` sections. There is a trade off to be made between performance and reliability:
>
> - `slow` calls git for every word to be absolutely sure.
> - `quick` skips words less than seven characters long.
> - `quicker` additionally skips words that don't contain a number.
> - `quickest` uses all words that are at least seven characters long and which contain at least one number as well as at least one letter.
>
> If nil, then no hashes are turned into sections, but you can still visit the commit at point using "RET".

The diffs shown in the revision buffer may be automatically restricted to a subset of the changed files. If the revision buffer is displayed from a log buffer, the revision buffer will share the same file restriction as that log buffer (also see the command `magit-diff-toggle-file-filter`).

`magit-revision-filter-files-on-follow`   [User Option]

> Whether showing a commit from a log buffer honors the log's file filter when the log arguments include `--follow`.
>
> When this option is nil, displaying a commit from a log ignores the log's file filter if the log arguments include `--follow`. Doing so avoids showing an empty diff in revision buffers for commits before a rename event. In such cases, the `--patch` argument of the log transient can be used to show the file-restricted diffs inline.
>
> Set this option to non-nil to keep the log's file restriction even if `--follow` is present in the log arguments.

If the revision buffer is not displayed from a log buffer, the file restriction is determined as usual (see Section 4.4 [Transient Arguments and Buffer Variables], page 20).

5.5 Ediffing

This section describes how to enter Ediff from Magit buffers. For information on how to
use Ediff itself, see `ediff`.

e   (`magit-ediff-dwim`)
>   Compare, stage, or resolve using Ediff.
>
>   This command tries to guess what file, and what commit or range the user
>   wants to compare, stage, or resolve using Ediff. It might only be able to guess
>   either the file, or range/commit, in which case the user is asked about the other.
>   It might not always guess right, in which case the appropriate `magit-ediff-*`
>   command has to be used explicitly. If it cannot read the user's mind at all,
>   then it asks the user for a command to run.

E   (`magit-ediff`)
>   This transient prefix command binds the following suffix commands and dis-
>   plays them in a temporary buffer until a suffix is invoked.

E r   (`magit-ediff-compare`)
>   Compare two revisions of a file using Ediff.
>
>   If the region is active, use the revisions on the first and last line of the region.
>   With a prefix argument, instead of diffing the revisions, choose a revision to
>   view changes along, starting at the common ancestor of both revisions (i.e., use
>   a `"..."` range).

E m   (`magit-ediff-resolve`)
>   Resolve outstanding conflicts in a file using Ediff, defaulting to the file at point.
>
>   Provided that the value of `merge.conflictstyle` is `diff3`, you can view the
>   file's merge-base revision using `/` in the Ediff control buffer.
>
>   In the rare event that you want to manually resolve all conflicts, including those
>   already resolved by Git, use `ediff-merge-revisions-with-ancestor`.

E s   (`magit-ediff-stage`)
>   Stage and unstage changes to a file using Ediff, defaulting to the file at point.

E u   (`magit-ediff-show-unstaged`)
>   Show unstaged changes to a file using Ediff.

E i   (`magit-ediff-show-staged`)
>   Show staged changes to a file using Ediff.

E w   (`magit-ediff-show-working-tree`)
>   Show changes in a file between `HEAD` and working tree using Ediff.

E c   (`magit-ediff-show-commit`)
>   Show changes to a file introduced by a commit using Ediff.

E z   (`magit-ediff-show-stash`)
>   Show changes to a file introduced by a stash using Ediff.

`magit-ediff-dwim-show-on-hunks`   [User Option]
>   This option controls what command `magit-ediff-dwim` calls when point is on
>   uncommitted hunks. When nil, always run `magit-ediff-stage`. Otherwise, use

> `magit-ediff-show-staged` and `magit-ediff-show-unstaged` to show staged and unstaged changes, respectively.

`magit-ediff-show-stash-with-index`   [User Option]
> This option controls whether `magit-ediff-show-stash` includes a buffer containing the file's state in the index at the time the stash was created. This makes it possible to tell which changes in the stash were staged.

`magit-ediff-quit-hook`   [User Option]
> This hook is run after quitting an Ediff session that was created using a Magit command. The hook functions are run inside the Ediff control buffer, and should not change the current buffer.
>
> This is similar to `ediff-quit-hook` but takes the needs of Magit into account. The regular `ediff-quit-hook` is ignored by Ediff sessions that were created using a Magit command.

5.6 References Buffer

y   (`magit-show-refs`)
> This command lists branches and tags in a dedicated buffer.
>
> However if this command is invoked again from this buffer or if it is invoked with a prefix argument, then it acts as a transient prefix command, which binds the following suffix commands and some infix arguments.

All of the following suffix commands list exactly the same branches and tags. The only difference the optional feature that can be enabled by changing the value of `magit-refs-show-commit-count` (see below). These commands specify a different branch or commit against which all the other references are compared.

y y   (`magit-show-refs-head`)
> This command lists branches and tags in a dedicated buffer. Each reference is being compared with `HEAD`.

y c   (`magit-show-refs-current`)
> This command lists branches and tags in a dedicated buffer. Each reference is being compared with the current branch or `HEAD` if it is detached.

y o   (`magit-show-refs-other`)
> This command lists branches and tags in a dedicated buffer. Each reference is being compared with a branch read from the user.

`magit-refs-show-commit-count`   [User Option]
> Whether to show commit counts in Magit-Refs mode buffers.
>
> - `all` Show counts for branches and tags.
> - `branch` Show counts for branches only.
> - `nil` Never show counts.
>
> The default is `nil` because anything else can be very expensive.

`magit-refs-pad-commit-counts`   [User Option]

Whether to pad all commit counts on all sides in Magit-Refs mode buffers.

If this is nil, then some commit counts are displayed right next to one of the branches that appear next to the count, without any space in between. This might look bad if the branch name faces look too similar to `magit-dimmed`.

If this is non-nil, then spaces are placed on both sides of all commit counts.

`magit-refs-show-remote-prefix`   [User Option]

Whether to show the remote prefix in lists of remote branches.

Showing the prefix is redundant because the name of the remote is already shown in the heading preceding the list of its branches.

`magit-refs-primary-column-width`   [User Option]

Width of the primary column in 'magit-refs-mode' buffers. The primary column is the column that contains the name of the branch that the current row is about.

If this is an integer, then the column is that many columns wide. Otherwise it has to be a cons-cell of two integers. The first specifies the minimal width, the second the maximal width. In that case the actual width is determined using the length of the names of the shown local branches. (Remote branches and tags are not taken into account when calculating to optimal width.)

`magit-refs-focus-column-width`   [User Option]

Width of the focus column in 'magit-refs-mode' buffers.

The focus column is the first column, which marks one branch (usually the current branch) as the focused branch using `*` or `@`. For each other reference, this column optionally shows how many commits it is ahead of the focused branch and `<`, or if it isn't ahead then the commits it is behind and `>`, or if it isn't behind either, then a `=`.

This column may also display only `*` or `@` for the focused branch, in which case this option is ignored. Use `L v` to change the verbosity of this column.

`magit-refs-margin`   [User Option]

This option specifies whether the margin is initially shown in Magit-Refs mode buffers and how it is formatted.

The value has the form (`INIT STYLE WIDTH AUTHOR AUTHOR-WIDTH`).

- If INIT is non-nil, then the margin is shown initially.
- STYLE controls how to format the committer date. It can be one of `age` (to show the age of the commit), `age-abbreviated` (to abbreviate the time unit to a character), or a string (suitable for `format-time-string`) to show the actual date.
- WIDTH controls the width of the margin. This exists for forward compatibility and currently the value should not be changed.
- AUTHOR controls whether the name of the author is also shown by default.
- AUTHOR-WIDTH has to be an integer. When the name of the author is shown, then this specifies how much space is used to do so.

`magit-refs-margin-for-tags`   [User Option]

   This option specifies whether to show information about tags in the margin. This is
   disabled by default because it is slow if there are many tags.

   The following variables control how individual refs are displayed. If you change one of
these variables (especially the "%c" part), then you should also change the others to keep
things aligned. The following %-sequences are supported:

- `%a` Number of commits this ref has over the one we compare to.
- `%b` Number of commits the ref we compare to has over this one.
- `%c` Number of commits this ref has over the one we compare to. For the ref which
  all other refs are compared this is instead "@", if it is the current branch, or "#"
  otherwise.
- `%C` For the ref which all other refs are compared this is "@", if it is the current branch,
  or "#" otherwise. For all other refs " ".
- `%h` Hash of this ref's tip.
- `%m` Commit summary of the tip of this ref.
- `%n` Name of this ref.
- `%u` Upstream of this local branch.
- `%U` Upstream of this local branch and additional local vs. upstream information.

`magit-refs-filter-alist`   [User Option]

   The purpose of this option is to forgo displaying certain refs based on their name.
   If you want to not display any refs of a certain type, then you should remove the
   appropriate function from `magit-refs-sections-hook` instead.

   This alist controls which tags and branches are omitted from being displayed
   in `magit-refs-mode` buffers. If it is `nil`, then all refs are displayed (subject to
   `magit-refs-sections-hook`).

   All keys are tried in order until one matches. Then its value is used and subsequent
   elements are ignored. If the value is non-nil, then the reference is displayed, otherwise
   it is not. If no element matches, then the reference is displayed.

   A key can either be a regular expression that the refname has to match, or a function
   that takes the refname as only argument and returns a boolean. A remote branch
   such as "origin/master" is displayed as just "master", however for this comparison
   the former is used.

RET   (`magit-visit-ref`)

            This command visits the reference or revision at point in another buffer. If
            there is no revision at point or with a prefix argument then it prompts for a
            revision.

            This command behaves just like `magit-show-commit` as described above, ex-
            cept if point is on a reference in a `magit-refs-mode` buffer, in which case
            the behavior may be different, but only if you have customized the option
            `magit-visit-ref-behavior`.

`magit-visit-ref-behavior`   [User Option]

> This option controls how `magit-visit-ref` behaves in `magit-refs-mode` buffers.
>
> By default `magit-visit-ref` behaves like `magit-show-commit`, in all buffers, including `magit-refs-mode` buffers. When the type of the section at point is `commit` then "RET" is bound to `magit-show-commit`, and when the type is either `branch` or `tag` then it is bound to `magit-visit-ref`.
>
> "RET" is one of Magit's most essential keys and at least by default it should behave consistently across all of Magit, especially because users quickly learn that it does something very harmless; it shows more information about the thing at point in another buffer.
>
> However "RET" used to behave differently in `magit-refs-mode` buffers, doing surprising things, some of which cannot really be described as "visit this thing". If you've grown accustomed this behavior, you can restore it by adding one or more of the below symbols to the value of this option. But keep in mind that by doing so you don't only introduce inconsistencies, you also lose some functionality and might have to resort to `M-x magit-show-commit` to get it back.
>
> `magit-visit-ref` looks for these symbols in the order in which they are described here. If the presence of a symbol applies to the current situation, then the symbols that follow do not affect the outcome.
>
> - `focus-on-ref`
>
>   With a prefix argument update the buffer to show commit counts and lists of cherry commits relative to the reference at point instead of relative to the current buffer or `HEAD`.
>
>   Instead of adding this symbol, consider pressing "C-u y o RET".
>
> - `create-branch`
>
>   If point is on a remote branch, then create a new local branch with the same name, use the remote branch as its upstream, and then check out the local branch.
>
>   Instead of adding this symbol, consider pressing "b c RET RET", like you would do in other buffers.
>
> - `checkout-any`
>
>   Check out the reference at point. If that reference is a tag or a remote branch, then this results in a detached `HEAD`.
>
>   Instead of adding this symbol, consider pressing "b b RET", like you would do in other buffers.
>
> - `checkout-branch`
>
>   Check out the local branch at point.
>
>   Instead of adding this symbol, consider pressing "b b RET", like you would do in other buffers.

5.6.1 References Sections

The contents of references buffers is controlled using the hook `magit-refs-sections-hook`. See Section 4.2.3 [Section Hooks], page 18, to learn about such hooks and how to customize them. All of the below functions are members of the default value. Note that it makes

much less sense to customize this hook than it does for the respective hook used for the status buffer.

magit-refs-sections-hook   [User Option]
        Hook run to insert sections into a references buffer.

magit-insert-local-branches   [Function]
        Insert sections showing all local branches.

magit-insert-remote-branches   [Function]
        Insert sections showing all remote-tracking branches.

magit-insert-tags   [Function]
        Insert sections showing all tags.

5.7 Bisecting

Also see the git-bisect(1) manpage.

B   (`magit-bisect`)
            This transient prefix command binds the following suffix commands and displays them in a temporary buffer until a suffix is invoked.

    When bisecting is not in progress, then the transient features the following suffix commands.

B B   (`magit-bisect-start`)
            Start a bisect session.

            Bisecting a bug means to find the commit that introduced it. This command starts such a bisect session by asking for a known good and a bad commit.

B s   (`magit-bisect-run`)
            Bisect automatically by running commands after each step.

    When bisecting in progress, then the transient instead features the following suffix commands.

B b   (`magit-bisect-bad`)
            Mark the current commit as bad. Use this after you have asserted that the commit does contain the bug in question.

B g   (`magit-bisect-good`)
            Mark the current commit as good. Use this after you have asserted that the commit does not contain the bug in question.

B k   (`magit-bisect-skip`)
            Skip the current commit. Use this if for some reason the current commit is not a good one to test. This command lets Git choose a different one.

B r   (`magit-bisect-reset`)
            After bisecting, cleanup bisection state and return to original `HEAD`.

    By default the status buffer shows information about the ongoing bisect session.

magit-bisect-show-graph   [User Option]
        This option controls whether a graph is displayed for the log of commits that still have to be bisected.

5.8 Visiting Files and Blobs

Magit provides several commands that visit a file or blob (the version of a file that is stored in a certain commit). Actually it provides several **groups** of such commands and the several **variants** within each group.

5.8.1 General-Purpose Visit Commands

These commands can be used anywhere to open any blob. Currently no keys are bound to these commands by default, but that is likely to change.

`magit-find-file`   [Command]

> This command reads a filename and revision from the user and visits the respective blob in a buffer. The buffer is displayed in the selected window.

`magit-find-file-other-window`   [Command]

> This command reads a filename and revision from the user and visits the respective blob in a buffer. The buffer is displayed in another window.

`magit-find-file-other-frame`   [Command]

> This command reads a filename and revision from the user and visits the respective blob in a buffer. The buffer is displayed in another frame.

5.8.2 Visiting Files and Blobs from a Diff

These commands can only be used when point is inside a diff.

RET   (`magit-diff-visit-file`)

> This command visits the appropriate version of the file that the diff at point is about.
>
> This commands visits the worktree version of the appropriate file. The location of point inside the diff determines which file is being visited. The visited version depends on what changes the diff is about.
>
> - If the diff shows uncommitted changes (i.e. staged or unstaged changes), then visit the file in the working tree (i.e. the same "real" file that `find-file` would visit. In all other cases visit a "blob" (i.e. the version of a file as stored in some commit).
> - If point is on a removed line, then visit the blob for the first parent of the commit that removed that line, i.e. the last commit where that line still exists.
> - If point is on an added or context line, then visit the blob that adds that line, or if the diff shows from more than a single commit, then visit the blob from the last of these commits.
>
> In the file-visiting buffer this command goes to the line that corresponds to the line that point is on in the diff.
>
> The buffer is displayed in the selected window. With a prefix argument the buffer is displayed in another window instead.

`magit-diff-visit-previous-blob`   [User Option]

> This option controls whether `magit-diff-visit-file` may visit the previous blob.
> When this is `t` (the default) and point is on a removed line in a diff for a committed
> change, then `magit-diff-visit-file` visits the blob from the last revision which
> still had that line.
>
> Currently this is only supported for committed changes, for staged and unstaged
> changes `magit-diff-visit-file` always visits the file in the working tree.

`C-<return>`   (`magit-diff-visit-file-worktree`)

> > This command visits the worktree version of the appropriate file. The loca-
> > tion of point inside the diff determines which file is being visited. Unlike
> > `magit-diff-visit-file` it always visits the "real" file in the working tree,
> > i.e the "current version" of the file.
> >
> > In the file-visiting buffer this command goes to the line that corresponds to the
> > line that point is on in the diff. Lines that were added or removed in the working
> > tree, the index and other commits in between are automatically accounted for.
> >
> > The buffer is displayed in the selected window. With a prefix argument the
> > buffer is displayed in another window instead.

Variants of the above two commands exist that instead visit the file in another window
or in another frame. If you prefer such behavior, then you may want to change the above
key bindings, but note that the above commands also use another window when invoked
with a prefix argument.

`magit-diff-visit-file-other-window`   [Command]

`magit-diff-visit-file-other-frame`   [Command]

`magit-diff-visit-worktree-file-other-window`   [Command]

`magit-diff-visit-worktree-file-other-frame`   [Command]

5.9 Blaming

Also see the git-blame(1) manpage.

To start blaming invoke the `magit-file-dispatch` transient prefix command by press-
ing `C-c M-g`. (This is only the default binding and the recommended binding is `C-c g`.
Also neither binding may be available if you disabled `global-magit-file-mode`. Also see
Section 8.8 [Minor Mode for Buffers Visiting Files], page 109.)

The blaming suffix commands can be invoked from the dispatch transient. However if
you want to set an infix argument, then you have to enter the blaming sub-transient first.

The key bindings shown below assume that you enter the dispatch transient using the
default binding.

`C-c M-g B`   (`magit-blame`)

> > This transient prefix command binds the following suffix commands along with
> > the appropriate infix arguments and displays them in a temporary buffer until
> > a suffix is invoked.

Note that not all of the following suffixes are available at all times. For example if `magit-blame-mode` is not enabled, then the command whose purpose is to turn off that mode would not be of any use and therefore isn't available.

C-c M-g b   (`magit-blame-addition`)
C-c M-g B b   (`magit-blame-addition`)
> This command augments each line or chunk of lines in the current file-visiting or blob-visiting buffer with information about what commits last touched these lines.
>
> If the buffer visits a revision of that file, then history up to that revision is considered. Otherwise, the file's full history is considered, including uncommitted changes.
>
> If Magit-Blame mode is already turned on in the current buffer then blaming is done recursively, by visiting REVISION:FILE (using `magit-find-file`), where REVISION is a parent of the revision that added the current line or chunk of lines.

C-c M-g r   (`magit-blame-removal`)
C-c M-g B r   (`magit-blame-removal`)
> This command augments each line or chunk of lines in the current blob-visiting buffer with information about the revision that removes it. It cannot be used in file-visiting buffers.
>
> Like `magit-blame-addition`, this command can be used recursively.

C-c M-g f   (`magit-blame-reverse`)
C-c M-g B f   (`magit-blame-reverse`)
> This command augments each line or chunk of lines in the current file-visiting or blob-visiting buffer with information about the last revision in which a line still existed.
>
> Like `magit-blame-addition`, this command can be used recursively.

C-c M-g e   (`magit-blame-echo`)
C-c M-g B e   (`magit-blame-echo`)
> This command is like `magit-blame-addition` except that it doesn't turn on `read-only-mode` and that it initially uses the visualization style specified by option `magit-blame-echo-style`.

The following key bindings are available when Magit-Blame mode is enabled and Read-Only mode is not enabled. These commands are also available in other buffers; here only the behavior is described that is relevant in file-visiting buffers that are being blamed.

RET   (`magit-show-commit`)
> This command shows the commit that last touched the line at point.

SPC   (`magit-diff-show-or-scroll-up`)
> This command updates the commit buffer.
>
> This either shows the commit that last touched the line at point in the appropriate buffer, or if that buffer is already being displayed in the current frame and if that buffer contains information about that commit, then the buffer is scrolled up instead.

DEL   (`magit-diff-show-or-scroll-down`)
> This command updates the commit buffer.
>
> This either shows the commit that last touched the line at point in the appropriate buffer, or if that buffer is already being displayed in the current frame and if that buffer contains information about that commit, then the buffer is scrolled down instead.

The following key bindings are available when both Magit-Blame mode and Read-Only mode are enabled.

b   (`magit-blame`)
> See above.

n   (`magit-blame-next-chunk`)
> This command moves to the next chunk.

N   (`magit-blame-next-chunk-same-commit`)
> This command moves to the next chunk from the same commit.

p   (`magit-blame-previous-chunk`)
> This command moves to the previous chunk.

P   (`magit-blame-previous-chunk-same-commit`)
> This command moves to the previous chunk from the same commit.

q   (`magit-blame-quit`)
> This command turns off Magit-Blame mode. If the buffer was created during a recursive blame, then it also kills the buffer.

M-w   (`magit-blame-copy-hash`)
> This command saves the hash of the current chunk's commit to the kill ring.
>
> When the region is active, the command saves the region's content instead of the hash, like `kill-ring-save` would.

c   (`magit-blame-cycle-style`)
> This command changes how blame information is visualized in the current buffer by cycling through the styles specified using the option `magit-blame-styles`.

Blaming is also controlled using the following options.

`magit-blame-styles`   [User Option]
> This option defines a list of styles used to visualize blame information. For now see its doc-string to learn more.

`magit-blame-echo-style`   [User Option]
> This option specifies the blame visualization style used by the command `magit-blame-echo`. This must be a symbol that is used as the identifier for one of the styles defined in `magit-blame-styles`.

`magit-blame-time-format`   [User Option]
> This option specifies the format string used to display times when showing blame information.

`magit-blame-read-only`   [User Option]
    This option controls whether blaming a buffer also makes temporarily read-only.

`magit-blame-disable-modes`   [User Option]
    This option lists incompatible minor-modes that should be disabled temporarily when
    a buffer contains blame information. They are enabled again when the buffer no longer
    shows blame information.

`magit-blame-goto-chunk-hook`   [User Option]
    This hook is run when moving between chunks.

6 Manipulating

6.1 Creating Repository

`M-x magit-init`   (magit-init)

> This command initializes a repository and then shows the status buffer for the new repository.
>
> If the directory is below an existing repository, then the user has to confirm that a new one should be created inside. If the directory is the root of the existing repository, then the user has to confirm that it should be reinitialized.

6.2 Cloning Repository

To clone a remote or local repository use `C`, which is bound to the command `magit-clone`. This command either act as a transient prefix command, which binds several infix arguments and suffix commands, or it can invoke `git clone` directly, depending on whether a prefix argument is used and on the value of `magit-clone-always-transient`.

`magit-clone-always-transient`   [User Option]

> This option controls whether the command `magit-clone` always acts as a transient prefix command, regardless of whether a prefix argument is used or not. If `t`, then that command always acts as a transient prefix. If `nil`, then a prefix argument has to be used for it to act as a transient.

`C`   (magit-clone)

> This command either acts as a transient prefix command as described above or does the same thing as `transient-clone-regular` as described below.
>
> If it acts as a transient prefix, then it binds the following suffix commands and several infix arguments.

`C C`   (magit-clone-regular)

> This command creates a regular clone of an existing repository. The repository and the target directory are read from the user.

`C s`   (magit-clone-shallow)

> This command creates a shallow clone of an existing repository. The repository and the target directory are read from the user. By default the depth of the cloned history is a single commit, but with a prefix argument the depth is read from the user.

`C b`   (magit-clone-bare)

> This command creates a bare clone of an existing repository. The repository and the target directory are read from the user.

`C m`   (magit-clone-mirror)

> This command creates a mirror of an existing repository. The repository and the target directory are read from the user.

The following suffixes are disabled by default. See Section "Enabling and Disabling Suffixes" in `transient` for how to enable them.

`C d` (`magit-clone-shallow-since`)

> This command creates a shallow clone of an existing repository. Only commits that were committed after a date are cloned, which is read from the user. The repository and the target directory are also read from the user.

`C e` (`magit-clone-shallow-exclude`)

> This command creates a shallow clone of an existing repository. This reads a branch or tag from the user. Commits that are reachable from that are not cloned. The repository and the target directory are also read from the user.

`magit-clone-set-remote-head`   [User Option]

> This option controls whether cloning causes the reference `refs/remotes/<remote>/HEAD` to be created in the clone. The default is to do so.
>
> Actually `git clone` itself does that and cannot be told to not do it. Therefore setting this to `nil` causes Magit to remove that reference after cloning.

`magit-clone-set-remote.pushDefault`   [User Option]

> This option controls whether the value of the Git variable `remote.pushDefault` is set after cloning.
>
> • If `t`, then it is always set without asking.
> • If `ask`, then the users are asked every time they clone a repository.
> • If `nil`, then it is never set.

`magit-clone-default-directory`   [User Option]

> This option control the default directory name used when reading the destination for a cloning operation.
>
> • If `nil` (the default), then the value of `default-directory` is used.
> • If a directory, then that is used.
> • If a function, then that is called with the remote url as the only argument and the returned value is used.

`magit-clone-name-alist`   [User Option]

> This option maps regular expressions, which match repository names, to repository urls, making it possible for users to enter short names instead of urls when cloning repositories.
>
> Each element has the form (REGEXP HOSTNAME USER). When the user enters a name when a cloning command asks for a name or url, then that is looked up in this list. The first element whose REGEXP matches is used.
>
> The format specified by option `magit-clone-url-format` is used to turn the name into an url, using HOSTNAME and the repository name. If the provided name contains a slash, then that is used. Otherwise if the name omits the owner of the repository, then the default user specified in the matched entry is used.
>
> If USER contains a dot, then it is treated as a Git variable and the value of that is used as the username. Otherwise it is used as the username itself.

`magit-clone-url-format`   [User Option]
> The format specified by this option is used when turning repository names into urls.
> `%h` is the hostname and `%n` is the repository name, including the name of the owner.

6.3 Staging and Unstaging

Like Git, Magit can of course stage and unstage complete files. Unlike Git, it also allows users to gracefully un-/stage individual hunks and even just part of a hunk. To stage individual hunks and parts of hunks using Git directly, one has to use the very modal and rather clumsy interface of a `git add --interactive` session.

With Magit, on the other hand, one can un-/stage individual hunks by just moving point into the respective section inside a diff displayed in the status buffer or a separate diff buffer and typing `s` or `u`. To operate on just parts of a hunk, mark the changes that should be un-/staged using the region and then press the same key that would be used to un-/stage. To stage multiple files or hunks at once use a region that starts inside the heading of such a section and ends inside the heading of a sibling section of the same type.

Besides staging and unstaging, Magit also provides several other "apply variants" that can also operate on a file, multiple files at once, a hunk, multiple hunks at once, and on parts of a hunk. These apply variants are described in the next section.

You can also use Ediff to stage and unstage. See Section 5.5 [Ediffing], page 51.

s   (`magit-stage`)
> Add the change at point to the staging area.
>
> With a prefix argument and an untracked file (or files) at point, stage the file but not its content. This makes it possible to stage only a subset of the new file's changes.

S   (`magit-stage-modified`)
> Stage all changes to files modified in the worktree. Stage all new content of tracked files and remove tracked files that no longer exist in the working tree from the index also. With a prefix argument also stage previously untracked (but not ignored) files.

u   (`magit-unstage`)
> Remove the change at point from the staging area.
>
> Only staged changes can be unstaged. But by default this command performs an action that is somewhat similar to unstaging, when it is called on a committed change: it reverses the change in the index but not in the working tree.

U   (`magit-unstage-all`)
> Remove all changes from the staging area.

`magit-unstage-committed`   [User Option]
> This option controls whether `magit-unstage` "unstages" committed changes by reversing them in the index but not the working tree. The alternative is to raise an error.

M-x magit-reverse-in-index   (magit-reverse-in-index)

> This command reverses the committed change at point in the index but not the
> working tree. By default no key is bound directly to this command, but it is
> indirectly called when `u` (`magit-unstage`) is pressed on a committed change.
>
> This allows extracting a change from `HEAD`, while leaving it in the working tree,
> so that it can later be committed using a separate commit. A typical workflow
> would be:
>
> * Optionally make sure that there are no uncommitted changes.
> * Visit the `HEAD` commit and navigate to the change that should not have
>   been included in that commit.
> * Type `u` (`magit-unstage`) to reverse it in the index. This assumes that
>   `magit-unstage-committed-changes` is non-nil.
> * Type `c e` to extend `HEAD` with the staged changes, including those that
>   were already staged before.
> * Optionally stage the remaining changes using `s` or `S` and then type `c c` to
>   create a new commit.

M-x magit-reset-index   (magit-reset-index)

> Reset the index to some commit. The commit is read from the user and defaults
> to the commit at point. If there is no commit at point, then it defaults to `HEAD`.

6.3.1 Staging from File-Visiting Buffers

Fine-grained un-/staging has to be done from the status or a diff buffer, but it's also possible
to un-/stage all changes made to the file visited in the current buffer right from inside that
buffer.

M-x magit-stage-file   (magit-stage-file)

> When invoked inside a file-visiting buffer, then stage all changes to that file.
> In a Magit buffer, stage the file at point if any. Otherwise prompt for a file to
> be staged. With a prefix argument always prompt the user for a file, even in a
> file-visiting buffer or when there is a file section at point.

M-x magit-unstage-file   (magit-unstage-file)

> When invoked inside a file-visiting buffer, then unstage all changes to that file.
> In a Magit buffer, unstage the file at point if any. Otherwise prompt for a file
> to be unstaged. With a prefix argument always prompt the user for a file, even
> in a file-visiting buffer or when there is a file section at point.

6.4 Applying

Magit provides several "apply variants": stage, unstage, discard, reverse, and "regular
apply". At least when operating on a hunk they are all implemented using `git apply`,
which is why they are called "apply variants".

* Stage. Apply a change from the working tree to the index. The change also remains
  in the working tree.
* Unstage. Remove a change from the index. The change remains in the working tree.

- Discard. On a staged change, remove it from the working tree and the index. On an unstaged change, remove it from the working tree only.
- Reverse. Reverse a change in the working tree. Both committed and staged changes can be reversed. Unstaged changes cannot be reversed. Discard them instead.
- Apply. Apply a change to the working tree. Both committed and staged changes can be applied. Unstaged changes cannot be applied - as they already have been applied.

The previous section described the staging and unstaging commands. What follows are the commands which implement the remaining apply variants.

`a` (`magit-apply`)

> Apply the change at point to the working tree.
>
> With a prefix argument fallback to a 3-way merge. Doing so causes the change to be applied to the index as well.

`k` (`magit-discard`)

> Remove the change at point from the working tree.

`v` (`magit-reverse`)

> Reverse the change at point in the working tree.
>
> With a prefix argument fallback to a 3-way merge. Doing so causes the change to be applied to the index as well.

With a prefix argument all apply variants attempt a 3-way merge when appropriate (i.e. when `git apply` is used internally).

6.5 Committing

When the user initiates a commit, Magit calls `git commit` without any arguments, so Git has to get it from the user. It creates the file `.git/COMMIT_EDITMSG` and then opens that file in an editor. Magit arranges for that editor to be the Emacsclient. Once the user finishes the editing session, the Emacsclient exits and Git creates the commit using the file's content as message.

6.5.1 Initiating a Commit

Also see the git-commit(1) manpage.

`c` (`magit-commit`)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

`c c` (`magit-commit-create`)

> Create a new commit on `HEAD`. With a prefix argument amend to the commit at `HEAD` instead.

`c a` (`magit-commit-amend`)

> Amend the last commit.

`c e` (`magit-commit-extend`)

> Amend the last commit, without editing the message. With a prefix argument keep the committer date, otherwise change it. The option `magit-commit-`

> `extend-override-date` can be used to inverse the meaning of the prefix argument.
>
> Non-interactively respect the optional OVERRIDE-DATE argument and ignore the option.

`c w`   (`magit-commit-reword`)

> Reword the last commit, ignoring staged changes. With a prefix argument keep the committer date, otherwise change it. The option `magit-commit-reword-override-date` can be used to inverse the meaning of the prefix argument.
>
> Non-interactively respect the optional OVERRIDE-DATE argument and ignore the option.

`c f`   (`magit-commit-fixup`)

> Create a fixup commit.
>
> With a prefix argument the target commit has to be confirmed. Otherwise the commit at point may be used without confirmation depending on the value of option `magit-commit-squash-confirm`.

`c F`   (`magit-commit-instant-fixup`)

> Create a fixup commit and instantly rebase.

`c s`   (`magit-commit-squash`)

> Create a squash commit, without editing the squash message.
>
> With a prefix argument the target commit has to be confirmed. Otherwise the commit at point may be used without confirmation depending on the value of option `magit-commit-squash-confirm`.

`c S`   (`magit-commit-instant-squash`)

> Create a squash commit and instantly rebase.

`c A`   (`magit-commit-augment`)

> Create a squash commit, editing the squash message.
>
> With a prefix argument the target commit has to be confirmed. Otherwise the commit at point may be used without confirmation depending on the value of option `magit-commit-squash-confirm`.

`magit-commit-ask-to-stage`   [User Option]

> Whether to ask to stage all unstaged changes when committing and nothing is staged.

`magit-commit-extend-override-date`   [User Option]

> Whether using `magit-commit-extend` changes the committer date.

`magit-commit-reword-override-date`   [User Option]

> Whether using `magit-commit-reword` changes the committer date.

`magit-commit-squash-confirm`   [User Option]

> Whether the commit targeted by squash and fixup has to be confirmed. When non-nil then the commit at point (if any) is used as default choice. Otherwise it has to be confirmed. This option only affects `magit-commit-squash` and `magit-commit-fixup`. The "instant" variants always require confirmation because making an error while using those is harder to recover from.

6.5.2 Editing Commit Messages

After initiating a commit as described in the previous section, two new buffers appear. One shows the changes that are about to committed, while the other is used to write the message. All regular editing commands are available in the commit message buffer. This section only describes the additional commands.

Commit messages are edited in an edit session - in the background Git is waiting for the editor, in our case the Emacsclient, to save the commit message in a file (in most cases `.git/COMMIT_EDITMSG`) and then return. If the Emacsclient returns with a non-zero exit status then Git does not create the commit. So the most important commands are those for finishing and aborting the commit.

C-c C-c   (with-editor-finish)
> Finish the current editing session by returning with exit code 0. Git then creates the commit using the message it finds in the file.

C-c C-k   (with-editor-cancel)
> Cancel the current editing session by returning with exit code 1. Git then cancels the commit, but leaves the file untouched.

In addition to being used by Git, these messages may also be stored in a ring that persists until Emacs is closed. By default the message is stored at the beginning and the end of an edit session (regardless of whether the session is finished successfully or was canceled). It is sometimes useful to bring back messages from that ring.

C-c M-s   (git-commit-save-message)
> Save the current buffer content to the commit message ring.

M-p   (git-commit-prev-message)
> Cycle backward through the commit message ring, after saving the current message to the ring. With a numeric prefix ARG, go back ARG comments.

M-n   (git-commit-next-message)
> Cycle forward through the commit message ring, after saving the current message to the ring. With a numeric prefix ARG, go back ARG comments.

By default the diff for the changes that are about to be committed are automatically shown when invoking the commit. When amending to an existing commit it may be useful to show either the changes that are about to be added to that commit or to show those changes together with those that are already committed.

C-c C-d   (magit-diff-while-committing)
> While committing, show the changes that are about to be committed. While amending, invoking the command again toggles between showing just the new changes or all the changes that will be committed.

C-c C-w   (magit-pop-revision-stack)
> This command inserts a representation of a revision into the current buffer. It can be used inside buffers used to write commit messages but also in other buffers such as buffers used to edit emails or ChangeLog files.
>
> By default this command pops the revision which was last added to the `magit-revision-stack` and inserts it into the current buffer according to

`magit-pop-revision-stack-format`. Revisions can be put on the stack using `magit-copy-section-value` and `magit-copy-buffer-revision`.

If the stack is empty or with a prefix argument it instead reads a revision in the minibuffer. By using the minibuffer history this allows selecting an item which was popped earlier or to insert an arbitrary reference or revision without first pushing it onto the stack.

When reading the revision from the minibuffer, then it might not be possible to guess the correct repository. When this command is called inside a repository (e.g. while composing a commit message), then that repository is used. Otherwise (e.g. while composing an email) then the repository recorded for the top element of the stack is used (even though we insert another revision). If not called inside a repository and with an empty stack, or with two prefix arguments, then read the repository in the minibuffer too.

`magit-pop-revision-stack-format`   [User Option]

This option controls how the command `magit-pop-revision-stack` inserts a revision into the current buffer.

The entries on the stack have the format (`HASH TOPLEVEL`) and this option has the format (`POINT-FORMAT EOB-FORMAT INDEX-REGEXP`), all of which may be nil or a string (though either one of EOB-FORMAT or POINT-FORMAT should be a string, and if INDEX-REGEXP is non-nil, then the two formats should be too).

First INDEX-REGEXP is used to find the previously inserted entry, by searching backward from point. The first submatch must match the index number. That number is incremented by one, and becomes the index number of the entry to be inserted. If you don't want to number the inserted revisions, then use nil for INDEX-REGEXP.

If INDEX-REGEXP is non-nil then both POINT-FORMAT and EOB-FORMAT should contain \"%N\", which is replaced with the number that was determined in the previous step.

Both formats, if non-nil and after removing %N, are then expanded using 'git show –format=FORMAT . . .' inside TOPLEVEL.

The expansion of POINT-FORMAT is inserted at point, and the expansion of EOB-FORMAT is inserted at the end of the buffer (if the buffer ends with a comment, then it is inserted right before that).

Some projects use pseudo headers in commit messages. Magit colorizes such headers and provides some commands to insert such headers.

`git-commit-known-pseudo-headers`   [User Option]

A list of Git pseudo headers to be highlighted.

`C-c C-a`   (`git-commit-ack`)

Insert a header acknowledging that you have looked at the commit.

`C-c C-r`   (`git-commit-review`)

Insert a header acknowledging that you have reviewed the commit.

`C-c C-s`   (git-commit-signoff)
　　　　　Insert a header to sign off the commit.

`C-c C-t`   (git-commit-test)
　　　　　Insert a header acknowledging that you have tested the commit.

`C-c C-o`   (git-commit-cc)
　　　　　Insert a header mentioning someone who might be interested.

`C-c C-p`   (git-commit-reported)
　　　　　Insert a header mentioning the person who reported the issue being fixed by
　　　　　the commit.

`C-c C-i`   (git-commit-suggested)
　　　　　Insert a header mentioning the person who suggested the change.

　　`git-commit-mode` is a minor mode that is only used to establish the above key bindings. This allows using an arbitrary major mode when editing the commit message. It's even possible to use a different major mode in different repositories, which is useful when different projects impose different commit message conventions.

`git-commit-major-mode`   [User Option]
　　　　The value of this option is the major mode used to edit Git commit messages.

　　Because `git-commit-mode` is a minor mode, we don't use its mode hook to setup the buffer, except for the key bindings. All other setup happens in the function `git-commit-setup`, which among other things runs the hook `git-commit-setup-hook`. The following functions are suitable for that hook.

`git-commit-setup-hook`   [User Option]
　　　　Hook run at the end of `git-commit-setup`.

`magit-revert-buffers` **&optional** *force*   [Function]
　　　　Revert unmodified file-visiting buffers of the current repository.

　　　　If either `magit-revert-buffers` is non-nil and `inhibit-magit-revert` is nil, or if optional FORCE is non-nil, then revert all unmodified buffers that visit files being tracked in the current repository.

`git-commit-save-message`   [Function]
　　　　Save the current buffer content to the commit message ring.

`git-commit-setup-changelog-support`   [Function]
　　　　After this function is called, ChangeLog entries are treated as paragraphs.

`git-commit-turn-on-auto-fill`   [Function]
　　　　Turn on `auto-fill-mode` and set `fill-column` to the value of `git-commit-fill-column`.

`git-commit-turn-on-flyspell`   [Function]
　　　　Turn on Flyspell mode. Also prevent comments from being checked and finally check current non-comment text.

`git-commit-propertize-diff`   [Function]
>   Propertize the diff shown inside the commit message buffer. Git inserts such diffs into
>   the commit message template when the `--verbose` argument is used. `magit-commit`
>   by default does not offer that argument because the diff that is shown in a separate
>   buffer is more useful. But some users disagree, which is why this function exists.

`with-editor-usage-message`   [Function]
>   Show usage information in the echo area.

Magit also helps with writing **good** commit messages by complaining when certain rules
are violated.

`git-commit-summary-max-length`   [User Option]
>   The intended maximal length of the summary line of commit messages. Characters
>   beyond this column are colorized to indicate that this preference has been violated.

`git-commit-fill-column`   [User Option]
>   Column beyond which automatic line-wrapping should happen in commit message
>   buffers.

`git-commit-finish-query-functions`   [User Option]
>   List of functions called to query before performing commit.
>
>   The commit message buffer is current while the functions are called. If any of them
>   returns nil, then the commit is not performed and the buffer is not killed. The user
>   should then fix the issue and try again.
>
>   The functions are called with one argument. If it is non-nil then that indicates that
>   the user used a prefix argument to force finishing the session despite issues. Functions
>   should usually honor this wish and return non-nil.

`git-commit-check-style-conventions`   [Function]
>   Check for violations of certain basic style conventions. For each violation ask the user
>   if she wants to proceed anyway. This makes sure the summary line isn't too long and
>   that the second line is empty.

To show no diff while committing remove `magit-commit-diff` from `server-switch-hook`.

6.6 Branching

6.6.1 The Two Remotes

The upstream branch of some local branch is the branch into which the commits on that
local branch should eventually be merged, usually something like `origin/master`. For the
`master` branch itself the upstream branch and the branch it is being pushed to, are usually
the same remote branch. But for a feature branch the upstream branch and the branch it
is being pushed to should differ.

The commits on feature branches too should *eventually* end up in a remote branch
such as `origin/master` or `origin/maint`. Such a branch should therefore be used as the
upstream. But feature branches shouldn't be pushed directly to such branches. Instead

a feature branch `my-feature` is usually pushed to `my-fork/my-feature` or if you are a contributor `origin/my-feature`. After the new feature has been reviewed, the maintainer merges the feature into `master`. And finally `master` (not `my-feature` itself) is pushed to `origin/master`.

But new features seldom are perfect on the first try, and so feature branches usually have to be reviewed, improved, and re-pushed several times. Pushing should therefore be easy to do, and for that reason many Git users have concluded that it is best to use the remote branch to which the local feature branch is being pushed as its upstream.

But luckily Git has long ago gained support for a push-remote which can be configured separately from the upstream branch, using the variables `branch.<name>.pushRemote` and `remote.pushDefault`. So we no longer have to choose which of the two remotes should be used as "the remote".

Each of the fetching, pulling, and pushing transient commands features three suffix commands that act on the current branch and some other branch. Of these, `p` is bound to a command which acts on the push-remote, `u` is bound to a command which acts on the upstream, and `e` is bound to a command which acts on any other branch. The status buffer shows unpushed and unpulled commits for both the push-remote and the upstream.

It's fairly simple to configure these two remotes. The values of all the variables that are related to fetching, pulling, and pushing (as well as some other branch-related variables) can be inspected and changed using the command `magit-branch-configure`, which is available from many transient prefix commands that deal with branches. It is also possible to set the push-remote or upstream while pushing (see Section 7.4 [Pushing], page 96).

6.6.2 Branch Commands

The transient prefix command `magit-branch` is used to create and checkout branches, and to make changes to existing branches. It is not used to fetch, pull, merge, rebase, or push branches, i.e. this command deals with branches themselves, not with the commits reachable from them. Those features are available from separate transient command.

b   (`magit-branch`)

This transient prefix command binds the following suffix commands and displays them in a temporary buffer until a suffix is invoked.

By default it also binds and displays the values of some branch-related Git variables and allows changing their values.

`magit-branch-direct-configure`   [User Option]

This option controls whether the transient command `magit-branch` can be used directly change the values Git variables. This defaults to `t` (to avoid changing key bindings). When set to `nil`, then no variables are displayed by that transient command, and its suffix command `magit-branch-configure` has to be used instead to view and change branch related variables.

`b C`   (magit-branch-configure)
`f C`   (magit-branch-configure)
`F C`   (magit-branch-configure)
`P C`   (magit-branch-configure)
>   This transient prefix command binds commands that set the value of branch-
>   related variables and displays them in a temporary buffer until the transient is
>   exited.
>
>   With a prefix argument, this command always prompts for a branch.
>
>   Without a prefix argument this depends on whether it was invoked as a suf-
>   fix of `magit-branch` and on the `magit-branch-direct-configure` option. If
>   `magit-branch` already displays the variables for the current branch, then it
>   isn't useful to invoke another transient that displays them for the same branch.
>   In that case this command prompts for a branch.

The variables are described in Section 6.6.3 [Branch Git Variables], page 76.

`b b`   (magit-checkout)
>   Checkout a revision read in the minibuffer and defaulting to the branch or ar-
>   bitrary revision at point. If the revision is a local branch then that becomes the
>   current branch. If it is something else then `HEAD` becomes detached. Checkout
>   fails if the working tree or the staging area contain changes.

`b n`   (magit-branch-create)
>   Create a new branch. The user is asked for a branch or arbitrary revision to
>   use as the starting point of the new branch. When a branch name is provided,
>   then that becomes the upstream branch of the new branch. The name of the
>   new branch is also read in the minibuffer.
>
>   Also see option `magit-branch-prefer-remote-upstream`.

`b c`   (magit-branch-and-checkout)
>   This command creates a new branch like `magit-branch`, but then also checks
>   it out.
>
>   Also see option `magit-branch-prefer-remote-upstream`.

`b l`   (magit-branch-checkout)
>   This command checks out an existing or new local branch. It reads a branch
>   name from the user offering all local branches and a subset of remote branches as
>   candidates. Remote branches for which a local branch by the same name exists
>   are omitted from the list of candidates. The user can also enter a completely
>   new branch name.
>
>   • If the user selects an existing local branch, then that is checked out.
>   • If the user selects a remote branch, then it creates and checks out a new
>   local branch with the same name, and configures the selected remote branch
>   as the push target.
>   • If the user enters a new branch name, then it creates and checks that out,
>   after also reading the starting-point from the user.
>
>   In the latter two cases the upstream is also set. Whether it is set to the
>   chosen starting point or something else depends on the value of `magit-branch-`
>   `adjust-remote-upstream-alist`.

b s   (magit-branch-spinoff)

    This command creates and checks out a new branch starting at and tracking the current branch. That branch in turn is reset to the last commit it shares with its upstream. If the current branch has no upstream or no unpushed commits, then the new branch is created anyway and the previously current branch is not touched.

    This is useful to create a feature branch after work has already began on the old branch (likely but not necessarily "master").

    If the current branch is a member of the value of option `magit-branch-prefer-remote-upstream` (which see), then the current branch will be used as the starting point as usual, but the upstream of the starting-point may be used as the upstream of the new branch, instead of the starting-point itself.

    If optional FROM is non-nil, then the source branch is reset to `FROM~`, instead of to the last commit it shares with its upstream. Interactively, FROM is only ever non-nil, if the region selects some commits, and among those commits, FROM is the commit that is the fewest commits ahead of the source branch.

    The commit at the other end of the selection actually does not matter, all commits between FROM and `HEAD` are moved to the new branch. If FROM is not reachable from `HEAD` or is reachable from the source branch's upstream, then an error is raised.

b S   (magit-branch-spinout)

    This command behaves like `magit-branch-spinoff`, except that it does not change the current branch. If there are any uncommitted changes, then it behaves exactly like `magit-branch-spinoff`.

b x   (magit-branch-reset)

    This command resets a branch, defaulting to the branch at point, to the tip of another branch or any other commit.

    When the branch being reset is the current branch, then a hard reset is performed. If there are any uncommitted changes, then the user has to confirm the reset because those changes would be lost.

    This is useful when you have started work on a feature branch but realize it's all crap and want to start over.

    When resetting to another branch and a prefix argument is used, then the target branch is set as the upstream of the branch that is being reset.

b k   (magit-branch-delete)

    Delete one or multiple branches. If the region marks multiple branches, then offer to delete those. Otherwise, prompt for a single branch to be deleted, defaulting to the branch at point.

b r   (magit-branch-rename)

    Rename a branch. The branch and the new name are read in the minibuffer. With prefix argument the branch is renamed even if that name conflicts with an existing branch.

`magit-branch-read-upstream-first`   [User Option]

> When creating a branch, whether to read the upstream branch before the name of the branch that is to be created. The default is `nil`, and I recommend you leave it at that.

`magit-branch-prefer-remote-upstream`   [User Option]

> This option specifies whether remote upstreams are favored over local upstreams when creating new branches.
>
> When a new branch is created, then the branch, commit, or stash at point is suggested as the starting point of the new branch, or if there is no such revision at point the current branch. In either case the user may choose another starting point.
>
> If the chosen starting point is a branch, then it may also be set as the upstream of the new branch, depending on the value of the Git variable 'branch.autoSetupMerge'. By default this is done for remote branches, but not for local branches.
>
> You might prefer to always use some remote branch as upstream. If the chosen starting point is (1) a local branch, (2) whose name matches a member of the value of this option, (3) the upstream of that local branch is a remote branch with the same name, and (4) that remote branch can be fast-forwarded to the local branch, then the chosen branch is used as starting point, but its own upstream is used as the upstream of the new branch.
>
> Members of this option's value are treated as branch names that have to match exactly unless they contain a character that makes them invalid as a branch name. Recommended characters to use to trigger interpretation as a regexp are `"*"` and `"^"`. Some other characters which you might expect to be invalid, actually are not, e.g. `".+$"` are all perfectly valid. More precisely, if 'git check-ref-format –branch STRING' exits with a non-zero status, then treat STRING as a regexp.
>
> Assuming the chosen branch matches these conditions you would end up with with e.g.:
>
> ```
>       feature --upstream--> origin/master
> ```
>
> instead of
>
> ```
>       feature --upstream--> master --upstream--> origin/master
> ```
>
> Which you prefer is a matter of personal preference. If you do prefer the former, then you should add branches such as `master`, `next`, and `maint` to the value of this options.

`magit-branch-adjust-remote-upstream-alist`   [User Option]

> The value of this option is an alist of branches to be used as the upstream when branching a remote branch.
>
> When creating a local branch from an ephemeral branch located on a remote, e.g. a feature or hotfix branch, then that remote branch should usually not be used as the upstream branch, since the push-remote already allows accessing it and having both the upstream and the push-remote reference the same related branch would be wasteful. Instead a branch like "maint" or "master" should be used as the upstream.
>
> This option allows specifying the branch that should be used as the upstream when branching certain remote branches. The value is an alist of the form `((UPSTREAM . RULE)...)`. The first matching element is used, the following elements are ignored.

UPSTREAM is the branch to be used as the upstream for branches specified by RULE. It can be a local or a remote branch.

RULE can either be a regular expression, matching branches whose upstream should be the one specified by UPSTREAM. Or it can be a list of the only branches that should **not** use UPSTREAM; all other branches will. Matching is done after stripping the remote part of the name of the branch that is being branched from.

If you use a finite set of non-ephemeral branches across all your repositories, then you might use something like:

```
(("origin/master" "master" "next" "maint"))
```

Or if the names of all your ephemeral branches contain a slash, at least in some repositories, then a good value could be:

```
(("origin/master" . "/"))
```

Of course you can also fine-tune:

```
(("origin/maint" . "\\`hotfix/")
 ("origin/master" . "\\`feature/"))
```

`magit-branch-orphan`   [Command]

This command creates and checks out a new orphan branch with contents from a given revision.

`magit-branch-or-checkout`   [Command]

This command is a hybrid between `magit-checkout` and `magit-branch-and-checkout` and is intended as a replacement for the former in `magit-branch`.

It first asks the user for an existing branch or revision. If the user input actually can be resolved as a branch or revision, then it checks that out, just like `magit-checkout` would.

Otherwise it creates and checks out a new branch using the input as its name. Before doing so it reads the starting-point for the new branch. This is similar to what `magit-branch-and-checkout` does.

To use this command instead of `magit-checkout` add this to your init file:

```
(transient-replace-suffix 'magit-branch 'magit-checkout
  '("b" "dwim" magit-branch-or-checkout))
```

6.6.3 Branch Git Variables

These variables can be set from the transient prefix command `magit-branch-configure`. By default they can also be set from `magit-branch`. See Section 6.6.2 [Branch Commands], page 72.

`branch.NAME.merge`   [Variable]

Together with `branch.NAME.remote` this variable defines the upstream branch of the local branch named NAME. The value of this variable is the full reference of the upstream *branch*.

`branch.NAME.remote`   [Variable]

Together with `branch.NAME.merge` this variable defines the upstream branch of the local branch named NAME. The value of this variable is the name of the upstream *remote*.

`branch.NAME.rebase`   [Variable]

This variable controls whether pulling into the branch named NAME is done by rebasing or by merging the fetched branch.

- When `true` then pulling is done by rebasing.
- When `false` then pulling is done by merging.
- When undefined then the value of `pull.rebase` is used. The default of that variable is `false`.

`branch.NAME.pushRemote`   [Variable]

This variable specifies the remote that the branch named NAME is usually pushed to. The value has to be the name of an existing remote.

It is not possible to specify the name of *branch* to push the local branch to. The name of the remote branch is always the same as the name of the local branch.

If this variable is undefined but `remote.pushDefault` is defined, then the value of the latter is used. By default `remote.pushDefault` is undefined.

`branch.NAME.description`   [Variable]

This variable can be used to describe the branch named NAME. That description is used e.g. when turning the branch into a series of patches.

The following variables specify defaults which are used if the above branch-specific variables are not set.

`pull.rebase`   [Variable]

This variable specifies whether pulling is done by rebasing or by merging. It can be overwritten using `branch.NAME.rebase`.

- When `true` then pulling is done by rebasing.
- When `false` (the default) then pulling is done by merging.

Since it is never a good idea to merge the upstream branch into a feature or hotfix branch and most branches are such branches, you should consider setting this to `true`, and `branch.master.rebase` to `false`.

`remote.pushDefault`   [Variable]

This variable specifies what remote the local branches are usually pushed to. This can be overwritten per branch using `branch.NAME.pushRemote`.

The following variables are used during the creation of a branch and control whether the various branch-specific variables are automatically set at this time.

`branch.autoSetupMerge`   [Variable]

This variable specifies under what circumstances creating a branch NAME should result in the variables `branch.NAME.merge` and `branch.NAME.remote` being set according to the starting point used to create the branch. If the starting point isn't a branch, then these variables are never set.

- When `always` then the variables are set regardless of whether the starting point is a local or a remote branch.

- When `true` (the default) then the variables are set when the starting point is a remote branch, but not when it is a local branch.
- When `false` then the variables are never set.

`branch.autoSetupRebase`   [Variable]

This variable specifies whether creating a branch NAME should result in the variable `branch.NAME.rebase` being set to `true`.

- When `always` then the variable is set regardless of whether the starting point is a local or a remote branch.
- When `local` then the variable are set when the starting point is a local branch, but not when it is a remote branch.
- When `remote` then the variable are set when the starting point is a remote branch, but not when it is a local branch.
- When `never` (the default) then the variable is never set.

Note that the respective commands always change the repository-local values. If you want to change the global value, which is used when the local value is undefined, then you have to do so on the command line, e.g.:

    git config --global remote.autoSetupMerge always

For more information about these variables you should also see

the git-config(1) manpage. Also see the git-branch(1) manpage. , the git-checkout(1) manpage. and Section 7.4 [Pushing], page 96.

`magit-prefer-remote-upstream`   [User Option]

This option controls whether commands that read a branch from the user and then set it as the upstream branch, offer a local or a remote branch as default completion candidate, when they have the choice.

This affects all commands that use `magit-read-upstream-branch` or `magit-read-starting-point`, which includes all commands that change the upstream and many which create new branches.

6.6.4 Auxiliary Branch Commands

These commands are not available from the transient `magit-branch` by default.

`magit-branch-shelve`   [Command]

This command shelves a branch. This is done by deleting the branch, and creating a new reference "refs/shelved/BRANCH-NAME" pointing at the same commit as the branch pointed at. If the deleted branch had a reflog, then that is preserved as the reflog of the new reference.

This is useful if you want to move a branch out of sight, but are not ready to completely discard it yet.

`magit-branch-unshelve`   [Command]

This command unshelves a branch that was previously shelved using `magit-branch-shelve`. This is done by deleting the reference "refs/shelved/BRANCH-NAME" and creating a branch "BRANCH-NAME" pointing at the same commit as the deleted reference pointed at. If the deleted reference had a reflog, then that is restored as the reflog of the branch.

6.7 Merging

Also see the git-merge(1) manpage. For information on how to resolve merge conflicts see the next section.

`m` (`magit-merge`)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

When no merge is in progress, then the transient features the following suffix commands.

`m m` (`magit-merge-plain`)

> This command merges another branch or an arbitrary revision into the current branch. The branch or revision to be merged is read in the minibuffer and defaults to the branch at point.
>
> Unless there are conflicts or a prefix argument is used, then the resulting merge commit uses a generic commit message, and the user does not get a chance to inspect or change it before the commit is created. With a prefix argument this does not actually create the merge commit, which makes it possible to inspect how conflicts were resolved and to adjust the commit message.

`m e` (`magit-merge-editmsg`)

> This command merges another branch or an arbitrary revision into the current branch and opens a commit message buffer, so that the user can make adjustments. The commit is not actually created until the user finishes with `C-c C-c`.

`m n` (`magit-merge-nocommit`)

> This command merges another branch or an arbitrary revision into the current branch, but does not actually create the merge commit. The user can then further adjust the merge, even when automatic conflict resolution succeeded and/or adjust the commit message.

`m a` (`magit-merge-absorb`)

> This command merges another local branch into the current branch and then removes the former.
>
> Before the source branch is merged, it is first force pushed to its push-remote, provided the respective remote branch already exists. This ensures that the respective pull-request (if any) won't get stuck on some obsolete version of the commits that are being merged. Finally, if `magit-branch-pull-request` was used to create the merged branch, then the respective remote branch is also removed.

`m i` (`magit-merge-into`)

> This command merges the current branch into another local branch and then removes the former. The latter becomes the new current branch.
>
> Before the source branch is merged, it is first force pushed to its push-remote, provided the respective remote branch already exists. This ensures that the respective pull-request (if any) won't get stuck on some obsolete version of the commits that are being merged. Finally, if `magit-branch-pull-request` was

> used to create the merged branch, then the respective remote branch is also removed.

m s   (`magit-merge-squash`)

> This command squashes the changes introduced by another branch or an arbitrary revision into the current branch. This only applies the changes made by the squashed commits. No information is preserved that would allow creating an actual merge commit. Instead of this command you should probably use a command from the apply transient.

m p   (`magit-merge-preview`)

> This command shows a preview of merging another branch or an arbitrary revision into the current branch.

When a merge is in progress, then the transient instead features the following suffix commands.

m m   (`magit-merge`)

> After the user resolved conflicts, this command proceeds with the merge. If some conflicts weren't resolved, then this command fails.

m a   (`magit-merge-abort`)

> This command aborts the current merge operation.

6.8 Resolving Conflicts

When merging branches (or otherwise combining or changing history) conflicts can occur. If you edited two completely different parts of the same file in two branches and then merge one of these branches into the other, then Git can resolve that on its own, but if you edit the same area of a file, then a human is required to decide how the two versions, or "sides of the conflict", are to be combined into one.

Here we can only provide a brief introduction to the subject and point you toward some tools that can help. If you are new to this, then please also consult Git's own documentation as well as other resources.

If a file has conflicts and Git cannot resolve them by itself, then it puts both versions into the affected file along with special markers whose purpose is to denote the boundaries of the unresolved part of the file and between the different versions. These boundary lines begin with the strings consisting of six times the same character, one of <, |, = and > and are followed by information about the source of the respective versions, e.g.:

```
<<<<<<< HEAD
Take the blue pill.
=======
Take the red pill.
>>>>>>> feature
```

In this case you have chosen to take the red pill on one branch and on another you picked the blue pill. Now that you are merging these two diverging branches, Git cannot possibly know which pill you want to take.

To resolve that conflict you have to create a version of the affected area of the file by keeping only one of the sides, possibly by editing it in order to bring in the changes from

the other side, remove the other versions as well as the markers, and then stage the result. A possible resolution might be:

```
Take both pills.
```

Often it is useful to see not only the two sides of the conflict but also the "original" version from before the same area of the file was modified twice on different branches. Instruct Git to insert that version as well by running this command once:

```
git config --global merge.conflictStyle diff3
```

The above conflict might then have looked like this:

```
<<<<<<< HEAD
Take the blue pill.
||||||| merged common ancestors
Take either the blue or the red pill, but not both.
=======
Take the red pill.
>>>>>>> feature
```

If that were the case, then the above conflict resolution would not have been correct, which demonstrates why seeing the original version alongside the conflicting versions can be useful.

You can perform the conflict resolution completely by hand, but Emacs also provides some packages that help in the process: Smerge, Ediff (`ediff`), and Emerge (Section "Emerge" in `emacs`). Magit does not provide its own tools for conflict resolution, but it does make using Smerge and Ediff more convenient. (Ediff supersedes Emerge, so you probably don't want to use the latter anyway.)

In the Magit status buffer, files with unresolved conflicts are listed in the "Unstaged changes" and/or "Staged changes" sections. They are prefixed with the word "unmerged", which in this context essentially is a synonym for "unresolved".

Pressing `RET` while point is on such a file section shows a buffer visiting that file, turns on `smerge-mode` in that buffer, and places point inside the first area with conflicts. You should then resolve that conflict using regular edit commands and/or Smerge commands.

Unfortunately Smerge does not have a manual, but you can get a list of commands and binding `C-c ^ C-h` and press `RET` while point is on a command name to read its documentation.

Normally you would edit one version and then tell Smerge to keep only that version. Use `C-c ^ m` (`smerge-keep-mine`) to keep the `HEAD` version or `C-c ^ o` (`smerge-keep-other`) to keep the version that follows "|||||||". Then use `C-c ^ n` to move to the next conflicting area in the same file. Once you are done resolving conflicts, return to the Magit status buffer. The file should now be shown as "modified", no longer as "unmerged", because Smerge automatically stages the file when you save the buffer after resolving the last conflict.

Alternatively you could use Ediff, which uses separate buffers for the different versions of the file. To resolve conflicts in a file using Ediff press `e` while point is on such a file in the status buffer.

Ediff can be used for other purposes as well. For more information on how to enter Ediff from Magit, see Section 5.5 [Ediffing], page 51. Explaining how to use Ediff is beyond the scope of this manual, instead see `ediff`.

If you are unsure whether you should Smerge or Ediff, then use the former. It is much easier to understand and use, and except for truly complex conflicts, the latter is usually overkill.

6.9 Rebasing

Also see the git-rebase(1) manpage. For information on how to resolve conflicts that occur during rebases see the preceding section.

`r`   (`magit-rebase`)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

When no rebase is in progress, then the transient features the following suffix commands.

Using one of these commands *starts* a rebase sequence. Git might then stop somewhere along the way, either because you told it to do so, or because applying a commit failed due to a conflict. When that happens, then the status buffer shows information about the rebase sequence which is in progress in a section similar to a log section. See Section 6.9.2 [Information About In-Progress Rebase], page 85.

For information about the upstream and the push-remote, see Section 6.6.1 [The Two Remotes], page 71.

`r p`   (`magit-rebase-onto-pushremote`)

> This command rebases the current branch onto its push-remote.
>
> When `magit-remote-set-if-missing` is non-nil and the push-remote is not configured, then configure it before rebasing. With a prefix argument let the user change the push-remote before rebasing onto it.

`r u`   (`magit-rebase-onto-upstream`)

> This command rebases the current branch onto its upstream branch.
>
> When `magit-remote-set-if-missing` is non-nil and the upstream is not configured, then configure it before rebasing. With a prefix argument let the user change the upstream before rebasing onto it.

`r e`   (`magit-rebase-branch`)

> This command rebases the current branch onto a branch read in the minibuffer. All commits that are reachable from head but not from the selected branch TARGET are being rebased.

`r s`   (`magit-rebase-subset`)

> This command starts a non-interactive rebase sequence to transfer commits from START to `HEAD` onto NEWBASE. START has to be selected from a list of recent commits.

By default Magit uses the `--autostash` argument, which causes uncommitted changes to be stored in a stash before the rebase begins. These changes are restored after the rebase completes and if possible the stash is removed. If the stash does not apply cleanly, then the stash is not removed. In case something goes wrong when resolving the conflicts, this allows you to start over.

Even though one of the actions is dedicated to interactive rebases, the transient also features the infix argument `--interactive`. This can be used to turn one of the other, non-interactive rebase variants into an interactive rebase.

For example if you want to clean up a feature branch and at the same time rebase it onto `master`, then you could use `r-iu`. But we recommend that you instead do that in two steps. First use `ri` to cleanup the feature branch, and then in a second step `ru` to rebase it onto `master`. That way if things turn out to be more complicated than you thought and/or you make a mistake and have to start over, then you only have to redo half the work.

Explicitly enabling `--interactive` won't have an effect on the following commands as they always use that argument anyway, even if it is not enabled in the transient.

r i   (magit-rebase-interactive)
>   This command starts an interactive rebase sequence.

r f   (magit-rebase-autosquash)
>   This command combines squash and fixup commits with their intended targets.

r m   (magit-rebase-edit-commit)
>   This command starts an interactive rebase sequence that lets the user edit a single older commit.

r w   (magit-rebase-reword-commit)
>   This command starts an interactive rebase sequence that lets the user reword a single older commit.

r k   (magit-rebase-remove-commit)
>   This command removes a single older commit using rebase.

When a rebase is in progress, then the transient instead features the following suffix commands.

r r   (magit-rebase-continue)
>   This command restart the current rebasing operation.
>
>   In some cases this pops up a commit message buffer for you do edit. With a prefix argument the old message is reused as-is.

r s   (magit-rebase-skip)
>   This command skips the current commit and restarts the current rebase operation.

r e   (magit-rebase-edit)
>   This command lets the user edit the todo list of the current rebase operation.

r a   (magit-rebase-abort)
>   This command aborts the current rebase operation, restoring the original branch.

6.9.1 Editing Rebase Sequences

C-c C-c   (with-editor-finish)
>   Finish the current editing session by returning with exit code 0. Git then uses the rebase instructions it finds in the file.

`C-c C-k`   (with-editor-cancel)
> Cancel the current editing session by returning with exit code 1. Git then forgoes starting the rebase sequence.

`RET`   (git-rebase-show-commit)
> Show the commit on the current line in another buffer and select that buffer.

`SPC`   (git-rebase-show-or-scroll-up)
> Show the commit on the current line in another buffer without selecting that buffer. If the revision buffer is already visible in another window of the current frame, then instead scroll that window up.

`DEL`   (git-rebase-show-or-scroll-down)
> Show the commit on the current line in another buffer without selecting that buffer. If the revision buffer is already visible in another window of the current frame, then instead scroll that window down.

`p`   (git-rebase-backward-line)
> Move to previous line.

`n`   (forward-line)
> Move to next line.

`M-p`   (git-rebase-move-line-up)
> Move the current commit (or command) up.

`M-n`   (git-rebase-move-line-down)
> Move the current commit (or command) down.

`r`   (git-rebase-reword)
> Edit message of commit on current line.

`e`   (git-rebase-edit)
> Stop at the commit on the current line.

`s`   (git-rebase-squash)
> Meld commit on current line into previous commit, and edit message.

`f`   (git-rebase-fixup)
> Meld commit on current line into previous commit, discarding the current commit's message.

`k`   (git-rebase-kill-line)
> Kill the current action line.

`c`   (git-rebase-pick)
> Use commit on current line.

`x`   (git-rebase-exec)
> Insert a shell command to be run after the proceeding commit.
>
> If there already is such a command on the current line, then edit that instead. With a prefix argument insert a new command even when there already is one on the current line. With empty input remove the command on the current line, if any.

`b`   (git-rebase-break)
> Insert a break action before the current line, instructing Git to return control to the user.

`y`   (git-rebase-insert)
> Read an arbitrary commit and insert it below current line.

`C-x u`   (git-rebase-undo)
> Undo some previous changes. Like `undo` but works in read-only buffers.

`git-rebase-auto-advance`   [User Option]
> Whether to move to next line after changing a line.

`git-rebase-show-instructions`   [User Option]
> Whether to show usage instructions inside the rebase buffer.

`git-rebase-confirm-cancel`   [User Option]
> Whether confirmation is required to cancel.

When a rebase is performed with the `--rebase-merges` option, the sequence will include a few other types of actions and the following commands become relevant.

`l`   (git-rebase-label)
> This commands inserts a label action or edits the one at point.

`t`   (git-rebase-reset)
> This command inserts a reset action or edits the one at point. The prompt will offer the labels that are currently present in the buffer.

`MM`   (git-rebase-merge)
> The command inserts a merge action or edits the one at point. The prompt will offer the labels that are currently present in the buffer. Specifying a message to reuse via `-c` or `-C` is not supported; an editor will always be invoked for the merge.

`Mt`   (git-rebase-merge-toggle-editmsg)
> This command toggles between the `-C` and `-c` options of the merge action at point. These options both specify a commit whose message should be reused. The lower-case variant instructs Git to invoke the editor when creating the merge, allowing the user to edit the message.

6.9.2 Information About In-Progress Rebase

While a rebase sequence is in progress, the status buffer features a section that lists the commits that have already been applied as well as the commits that still have to be applied.

The commits are split in two halves. When rebase stops at a commit, either because the user has to deal with a conflict or because s/he explicitly requested that rebase stops at that commit, then point is placed on the commit that separates the two groups, i.e. on `HEAD`. The commits above it have not been applied yet, while the `HEAD` and the commits below it have already been applied. In between these two groups of applied and yet-to-be applied commits, there sometimes is a commit which has been dropped.

Each commit is prefixed with a word and these words are additionally shown in different colors to indicate the status of the commits.

The following colors are used:

- Yellow commits have not been applied yet.
- Gray commits have already been applied.
- The blue commit is the `HEAD` commit.
- The green commit is the commit the rebase sequence stopped at. If this is the same commit as `HEAD` (e.g. because you haven't done anything yet after rebase stopped at the commit, then this commit is shown in blue, not green). There can only be a green **and** a blue commit at the same time, if you create one or more new commits after rebase stops at a commit.
- Red commits have been dropped. They are shown for reference only, e.g. to make it easier to diff.

Of course these colors are subject to the color-theme in use.

The following words are used:

- Commits prefixed with `pick`, `reword`, `edit`, `squash`, and `fixup` have not been applied yet. These words have the same meaning here as they do in the buffer used to edit the rebase sequence. See Section 6.9.1 [Editing Rebase Sequences], page 83. When the `--rebase-merges` option was specified, `reset`, `label`, and `merge` lines may also be present.
- Commits prefixed with `done` and `onto` have already been applied. It is possible for such a commit to be the `HEAD`, in which case it is blue. Otherwise it is grey.
  - The commit prefixed with `onto` is the commit on top of which all the other commits are being re-applied. This commit itself did not have to be re-applied, it is the commit rebase did rewind to before starting to re-apply other commits.
  - Commits prefixed with `done` have already been re-applied. This includes commits that have been re-applied but also new commits that you have created during the rebase.
- All other commits, those not prefixed with any of the above words, are in some way related to the commit at which rebase stopped.

  To determine whether a commit is related to the stopped-at commit their hashes, trees and patch-ids[1] are being compared. The commit message is not used for this purpose.

  Generally speaking commits that are related to the stopped-at commit can have any of the used colors, though not all color/word combinations are possible.

  Words used for stopped-at commits are:
  - When a commit is prefixed with `void`, then that indicates that Magit knows for sure that all the changes in that commit have been applied using several new

---

[1] The patch-id is a hash of the *changes* introduced by a commit. It differs from the hash of the commit itself, which is a hash of the result of applying that change (i.e. the resulting trees and blobs) as well as author and committer information, the commit message, and the hashes of the parents of the commit. The patch-id hash on the other hand is created only from the added and removed lines, even line numbers and whitespace changes are ignored when calculating this hash. The patch-ids of two commits can be used to answer the question "Do these commits make the same change?".

commits. This commit is no longer reachable from `HEAD`, and it also isn't one of the commits that will be applied when resuming the session.

- When a commit is prefixed with `join`, then that indicates that the rebase sequence stopped at that commit due to a conflict - you now have to join (merge) the changes with what has already been applied. In a sense this is the commit rebase stopped at, but while its effect is already in the index and in the worktree (with conflict markers), the commit itself has not actually been applied yet (it isn't the `HEAD`). So it is shown in yellow, like the other commits that still have to be applied.

- When a commit is prefixed with `stop` or a *blue* or *green* `same`, then that indicates that rebase stopped at this commit, that it is still applied or has been applied again, and that at least its patch-id is unchanged.

  - When a commit is prefixed with `stop`, then that indicates that rebase stopped at that commit because you requested that earlier, and its patch-id is unchanged. It might even still be the exact same commit.

  - When a commit is prefixed with a *blue* or *green* `same`, then that indicates that while its tree or hash changed, its patch-id did not. If it is blue, then it is the `HEAD` commit (as always for blue). When it is green, then it no longer is `HEAD` because other commit have been created since (but before continuing the rebase).

- When a commit is prefixed with `goal`, a *yellow* `same,` or `work`, then that indicates that rebase applied that commit but that you then reset `HEAD` to an earlier commit (likely to split it up into multiple commits), and that there are some uncommitted changes remaining which likely (but not necessarily) originate from that commit.

  - When a commit is prefixed with `goal`, then that indicates that it is still possible to create a new commit with the exact same tree (the "goal") without manually editing any files, by committing the index, or by staging all changes and then committing that. This is the case when the original tree still exists in the index or worktree in untainted form.

  - When a commit is prefixed with a yellow `same`, then that indicates that it is no longer possible to create a commit with the exact same tree, but that it is still possible to create a commit with the same patch-id. This would be the case if you created a new commit with other changes, but the changes from the original commit still exist in the index or working tree in untainted form.

  - When a commit is prefixed with `work`, then that indicates that you reset `HEAD` to an earlier commit, and that there are some staged and/or unstaged changes (likely, but not necessarily) originating from that commit. However it is no longer possible to create a new commit with the same tree or at least the same patch-id because you have already made other changes.

- When a commit is prefixed with `poof` or `gone`, then that indicates that rebase applied that commit but that you then reset `HEAD` to an earlier commit (likely to split it up into multiple commits), and that there are no uncommitted changes.

  - When a commit is prefixed with `poof`, then that indicates that it is no longer reachable from `HEAD`, but that it has been replaced with one or more commits, which together have the exact same effect.

- When a commit is prefixed with `gone`, then that indicates that it is no longer reachable from `HEAD` and that we also cannot determine whether its changes are still in effect in one or more new commits. They might be, but if so, then there must also be other changes which makes it impossible to know for sure.

Do not worry if you do not fully understand the above. That's okay, you will acquire a good enough understanding through practice.

For other sequence operations such as cherry-picking, a similar section is displayed, but they lack some of the features described above, due to limitations in the git commands used to implement them. Most importantly these sequences only support "picking" a commit but not other actions such as "rewording", and they do not keep track of the commits which have already been applied.

6.10 Cherry Picking

Also see the git-cherry-pick(1) manpage.

A (`magit-cherry-pick`)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

When no cherry-pick or revert is in progress, then the transient features the following suffix commands.

A A (`magit-cherry-copy`)

> This command copies COMMITS from another branch onto the current branch. If the region selects multiple commits, then those are copied, without prompting. Otherwise the user is prompted for a commit or range, defaulting to the commit at point.

A a (`magit-cherry-apply`)

> This command applies the changes in COMMITS from another branch onto the current branch. If the region selects multiple commits, then those are used, without prompting. Otherwise the user is prompted for a commit or range, defaulting to the commit at point.
>
> This command also has a top-level binding, which can be invoked without using the transient by typing `a` at the top-level.

The following commands not only apply some commits to some branch, but also remove them from some other branch. The removal is performed using either `git-update-ref` or if necessary `git-rebase`. Both applying commits as well as removing them using `git-rebase` can lead to conflicts. If that happens, then these commands abort and you not only have to resolve the conflicts but also finish the process the same way you would have to if these commands didn't exist at all.

A h (`magit-cherry-harvest`)

> This command moves the selected COMMITS that must be located on another BRANCH onto the current branch instead, removing them from the former. When this command succeeds, then the same branch is current as before.

Applying the commits on the current branch or removing them from the other branch can lead to conflicts. When that happens, then this command stops and you have to resolve the conflicts and then finish the process manually.

A d   (`magit-cherry-donate`)

This command moves the selected COMMITS from the current branch onto another existing BRANCH, removing them from the former. When this command succeeds, then the same branch is current as before.

Applying the commits on the other branch or removing them from the current branch can lead to conflicts. When that happens, then this command stops and you have to resolve the conflicts and then finish the process manually.

A n   (`magit-cherry-spinout`)

This command moves the selected COMMITS from the current branch onto a new branch BRANCH, removing them from the former. When this command succeeds, then the same branch is current as before.

Applying the commits on the other branch or removing them from the current branch can lead to conflicts. When that happens, then this command stops and you have to resolve the conflicts and then finish the process manually.

A s   (`magit-cherry-spinoff`)

This command moves the selected COMMITS from the current branch onto a new branch BRANCH, removing them from the former. When this command succeeds, then the new branch is checked out.

Applying the commits on the other branch or removing them from the current branch can lead to conflicts. When that happens, then this command stops and you have to resolve the conflicts and then finish the process manually.

When a cherry-pick or revert is in progress, then the transient instead features the following suffix commands.

A A   (`magit-sequence-continue`)

Resume the current cherry-pick or revert sequence.

A s   (`magit-sequence-skip`)

Skip the stopped at commit during a cherry-pick or revert sequence.

A a   (`magit-sequence-abort`)

Abort the current cherry-pick or revert sequence. This discards all changes made since the sequence started.

6.10.1 Reverting

V   (`magit-revert`)

This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

When no cherry-pick or revert is in progress, then the transient features the following suffix commands.

V V   (`magit-revert-and-commit`)
> Revert a commit by creating a new commit. Prompt for a commit, defaulting
> to the commit at point. If the region selects multiple commits, then revert all
> of them, without prompting.

V v   (`magit-revert-no-commit`)
> Revert a commit by applying it in reverse to the working tree. Prompt for
> a commit, defaulting to the commit at point. If the region selects multiple
> commits, then revert all of them, without prompting.

When a cherry-pick or revert is in progress, then the transient instead features the
following suffix commands.

V A   (`magit-sequence-continue`)
> Resume the current cherry-pick or revert sequence.

V s   (`magit-sequence-skip`)
> Skip the stopped at commit during a cherry-pick or revert sequence.

V a   (`magit-sequence-abort`)
> Abort the current cherry-pick or revert sequence. This discards all changes
> made since the sequence started.

6.11 Resetting

Also see the git-reset(1) manpage.

x   (`magit-reset-quickly`)
> Reset the `HEAD` and index to some commit read from the user and defaulting
> to the commit at point, and possibly also reset the working tree. With a prefix
> argument reset the working tree otherwise don't.

X m   (`magit-reset-mixed`)
> Reset the `HEAD` and index to some commit read from the user and defaulting
> to the commit at point. The working tree is kept as-is.

X s   (`magit-reset-soft`)
> Reset the `HEAD` to some commit read from the user and defaulting to the commit
> at point. The index and the working tree are kept as-is.

X h   (`magit-reset-hard`)
> Reset the `HEAD`, index, and working tree to some commit read from the user
> and defaulting to the commit at point.

X i   (`magit-reset-index`)
> Reset the index to some commit read from the user and defaulting to the commit
> at point. Keep the `HEAD` and working tree as-is, so if the commit refers to the
> `HEAD`, then this effectively unstages all changes.

X w   (`magit-reset-worktree`)
> Reset the working tree to some commit read from the user and defaulting to
> the commit at point. Keep the `HEAD` and index as-is.

X f   (`magit-file-checkout`)

> Update file in the working tree and index to the contents from a revision. Both the revision and file are read from the user.

6.12 Stashing

Also see the git-stash(1) manpage.

z   (`magit-stash`)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

z z   (`magit-stash-both`)

> Create a stash of the index and working tree. Untracked files are included according to infix arguments. One prefix argument is equivalent to `--include-untracked` while two prefix arguments are equivalent to `--all`.

z i   (`magit-stash-index`)

> Create a stash of the index only. Unstaged and untracked changes are not stashed.

z w   (`magit-stash-worktree`)

> Create a stash of unstaged changes in the working tree. Untracked files are included according to infix arguments. One prefix argument is equivalent to `--include-untracked` while two prefix arguments are equivalent to `--all`.

z x   (`magit-stash-keep-index`)

> Create a stash of the index and working tree, keeping index intact. Untracked files are included according to infix arguments. One prefix argument is equivalent to `--include-untracked` while two prefix arguments are equivalent to `--all`.

z Z   (`magit-snapshot-both`)

> Create a snapshot of the index and working tree. Untracked files are included according to infix arguments. One prefix argument is equivalent to `--include-untracked` while two prefix arguments are equivalent to `--all`.

z I   (`magit-snapshot-index`)

> Create a snapshot of the index only. Unstaged and untracked changes are not stashed.

z W   (`magit-snapshot-worktree`)

> Create a snapshot of unstaged changes in the working tree. Untracked files are included according to infix arguments. One prefix argument is equivalent to `--include-untracked` while two prefix arguments are equivalent to `--all-`.

z a   (`magit-stash-apply`)

> Apply a stash to the working tree. Try to preserve the stash index. If that fails because there are staged changes, apply without preserving the stash index.

z p   (`magit-stash-pop`)

> Apply a stash to the working tree and remove it from stash list. Try to preserve the stash index. If that fails because there are staged changes, apply without preserving the stash index and forgo removing the stash.

z k   (`magit-stash-drop`)

> Remove a stash from the stash list. When the region is active, offer to drop all contained stashes.

z v   (`magit-stash-show`)

> Show all diffs of a stash in a buffer.

z b   (`magit-stash-branch`)

> Create and checkout a new BRANCH from STASH. The branch starts at the commit that was current when the stash was created.

z B   (`magit-stash-branch-here`)

> Create and checkout a new BRANCH using `magit-branch` with the current branch or `HEAD` as the starting-point. Then apply STASH, dropping it if it applies cleanly.

z f   (`magit-stash-format-patch`)

> Create a patch from STASH.

k   (`magit-stash-clear`)

> Remove all stashes saved in REF's reflog by deleting REF.

z l   (`magit-stash-list`)

> List all stashes in a buffer.

`magit-stashes-margin`   [User Option]

> This option specifies whether the margin is initially shown in stashes buffers and how it is formatted.
>
> The value has the form (`INIT STYLE WIDTH AUTHOR AUTHOR-WIDTH`).
>
> - If INIT is non-nil, then the margin is shown initially.
> - STYLE controls how to format the committer date. It can be one of `age` (to show the age of the commit), `age-abbreviated` (to abbreviate the time unit to a character), or a string (suitable for `format-time-string`) to show the actual date.
> - WIDTH controls the width of the margin. This exists for forward compatibility and currently the value should not be changed.
> - AUTHOR controls whether the name of the author is also shown by default.
> - AUTHOR-WIDTH has to be an integer. When the name of the author is shown, then this specifies how much space is used to do so.

7 Transferring

7.1 Remotes

7.1.1 Remote Commands

The transient prefix command `magit-remote` is used to add remotes and to make changes to existing remotes. This command only deals with remotes themselves, not with branches or the transfer of commits. Those features are available from separate transient commands.

   Also see the git-remote(1) manpage.

M   (magit-remote)

   This transient prefix command binds the following suffix commands and displays them in a temporary buffer until a suffix is invoked.

   By default it also binds and displays the values of some remote-related Git variables and allows changing their values.

`magit-remote-direct-configure`   [User Option]

   This option controls whether remote-related Git variables are accessible directly from the transient `magit-remote`.

   If `t` (the default) and a local branch is checked out, then `magit-remote` features the variables for the upstream remote of that branch, or if `HEAD` is detached, for `origin`, provided that exists.

   If `nil`, then `magit-remote-configure` has to be used to do so.

M C   (magit-remote-configure)

   This transient prefix command binds commands that set the value of remote-related variables and displays them in a temporary buffer until the transient is exited.

   With a prefix argument, this command always prompts for a remote.

   Without a prefix argument this depends on whether it was invoked as a suffix of `magit-remote` and on the `magit-remote-direct-configure` option. If `magit-remote` already displays the variables for the upstream, then it does not make sense to invoke another transient that displays them for the same remote. In that case this command prompts for a remote.

   The variables are described in Section 7.1.2 [Remote Git Variables], page 94.

M a   (magit-remote-add)

   This command add a remote and fetches it. The remote name and url are read in the minibuffer.

M r   (magit-remote-rename)

   This command renames a remote. Both the old and the new names are read in the minibuffer.

M u   (magit-remote-set-url)

   This command changes the url of a remote. Both the remote and the new url are read in the minibuffer.

M k   (magit-remote-remove)
> This command deletes a remote, read in the minibuffer.

M p   (magit-remote-prune)
> This command removes stale remote-tracking branches for a remote read in the minibuffer.

M P   (magit-remote-prune-refspecs)
> This command removes stale refspecs for a remote read in the minibuffer.
>
> A refspec is stale if there no longer exists at least one branch on the remote that would be fetched due to that refspec. A stale refspec is problematic because its existence causes Git to refuse to fetch according to the remaining non-stale refspecs.
>
> If only stale refspecs remain, then this command offers to either delete the remote or to replace the stale refspecs with the default refspec ("+refs/heads/*:refs/remotes/REMOTE/*").
>
> This command also removes the remote-tracking branches that were created due to the now stale refspecs. Other stale branches are not removed.

`magit-remote-add-set-remote.pushDefault`   [User Option]
> This option controls whether the user is asked whether they want to set `remote.pushDefault` after adding a remote.
>
> If `ask`, then users is always ask. If `ask-if-unset`, then the user is only if the variable isn't set already. If `nil`, then the user isn't asked and the variable isn't set. If the value is a string, then the variable is set without the user being asked, provided that the name of the added remote is equal to that string and the variable isn't already set.

7.1.2 Remote Git Variables

These variables can be set from the transient prefix command `magit-remote-configure`. By default they can also be set from `magit-remote`. See Section 7.1.1 [Remote Commands], page 93.

`remote.NAME.url`   [Variable]
> This variable specifies the url of the remote named NAME. It can have multiple values.

`remote.NAME.fetch`   [Variable]
> The refspec used when fetching from the remote named NAME. It can have multiple values.

`remote.NAME.pushurl`   [Variable]
> This variable specifies the url used for fetching from the remote named NAME. If it is not specified, then `remote.NAME.url` is used instead. It can have multiple values.

`remote.NAME.push`   [Variable]
> The refspec used when pushing to the remote named NAME. It can have multiple values.

`remote.NAME.tagOpts`   [Variable]

> This variable specifies what tags are fetched by default. If the value is `--no-tags` then no tags are fetched. If the value is `--tags`, then all tags are fetched. If this variable has no value, then only tags are fetched that are reachable from fetched branches.

`magit-remote-set-if-missing`   [User Option]

> This option controls whether missing remotes are configured before fetching, pulling, pushing and rebasing.
>
> When `nil`, then commands such as `magit-pull-from-upstream` and `magit-rebase-onto-upstream` are not available as suffix commands if the push-remote or upstream is not configured.
>
> When `non-nil`, then these commands are always available in their respective transient, but if the required configuration is missing, then they do appear in a way that indicates that this is the case. If the user invokes such a command, then it asks for the necessary configuration, stores the configuration, and only then performs the action.
>
> This option also affects whether the argument `--set-upstream` is available in `magit-push`. If the value is `non-nil`, then that argument would be redundant. But note that changing the value of this option does not take affect immediately, the argument will only be added or removed after restarting Emacs.

7.2 Fetching

Also see the git-fetch(1) manpage. For information about the upstream and the push-remote, see Section 6.6.1 [The Two Remotes], page 71.

`f`   (`magit-fetch`)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

`f p`   (`magit-fetch-from-pushremote`)

> This command fetches from the push-remote of the current branch.
>
> When `magit-remote-set-if-missing` is non-nil and the push-remote is not configured, then configure it before fetching. With a prefix argument let the user change the push-remote before fetching from it.

`f u`   (`magit-fetch-from-upstream`)

> This command fetch from the upstream of the current branch.
>
> When `magit-remote-set-if-missing` is non-nil and the upstream is not configured, then configure it before fetching. With a prefix argument let the user change the upstream before fetching from it.

`f e`   (`magit-fetch-other`)

> This command fetch from a repository read from the minibuffer.

`f o`   (`magit-fetch-branch`)

> This command fetches a branch from a remote, both of which are read from the minibuffer.

f r   (`magit-fetch-refspec`)
>   This command fetches from a remote using an explicit refspec, both of which
>   are read from the minibuffer.

f a   (`magit-fetch-all`)
>   This command fetches from all remotes.

f m   (`magit-submodule-fetch`)
>   This command fetches all submodules. With a prefix argument it fetches all
>   remotes of all submodules.

`magit-pull-or-fetch`   [User Option]
>   By default fetch and pull commands are available from separate transient prefix com-
>   mand. Setting this to `t` adds some (but not all) of the above suffix commands to the
>   `magit-pull` transient.
>
>   If you do that, then you might also want to change the key binding for these prefix
>   commands, e.g.:
>
> ```
> (setq magit-pull-or-fetch t)
> (define-key magit-mode-map "f" 'magit-pull) ; was magit-fetch
> (define-key magit-mode-map "F" nil)         ; was magit-pull
> ```

7.3 Pulling

Also see the git-pull(1) manpage. For information about the upstream and the push-remote,
see Section 6.6.1 [The Two Remotes], page 71.

F   (`magit-pull`)
>   This transient prefix command binds the following suffix commands and dis-
>   plays them in a temporary buffer until a suffix is invoked.

F p   (`magit-pull-from-pushremote`)
>   This command pulls from the push-remote of the current branch.
>
>   When `magit-remote-set-if-missing` is non-nil and the push-remote is not
>   configured, then configure it before pulling. With a prefix argument let the user
>   change the push-remote before pulling from it.

F u   (`magit-pull-from-upstream`)
>   This command pulls from the upstream of the current branch.
>
>   When `magit-remote-set-if-missing` is non-nil and the upstream is not con-
>   figured, then configure it before pulling. With a prefix argument let the user
>   change the upstream before pulling from it.

F e   (`magit-pull-branch`)
>   This command pulls from a branch read in the minibuffer.

7.4 Pushing

Also see the git-push(1) manpage. For information about the upstream and the push-
remote, see Section 6.6.1 [The Two Remotes], page 71.

P   (`magit-push`)
>   This transient prefix command binds the following suffix commands along with
>   the appropriate infix arguments and displays them in a temporary buffer until
>   a suffix is invoked.

P p   (`magit-push-current-to-pushremote`)
>   This command pushes the current branch to its push-remote.
>
>   When `magit-remote-set-if-missing` is non-nil and the push-remote is not
>   configured, then configure it before pushing. With a prefix argument let the
>   user change the push-remote before pushing to it.

P u   (`magit-push-current-to-upstream`)
>   This command pushes the current branch to its upstream branch.
>
>   When `magit-remote-set-if-missing` is non-nil and the upstream is not con-
>   figured, then configure it before pushing. With a prefix argument let the user
>   change the upstream before pushing to it.

P e   (`magit-push-current`)
>   This command pushes the current branch to a branch read in the minibuffer.

P o   (`magit-push-other`)
>   This command pushes an arbitrary branch or commit somewhere. Both the
>   source and the target are read in the minibuffer.

P r   (`magit-push-refspecs`)
>   This command pushes one or multiple refspecs to a remote, both of which are
>   read in the minibuffer.
>
>   To use multiple refspecs, separate them with commas. Completion is only
>   available for the part before the colon, or when no colon is used.

P m   (`magit-push-matching`)
>   This command pushes all matching branches to another repository.
>
>   If only one remote exists, then push to that. Otherwise prompt for a remote,
>   offering the remote configured for the current branch as default.

P t   (`magit-push-tags`)
>   This command pushes all tags to another repository.
>
>   If only one remote exists, then push to that. Otherwise prompt for a remote,
>   offering the remote configured for the current branch as default.

P T   (`magit-push-tag`)
>   This command pushes a tag to another repository.

   Two more push commands exist, which by default are not available from the push
transient. See their doc-strings for instructions on how to add them to the transient.

`magit-push-implicitly` *args*   [Command]
>   This command pushes somewhere without using an explicit refspec.
>
>   This command simply runs `git push -v [ARGS]`. ARGS are the infix arguments. No
>   explicit refspec arguments are used. Instead the behavior depends on at least these
>   Git variables: `push.default`, `remote.pushDefault`, `branch.<branch>.pushRemote`,
>   `branch.<branch>.remote`, `branch.<branch>.merge`, and `remote.<remote>.push`.

`magit-push-to-remote` *remote args*   [Command]
>   This command pushes to the remote REMOTE without using an explicit refspec.
>   The remote is read in the minibuffer.
>
>   This command simply runs `git push -v [ARGS] REMOTE`. ARGS are the infix argu-
>   ments. No refspec arguments are used. Instead the behavior depends on at least these
>   Git variables: `push.default`, `remote.pushDefault`, `branch.<branch>.pushRemote`,
>   `branch.<branch>.remote`, `branch.<branch>.merge`, and `remote.<remote>.push`.

7.5 Plain Patches

`W`   (`magit-patch`)
>   This transient prefix command binds the following suffix commands along with
>   the appropriate infix arguments and displays them in a temporary buffer until
>   a suffix is invoked.

`W c`   (`magit-patch-create`)
>   This command creates patches for a set commits. If the region marks several
>   commits, then it creates patches for all of them. Otherwise it functions as a
>   transient prefix command, which features several infix arguments and binds
>   itself as a suffix command. When this command is invoked as a suffix of itself,
>   then it creates a patch using the specified infix arguments.

`w a`   (`magit-patch-apply`)
>   This command applies a patch. This is a transient prefix command, which
>   features several infix arguments and binds itself as a suffix command. When
>   this command is invoked as a suffix of itself, then it applies a patch using the
>   specified infix arguments.

`W s`   (`magit-patch-save`)
>   This command creates a patch from the current diff.
>
>   Inside `magit-diff-mode` or `magit-revision-mode` buffers, `C-x C-w` is also
>   bound to this command.

It is also possible to save a plain patch file by using `C-x C-w` inside a `magit-diff-mode`
or `magit-revision-mode` buffer.

7.6 Maildir Patches

Also see the git-am(1) manpage. and the git-apply(1) manpage.

`w`   (`magit-am`)
>   This transient prefix command binds the following suffix commands along with
>   the appropriate infix arguments and displays them in a temporary buffer until
>   a suffix is invoked.

`w w`   (`magit-am-apply-patches`)
>   This command applies one or more patches. If the region marks files, then
>   those are applied as patches. Otherwise this command reads a file-name in the
>   minibuffer, defaulting to the file at point.

w m   (`magit-am-apply-maildir`)
          This command applies patches from a maildir.

w a   (`magit-patch-apply`)
          This command applies a plain patch. For a longer description see Section 7.5
          [Plain Patches], page 98. This command is only available from the `magit-am`
          transient for historic reasons.

When an "am" operation is in progress, then the transient instead features the following
suffix commands.

w w   (`magit-am-continue`)
          This command resumes the current patch applying sequence.

w s   (`magit-am-skip`)
          This command skips the stopped at patch during a patch applying sequence.

w a   (`magit-am-abort`)
          This command aborts the current patch applying sequence. This discards all
          changes made since the sequence started.

8 Miscellaneous

8.1 Tagging

Also see the git-tag(1) manpage.

t   (`magit-tag`)

>   This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

t t   (`magit-tag-create`)

>   This command creates a new tag with the given NAME at REV. With a prefix argument it creates an annotate tag.

t r   (`magit-tag-release`)

>   This commands creates an annotated release tag. It assumes that release tags match `magit-release-tag-regexp`.
>
>   First it prompts for the name of the new tag using the highest existing tag as initial input and leaving it to the user to increment the desired part of the version string.
>
>   Then it prompts for the message of the new tag. The proposed tag message is based on the message of the highest tag, provided that that contains the corresponding version string and substituting the new version string for that. Otherwise it proposes something like "Foo-Bar 1.2.3", given, for example, a TAG "v1.2.3" and a repository located at something like "/path/to/foo-bar".
>
>   Then it calls "git tag –annotate –sign -m MSG TAG" to create the tag, regardless of whether these arguments are enabled in the transient. Finally it shows the refs buffer to let the user quickly review the result.

t k   (`magit-tag-delete`)

>   This command deletes one or more tags. If the region marks multiple tags (and nothing else), then it offers to delete those. Otherwise, it prompts for a single tag to be deleted, defaulting to the tag at point.

t p   (`magit-tag-prune`)

>   This command offers to delete tags missing locally from REMOTE, and vice versa.

8.2 Notes

Also see the git-notes(1) manpage.

T   (`magit-notes`)

>   This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.

T T   (`magit-notes-edit`)

>   Edit the note attached to a commit, defaulting to the commit at point.

By   default   use   the   value   of   Git   variable   `core.notesRef`   or
"refs/notes/commits" if that is undefined.

T r   (`magit-notes-remove`)
Remove the note attached to a commit, defaulting to the commit at point.

By   default   use   the   value   of   Git   variable   `core.notesRef`   or
"refs/notes/commits" if that is undefined.

T p   (`magit-notes-prune`)
Remove notes about unreachable commits.

It is possible to merge one note ref into another. That may result in conflicts which have
to resolved in the temporary worktree ".git/$NOTES_MERGE_WORKTREE$".

T m   (`magit-notes-merge`)
Merge the notes of a ref read from the user into the current notes ref.
The current notes ref is the value of Git variable `core.notesRef` or
"refs/notes/commits" if that is undefined.

When a notes merge is in progress then the transient features the following suffix com-
mands, instead of those listed above.

T c   (`magit-notes-merge-commit`)
Commit the current notes ref merge, after manually resolving conflicts.

T a   (`magit-notes-merge-abort`)
Abort the current notes ref merge.

The following variables control what notes reference `magit-notes-*`, `git notes` and `git
show` act on and display. Both the local and global values are displayed and can be modified.

`core.notesRef`   [Variable]
This variable specifies the notes ref that is displayed by default and which commands
act on by default.

`notes.displayRef`   [Variable]
This variable specifies additional notes ref to be displayed in addition to the ref
specified by `core.notesRef`. It can have multiple values and may end with `*` to
display all refs in the `refs/notes/` namespace (or `**` if some names contain slashes).

8.3 Submodules

Also see the git-submodule(1) manpage.

8.3.1 Listing Submodules

The command `magit-list-submodules` displays a list of the current repository's submod-
ules in a separate buffer. It's also possible to display information about submodules directly
in the status buffer of the super-repository by adding `magit-insert-submodules` to the
hook `magit-status-sections-hook` as described in Section 5.1.3 [Status Module Sections],
page 36.

`magit-list-submodules`   [Command]
>   This command displays a list of the current repository's submodules in a separate
>   buffer.
>
>   It can be invoked by pressing `RET` on the section titled "Modules".

`magit-submodule-list-columns`   [User Option]
>   This option controls what columns are displayed by the command `magit-list-`
>   `submodules` and how they are displayed.
>
>   Each element has the form (`HEADER WIDTH FORMAT PROPS`).
>
>   HEADER is the string displayed in the header. WIDTH is the width of the column.
>   FORMAT is a function that is called with one argument, the repository identification
>   (usually its basename), and with `default-directory` bound to the toplevel of its
>   working tree. It has to return a string to be inserted or nil. PROPS is an alist that
>   supports the keys `:right-align` and `:pad-right`.

`magit-insert-submodules`   [Function]
>   Insert sections for all submodules. For each section insert the path, the branch, and
>   the output of `git describe --tags`, or, failing that, the abbreviated HEAD commit
>   hash.
>
>   Press `RET` on such a submodule section to show its own status buffer. Press `RET` on
>   the "Modules" section to display a list of submodules in a separate buffer. This shows
>   additional information not displayed in the super-repository's status buffer.

8.3.2 Submodule Transient

o   (`magit-submodule`)
>   This transient prefix command binds the following suffix commands along with
>   the appropriate infix arguments and displays them in a temporary buffer until
>   a suffix is invoked.

  Some of the below commands default to act on the modules that are selected using the
region. For brevity their description talk about "the selected modules", but if no modules
are selected, then they act on the current module instead, or if point isn't on a module,
then the read a single module to act on. With a prefix argument these commands ignore
the selection and the current module and instead act on all suitable modules.

o a   (`magit-submodule-add`)
>   This commands adds the repository at URL as a module. Optional PATH is
>   the path to the module relative to the root of the super-project. If it is nil then
>   the path is determined based on URL.

o r   (`magit-submodule-register`)
>   This command registers the selected modules by copying their urls from ".git-
>   modules" to "GIT_D$IR/config". These values can then be edited before run-
>   ning `magit-submodule-populate`. If you don't need to edit any urls, then use
>   the latter directly.

o p   (`magit-submodule-populate`)
>   This command creates the working directory or directories of the selected mod-
>   ules, checking out the recorded commits.

o u   (`magit-submodule-update`)
>   This command updates the selected modules checking out the recorded com-
>   mits.

o s   (`magit-submodule-synchronize`)
>   This command synchronizes the urls of the selected modules, copying the values
>   from ".gitmodules" to the ".git/config" of the super-project as well those of the
>   modules.

o d   (`magit-submodule-unpopulate`)
>   This command removes the working directory of the selected modules.

o l   (`magit-list-submodules`)
>   This command displays a list of the current repository's modules.

o f   (`magit-fetch-modules`)
>   This command fetches all modules.
>
>   Option `magit-fetch-modules-jobs` controls how many submodules are being
>   fetched in parallel. Also fetch the super-repository, because `git fetch` does not
>   support not doing that. With a prefix argument fetch all remotes.

8.4 Subtree

Also see the git-subtree(1) manpage.

O   (`magit-subtree`)
>   This transient prefix command binds the two sub-transients; one for importing
>   a subtree and one for exporting a subtree.

O i   (`magit-subtree-import`)
>   This transient prefix command binds the following suffix commands along with
>   the appropriate infix arguments and displays them in a temporary buffer until
>   a suffix is invoked.
>
>   The suffixes of this command import subtrees.
>
>   If the `--prefix` argument is set, then the suffix commands use that prefix
>   without prompting the user. If it is unset, then they read the prefix in the
>   minibuffer.

O i a (`magit-subtree-add`)
>   This command adds COMMIT from REPOSITORY as a new subtree at PRE-
>   FIX.

O i c (`magit-subtree-add-commit`)
>   This command add COMMIT as a new subtree at PREFIX.

O i m (`magit-subtree-merge`)
>   This command merges COMMIT into the PREFIX subtree.

O i f (`magit-subtree-pull`)
>   This command pulls COMMIT from REPOSITORY into the PREFIX subtree.

O e   (`magit-subtree-export`)

> This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.
>
> The suffixes of this command export subtrees.
>
> If the `--prefix` argument is set, then the suffix commands use that prefix without prompting the user. If it is unset, then they read the prefix in the minibuffer.

O e p   (`magit-subtree-push`)

> This command extract the history of the subtree PREFIX and pushes it to REF on REPOSITORY.

O e s   (`magit-subtree-split`)

> This command extracts the history of the subtree PREFIX.

8.5 Worktree

Also see the git-worktree(1) manpage.

%   (`magit-worktree`)

> This transient prefix command binds the following suffix commands and displays them in a temporary buffer until a suffix is invoked.

% b   (`magit-worktree-checkout`)

> Checkout BRANCH in a new worktree at PATH.

% c   (`magit-worktree-branch`)

> Create a new BRANCH and check it out in a new worktree at PATH.

% k   (`magit-worktree-delete`)

> Delete a worktree, defaulting to the worktree at point. The primary worktree cannot be deleted.

% g   (`magit-worktree-status`)

> Show the status for the worktree at point.
>
> If there is no worktree at point, then read one in the minibuffer. If the worktree at point is the one whose status is already being displayed in the current buffer, then show it in Dired instead.

8.6 Common Commands

These are some of the commands that can be used in all buffers whose major-modes derive from `magit-mode`. There are other common commands beside the ones below, but these didn't fit well anywhere else.

M-w   (`magit-copy-section-value`)

> This command saves the value of the current section to the `kill-ring`, and, provided that the current section is a commit, branch, or tag section, it also pushes the (referenced) revision to the `magit-revision-stack`.

> When the current section is a branch or a tag, and a prefix argument is used, then it saves the revision at its tip to the `kill-ring` instead of the reference name.
>
> When the region is active, this command saves that to the `kill-ring`, like `kill-ring-save` would, instead of behaving as described above. If a prefix argument is used and the region is within a hunk, it strips the outer diff marker column before saving the text.

`C-w`   (`magit-copy-buffer-revision`)

> This command saves the revision being displayed in the current buffer to the `kill-ring` and also pushes it to the `magit-revision-stack`. It is mainly intended for use in `magit-revision-mode` buffers, the only buffers where it is always unambiguous exactly which revision should be saved.
>
> Most other Magit buffers usually show more than one revision, in some way or another, so this command has to select one of them, and that choice might not always be the one you think would have been the best pick.

Outside of Magit `M-w` and `C-w` are usually bound to `kill-ring-save` and `kill-region`, and these commands would also be useful in Magit buffers. Therefore when the region is active, then both of these commands behave like `kill-ring-save` instead of as described above.

8.7 Wip Modes

Git keeps **committed** changes around long enough for users to recover changes they have accidentally deleted. It does so by not garbage collecting any committed but no longer referenced objects for a certain period of time, by default 30 days.

But Git does **not** keep track of **uncommitted** changes in the working tree and not even the index (the staging area). Because Magit makes it so convenient to modify uncommitted changes, it also makes it easy to shoot yourself in the foot in the process.

For that reason Magit provides a global mode that saves **tracked** files to work-in-progress references after or before certain actions. (At present untracked files are never saved and for technical reasons nothing is saved before the first commit has been created).

Two separate work-in-progress references are used to track the state of the index and of the working tree: `refs/wip/index/<branchref>` and `refs/wip/wtree/<branchref>`, where `<branchref>` is the full ref of the current branch, e.g. `refs/heads/master`. When the `HEAD` is detached then `HEAD` is used in place of `<branchref>`.

Checking out another branch (or detaching `HEAD`) causes the use of different wip refs for subsequent changes.

`magit-wip-mode`   [User Option]

> When this mode is enabled, then uncommitted changes are committed to dedicated work-in-progress refs whenever appropriate (i.e. when dataloss would be a possibility otherwise).
>
> Setting this variable directly does not take effect; either use the Custom interface to do so or call the respective mode function.

For historic reasons this mode is implemented on top of four other `magit-wip-*`
modes, which can also be used individually, if you want finer control over when the
wip refs are updated; but that is discouraged. See Section 8.7.2 [Legacy Wip Modes],
page 108.

To view the log for a branch and its wip refs use the commands `magit-wip-log` and
`magit-wip-log-current`. You should use `--graph` when using these commands.

`magit-wip-log`   [Command]
This command shows the log for a branch and its wip refs. With a negative prefix
argument only the worktree wip ref is shown.

The absolute numeric value of the prefix argument controls how many "branches" of
each wip ref are shown. This is only relevant if the value of `magit-wip-merge-branch`
is `nil`.

`magit-wip-log-current`   [Command]
This command shows the log for the current branch and its wip refs. With a negative
prefix argument only the worktree wip ref is shown.

The absolute numeric value of the prefix argument controls how many "branches" of
each wip ref are shown. This is only relevant if the value of `magit-wip-merge-branch`
is `nil`.

`X w`   (`magit-reset-worktree`)
This command resets the working tree to some commit read from the user and
defaulting to the commit at point, while keeping the `HEAD` and index as-is.

This can be used to restore files to the state committed to a wip ref. Note that
this will discard any unstaged changes that might have existed before invoking
this command (but of course only after committing that to the working tree
wip ref).

Note that even if you enable `magit-wip-mode` this won't give you perfect protection.
The most likely scenario for losing changes despite the use of `magit-wip-mode` is making a
change outside Emacs and then destroying it also outside Emacs. In some such a scenario,
Magit, being an Emacs package, didn't get the opportunity to keep you from shooting
yourself in the foot.

When you are unsure whether Magit did commit a change to the wip refs, then you can
explicitly request that all changes to all tracked files are being committed.

`M-x magit-wip-commit`   (`magit-wip-commit`)
This command commits all changes to all tracked files to the index and working
tree work-in-progress refs. Like the modes described above, it does not commit
untracked files, but it does check all tracked files for changes. Use this command
when you suspect that the modes might have overlooked a change made outside
Emacs/Magit.

`magit-wip-namespace`   [User Option]
The namespace used for work-in-progress refs. It has to end with a slash. The wip refs
are named `<namespace>index/<branchref>` and `<namespace>wtree/<branchref>`.
When snapshots are created while the `HEAD` is detached then `HEAD` is used in place of
`<branchref>`.

`magit-wip-mode-lighter`   [User Option]
    Mode-line lighter for `magit-wip--mode`.

8.7.1 Wip Graph

`magit-wip-merge-branch`   [User Option]
    This option controls whether the current branch is merged into the wip refs after a
    new commit was created on the branch.

    If non-nil and the current branch has new commits, then it is merged into the wip ref
    before creating a new wip commit. This makes it easier to inspect wip history and
    the wip commits are never garbage collected.

    If nil and the current branch has new commits, then the wip ref is reset to the tip
    of the branch before creating a new wip commit. With this setting wip commits are
    eventually garbage collected.

    When `magit-wip-merge-branch` is `t`, then the history looks like this:

```
  *--*--*--*--*--*        refs/wip/index/refs/heads/master
 /     /     /
A-----B-----C             refs/heads/master
```

When `magit-wip-merge-branch` is `nil`, then creating a commit on the real branch and
then making a change causes the wip refs to be recreated to fork from the new commit. But
the old commits on the wip refs are not lost. They are still available from the reflog. To
make it easier to see when the fork point of a wip ref was changed, an additional commit with
the message "restart autosaving" is created on it (`xx0` commits below are such boundary
commits).

    Starting with

```
      BI0---BI1     refs/wip/index/refs/heads/master
     /
  A---B             refs/heads/master
     \
      BW0---BW1     refs/wip/wtree/refs/heads/master
```

and committing the staged changes and editing and saving a file would result in

```
      BI0---BI1          refs/wip/index/refs/heads/master
     /
  A---B---C              refs/heads/master
     \   \
      \   CW0---CW1      refs/wip/wtree/refs/heads/master
       \
        BW0---BW1        refs/wip/wtree/refs/heads/master@{2}
```

    The fork-point of the index wip ref is not changed until some change is being staged.
Likewise just checking out a branch or creating a commit does not change the fork-point of
the working tree wip ref. The fork-points are not adjusted until there actually is a change
that should be committed to the respective wip ref.

8.7.2 Legacy Wip Modes

It is recommended that you use the mode `magit-wip-mode` (which see) and ignore the existence of the following modes, which are preserved for historic reasons.

Setting the following variables directly does not take effect; either use the Custom interface to do so or call the respective mode functions.

`magit-wip-after-save-mode`   [User Option]
When this mode is enabled, then saving a buffer that visits a file tracked in a Git repository causes its current state to be committed to the working tree wip ref for the current branch.

`magit-wip-after-apply-mode`   [User Option]
When this mode is enabled, then applying (i.e. staging, unstaging, discarding, reversing, and regularly applying) a change to a file tracked in a Git repository causes its current state to be committed to the index and/or working tree wip refs for the current branch.

If you only ever edit files using Emacs and only ever interact with Git using Magit, then the above two modes should be enough to protect each and every change from accidental loss. In practice nobody does that. Two additional modes exists that do commit to the wip refs before making changes that could cause the loss of earlier changes.

`magit-wip-before-change-mode`   [User Option]
When this mode is enabled, then certain commands commit the existing changes to the files they are about to make changes to.

`magit-wip-initial-backup-mode`   [User Option]
When this mode is enabled, then the current version of a file is committed to the worktree wip ref before the buffer visiting that file is saved for the first time since the buffer was created.

This backs up the same version of the file that `backup-buffer` would save. While `backup-buffer` uses a backup file, this mode uses the same worktree wip ref as used by the other Magit Wip modes. Like `backup-buffer`, it only does this once; unless you kill the buffer and visit the file again only one backup will be created per Emacs session.

This mode ignores the variables that affect `backup-buffer` and can be used along-side that function, which is recommended because it only backs up files that are tracked in a Git repository.

`magit-wip-after-save-local-mode-lighter`   [User Option]
Mode-line lighter for `magit-wip-after-save-local-mode`.

`magit-wip-after-apply-mode-lighter`   [User Option]
Mode-line lighter for `magit-wip-after-apply-mode`.

`magit-wip-before-change-mode-lighter`   [User Option]
Mode-line lighter for `magit-wip-before-change-mode`.

`magit-wip-initial-backup-mode-lighter`   [User Option]
Mode-line lighter for `magit-wip-initial-backup-mode`.

8.8 Minor Mode for Buffers Visiting Files

The minor-mode `magit-file-mode` enables certain Magit features in file-visiting buffers belonging to a Git repository. The globalized variant `global-magit-file-mode` enables the local mode in all such buffers. It is enabled by default. Currently the local mode only establishes a few key bindings, but this might be extended in the future.

`global-magit-file-mode`   [User Option]
>   Whether to establish certain Magit key bindings in all file-visiting buffers belonging to any Git repository. This is enabled by default. This globalized mode turns on the local minor-mode `magit-file-mode` in all suitable buffers.

`magit-file-mode-map`   [Variable]
>   This keymap is used by the local minor-mode `magit-file-mode` and establishes the key bindings described below.
>
>   Note that the default binding for `magit-file-dispatch` is very cumbersome to use and that we recommend that you add a better binding.
>
>   Instead of `C-c M-g` I would have preferred to use `C-c g` because (1) it is similar to `C-x g` (the recommended global binding for `~magit-status`), (2) we cannot use `C-c C-g` because we have been recommending that that be bound to `magit-dispatch` for a long time, (3) we cannot use `C-x C-g` because that is a convenient way of aborting the incomplete key sequence `C-x`, and most importantly (4) it would make it much easier to type the next key (a suffix binding) because most of those are letters.
>
>   For example `C-c g b` is much easier to type than `C-c M-g b`. For suffix bindings that use uppercase letters, the default is just horrible—having to use e.g. `C-c M-g B` (`Control+c Meta+g Shift+b`) would drive anyone up the walls (or to Vim).
>
>   However `C-c LETTER` bindings are reserved for users (see Section "Key Binding Conventions" in `elisp`). Packages are forbidden from using those. Doing so anyway is considered heresy. Therefore if you want a better binding, you have to add it yourself:
>
>    ```
>    (define-key magit-file-mode-map
>      (kbd "C-c g") 'magit-file-dispatch)
>    ```

The key bindings shown below assume that you have not improved the binding for `magit-file-dispatch`.

C-c M-g   (`magit-file-dispatch`)
>   This transient prefix command binds the following suffix commands and displays them in a temporary buffer until a suffix is invoked.

C-c M-g s   (`magit-stage-file`)
>   Stage all changes to the file being visited in the current buffer.

C-c M-g u   (`magit-unstage-file`)
>   Unstage all changes to the file being visited in the current buffer.

C-c M-g c   (`magit-commit`)
>   This transient prefix command binds the following suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked. See Section 6.5.1 [Initiating a Commit], page 66.

`C-c M-g D`   (`magit-diff`)

> This transient prefix command binds several diff suffix commands and infix arguments and displays them in a temporary buffer until a suffix is invoked. See Section 5.4 [Diffing], page 44.
>
> This is the same command that `d` is bound to in Magit buffers. If this command is invoked from a file-visiting buffer, then the initial value of the option (`--`) that limits the diff to certain file(s) is set to the visited file.

`C-c M-g d`   (`magit-diff-buffer-file`)

> This command shows the diff for the file of blob that the current buffer visits.

`magit-diff-buffer-file-locked`   [User Option]

> This option controls whether `magit-diff-buffer-file` uses a dedicated buffer. See Section 4.1 [Modes and Buffers], page 7.

`C-c M-g L`   (`magit-log`)

> This transient prefix command binds several log suffix commands and infix arguments and displays them in a temporary buffer until a suffix is invoked. See Section 5.3 [Logging], page 38.
>
> This is the same command that `l` is bound to in Magit buffers. If this command is invoked from a file-visiting buffer, then the initial value of the option (`--`) that limits the log to certain file(s) is set to the visited file.

`C-c M-g l`   (`magit-log-buffer-file`)

> This command shows the log for the file of blob that the current buffer visits. Renames are followed when a prefix argument is used or when `--follow` is an active log argument. When the region is active, the log is restricted to the selected line range.

`C-c M-g t`   (`magit-log-trace-definition`)

> This command shows the log for the definition at point.

`magit-log-buffer-file-locked`   [User Option]

> This option controls whether `magit-log-buffer-file` uses a dedicated buffer. See Section 4.1 [Modes and Buffers], page 7.

`C-c M-g B`   (`magit-blame`)

> This transient prefix command binds all blaming suffix commands along with the appropriate infix arguments and displays them in a temporary buffer until a suffix is invoked.
>
> For more information about this and the following commands also see Section 5.9 [Blaming], page 58.

In addition to the `magit-blame` sub-transient, the dispatch transient also binds several blaming suffix commands directly. See Section 5.9 [Blaming], page 58, for information about those commands and bindings.

`C-c M-g e`   (`magit-edit-line-commit`)

> This command makes the commit editable that added the current line.
>
> With a prefix argument it makes the commit editable that removes the line, if any. The commit is determined using `git blame` and made editable using

> `git rebase --interactive` if it is reachable from `HEAD`, or by checking out the commit (or a branch that points at it) otherwise.

`C-c M-g p`   (`magit-blob-previous`)
>   Visit the previous blob which modified the current file.

There are a few additional commands that operate on a single file but are not enabled in the file transient command by default:

`magit-file-rename`   [Command]
>   This command renames a file read from the user.

`magit-file-delete`   [Command]
>   This command deletes a file read from the user.

`magit-file-untrack`   [Command]
>   This command untracks a file read from the user.

`magit-file-checkout`   [Command]
>   This command updates a file in the working tree and index to the contents from a revision. Both the revision and file are read from the user.

To enable them invoke the transient (`C-c M-g`), enter "edit mode" (`C-x l`), set the "transient level" (`C-x l` again), enter `5`, and leave edit mode (`C-g`). Also see Section "Enabling and Disabling Suffixes" in `transient`.

8.9 Minor Mode for Buffers Visiting Blobs

The `magit-blob-mode` enables certain Magit features in blob-visiting buffers. Such buffers can be created using `magit-find-file` and some of the commands mentioned below, which also take care of turning on this minor mode. Currently this mode only establishes a few key bindings, but this might be extended.

`p`   (`magit-blob-previous`)
>   Visit the previous blob which modified the current file.

`n`   (`magit-blob-next`)
>   Visit the next blob which modified the current file.

`q`   (`magit-kill-this-buffer`)
>   Kill the current buffer.

9 Customizing

Both Git and Emacs are highly customizable. Magit is both a Git porcelain as well as an Emacs package, so it makes sense to customize it using both Git variables as well as Emacs options. However this flexibility doesn't come without problems, including but not limited to the following.

- Some Git variables automatically have an effect in Magit without requiring any explicit support. Sometimes that is desirable - in other cases, it breaks Magit.

  When a certain Git setting breaks Magit but you want to keep using that setting on the command line, then that can be accomplished by overriding the value for Magit only by appending something like (`"-c"` `"some.variable=compatible-value"`) to `magit-git-global-arguments`.

- Certain settings like `fetch.prune=true` are respected by Magit commands (because they simply call the respective Git command) but their value is not reflected in the respective transient buffers. In this case the `--prune` argument in `magit-fetch` might be active or inactive, but that doesn't keep the Git variable from being honored by the suffix commands anyway. So pruning might happen despite the `--prune` arguments being displayed in a way that seems to indicate that no pruning will happen.

I intend to address these and similar issues in a future release.

9.1 Per-Repository Configuration

Magit can be configured on a per-repository level using both Git variables as well as Emacs options.

To set a Git variable for one repository only, simply set it in `/path/to/repo/.git/config` instead of `$HOME/.gitconfig` or `/etc/gitconfig`. See the git-config(1) manpage.

Similarly, Emacs options can be set for one repository only by editing `/path/to/repo/.dir-locals.el`. See Section "Directory Variables" in `emacs`. For example to disable automatic refreshes of file-visiting buffers in just one huge repository use this:

- `/path/to/huge/repo/.dir-locals.el`

      ((nil . ((magit-refresh-buffers . nil))))

If you want to apply the same settings to several, but not all, repositories then keeping the repository-local config files in sync would quickly become annoying. To avoid that you can create config files for certain classes of repositories (e.g. "huge repositories") and then include those files in the per-repository config files. For example:

- `/path/to/huge/repo/.git/config`

      [include]
              path = /path/to/huge-gitconfig

- `/path/to/huge-gitconfig`

      [status]
              showUntrackedFiles = no

- `$HOME/.emacs.d/init.el`

      (dir-locals-set-class-variables 'huge-git-repository
         '((nil . ((magit-refresh-buffers . nil)))))

      (dir-locals-set-directory-class
         "/path/to/huge/repo/" 'huge-git-repository)

9.2 Essential Settings

The next two sections list and discuss several variables that many users might want to customize, for safety and/or performance reasons.

9.2.1 Safety

This section discusses various variables that you might want to change (or **not** change) for safety reasons.

Git keeps **committed** changes around long enough for users to recover changes they have accidentally been deleted. It does not do the same for **uncommitted** changes in the working tree and not even the index (the staging area). Because Magit makes it so easy to modify uncommitted changes, it also makes it easy to shoot yourself in the foot in the process. For that reason Magit provides three global modes that save **tracked** files to work-in-progress references after or before certain actions. See Section 8.7 [Wip Modes], page 105.

These modes are not enabled by default because of performance concerns. Instead a lot of potentially destructive commands require confirmation every time they are used. In many cases this can be disabled by adding a symbol to `magit-no-confirm` (see Section 4.5.2 [Completion and Confirmation], page 24). If you enable the various wip modes then you should add `safe-with-wip` to this list.

Similarly it isn't necessary to require confirmation before moving a file to the system trash - if you trashed a file by mistake then you can recover it from the there. Option `magit-delete-by-moving-to-trash` controls whether the system trash is used, which is the case by default. Nevertheless, `trash` isn't a member of `magit-no-confirm` - you might want to change that.

By default buffers visiting files are automatically reverted when the visited file changes on disk. This isn't as risky as it might seem, but to make an informed decision you should see [Risk of Reverting Automatically], page 13.

9.2.2 Performance

After Magit has run `git` for side-effects, it also refreshes the current Magit buffer and the respective status buffer. This is necessary because otherwise outdated information might be displayed without the user noticing. Magit buffers are updated by recreating their content from scratch, which makes updating simpler and less error-prone, but also more costly. Keeping it simple and just re-creating everything from scratch is an old design decision and departing from that will require major refactoring.

I plan to do that in time for the next major release. I also intend to create logs and diffs asynchronously, which should also help a lot but also requires major refactoring.

Meanwhile you can tell Magit to only automatically refresh the current Magit buffer, but not the status buffer. If you do that, then the status buffer is only refreshed automatically if it is the current buffer.

```
(setq magit-refresh-status-buffer nil)
```

You should also check whether any third-party packages have added anything to `magit-refresh-buffer-hook`, `magit-status-refresh-hook`, `magit-pre-refresh-hook`, and `magit-post-refresh-hook`. If so, then check whether those additions impact performance significantly. Setting `magit-refresh-verbose` and then inspecting the output in the `*Messages*` buffer, should help doing so.

Magit also reverts buffers for visited files located inside the current repository when the visited file changes on disk. That is implemented on top of `auto-revert-mode` from the built-in library `autorevert`. To figure out whether that impacts performance, check whether performance is significantly worse, when many buffers exist and/or when some buffers visit files using TRAMP. If so, then this should help.

```
(setq auto-revert-buffer-list-filter
      'magit-auto-revert-repository-buffer-p)
```

For alternative approaches see Section 4.1.6 [Automatic Reverting of File-Visiting Buffers], page 12.

If you have enabled any features that are disabled by default, then you should check whether they impact performance significantly. It's likely that they were not enabled by default because it is known that they reduce performance at least in large repositories.

If performance is only slow inside certain unusually large repositories, then you might want to disable certain features on a per-repository or per-repository-class basis only. See Section 9.1 [Per-Repository Configuration], page 112.

Log Performance

When showing logs, Magit limits the number of commits initially shown in the hope that this avoids unnecessary work. When using `--graph` is used, then this unfortunately does not have the desired effect for large histories. Junio, Git's maintainer, said on the git mailing list (http://www.spinics.net/lists/git/msg232230.html): "`--graph` wants to compute the whole history and the max-count only affects the output phase after `--graph` does its computation".

In other words, it's not that Git is slow at outputting the differences, or that Magit is slow at parsing the output - the problem is that Git first goes outside and has a smoke.

We actually work around this issue by limiting the number of commits not only by using `-<N>` but by also using a range. But unfortunately that's not always possible.

When more than a few thousand commits are shown, then the use of `--graph` can slow things down.

Using `--color --graph` is even slower. Magit uses code that is part of Emacs to turn control characters into faces. That code is pretty slow and this is quite noticeable when showing a log with many branches and merges. For that reason `--color` is not enabled by default anymore. Consider leaving it at that.

Diff Performance

If diffs are slow, then consider turning off some optional diff features by setting all or some of the following variables to `nil`: `magit-diff-highlight-indentation`, `magit-diff-highlight-trailing`, `magit-diff-paint-whitespace`, `magit-diff-highlight-hunk-body`, and `magit-diff-refine-hunk`.

When showing a commit instead of some arbitrary diff, then some additional information is displayed. Calculating this information can be quite expensive given certain circumstances. If looking at a commit using `magit-revision-mode` takes considerably more time than looking at the same commit in `magit-diff-mode`, then consider setting `magit-revision-insert-related-refs` to `nil`.

Refs Buffer Performance

When refreshing the "references buffer" is slow, then that's usually because several hundred refs are being displayed. The best way to address that is to display fewer refs, obviously.

If you are not, or only mildly, interested in seeing the list of tags, then start by not displaying them:

```
(remove-hook 'magit-refs-sections-hook 'magit-insert-tags)
```

Then you should also make sure that the listed remote branches actually all exist. You can do so by pruning branches which no longer exist using `f-pa`.

Committing Performance

When you initiate a commit, then Magit by default automatically shows a diff of the changes you are about to commit. For large commits this can take a long time, which is especially distracting when you are committing large amounts of generated data which you don't actually intend to inspect before committing. This behavior can be turned off using:

```
(remove-hook 'server-switch-hook 'magit-commit-diff)
```

Then you can type `C-c C-d` to show the diff when you actually want to see it, but only then. Alternatively you can leave the hook alone and just type `C-g` in those cases when it takes too long to generate the diff. If you do that, then you will end up with a broken diff buffer, but doing it this way has the advantage that you usually get to see the diff, which is useful because it increases the odds that you spot potential issues.

The Built-In VC Package

Emacs comes with a version control interface called "VC", see Section "Version Control" in `emacs`. It is enabled be default, and if you don't use it in addition to Magit, then you should disable it to keep it from performing unnecessary work:

```
(setq vc-handled-backends nil)
```

You can also disable its use for Git but keep using it when using another version control system:

```
(setq vc-handled-backends (delq 'Git vc-handled-backends))
```

Microsoft Windows Performance

In order to update the status buffer, `git` has to be run a few dozen times. That is problematic on Microsoft Windows, because that operating system is exceptionally slow at starting

processes. Sadly this is an issue that can only be fixed by Microsoft itself, and they don't appear to be particularly interested in doing so.

Beside the subprocess issue, there are also other Windows-specific performance issues. Some of these have workarounds. The maintainers of "Git for Windows" try to improve performance on Windows. Always use the latest release in order to benefit from the latest performance tweaks. Magit too tries to work around some Windows-specific issues.

According to some sources, setting the following Git variables can also help.

```
git config --global core.preloadindex true   # default since v2.1
git config --global core.fscache true         # default since v2.8
git config --global gc.auto 256
```

You should also check whether an anti-virus program is affecting performance.

MacOS Performance

Before Emacs 26.1 child processes were created using `fork` on macOS. That needlessly copied GUI resources, which is expensive. The result was that forking took about 30 times as long on Darwin than on Linux, and because Magit starts many `git` processes that made quite a difference.

So make sure that you are using at least Emacs 26.1, in which case the faster `vfork` will be used. (The creation of child processes still takes about twice as long on Darwin compared to Linux.) See[1] for more information.

---

[1] `https://lists.gnu.org/archive/html/bug-gnu-emacs/2017-04/msg00201.html`

10 Plumbing

The following sections describe how to use several of Magit's core abstractions to extend Magit itself or implement a separate extension.

A few of the low-level features used by Magit have been factored out into separate libraries/packages, so that they can be used by other packages, without having to depend on Magit. See `with-editor` for information about `with-editor`. `transient` doesn't have a manual yet.

If you are trying to find an unused key that you can bind to a command provided by your own Magit extension, then checkout `https://github.com/magit/magit/wiki/Plugin-Dispatch-Key-Registry`.

10.1 Calling Git

Magit provides many specialized functions for calling Git. All of these functions are defined in either `magit-git.el` or `magit-process.el` and have one of the prefixes `magit-run-`, `magit-call-`, `magit-start-`, or `magit-git-` (which is also used for other things).

All of these functions accept an indefinite number of arguments, which are strings that specify command line arguments for Git (or in some cases an arbitrary executable). These arguments are flattened before being passed on to the executable; so instead of strings they can also be lists of strings and arguments that are `nil` are silently dropped. Some of these functions also require a single mandatory argument before these command line arguments.

Roughly speaking, these functions run Git either to get some value or for side-effects. The functions that return a value are useful to collect the information necessary to populate a Magit buffer, while the others are used to implement Magit commands.

The functions in the value-only group always run synchronously, and they never trigger a refresh. The function in the side-effect group can be further divided into subgroups depending on whether they run Git synchronously or asynchronously, and depending on whether they trigger a refresh when the executable has finished.

10.1.1 Getting a Value from Git

These functions run Git in order to get a value, an exit status, or output. Of course you could also use them to run Git commands that have side-effects, but that should be avoided.

`magit-git-exit-code` **&rest** *args*   [Function]
>   Executes git with ARGS and returns its exit code.

`magit-git-success` **&rest** *args*   [Function]
>   Executes git with ARGS and returns `t` if the exit code is `0`, `nil` otherwise.

`magit-git-failure` **&rest** *args*   [Function]
>   Executes git with ARGS and returns `t` if the exit code is `1`, `nil` otherwise.

`magit-git-true` **&rest** *args*   [Function]
>   Executes git with ARGS and returns `t` if the first line printed by git is the string
>   `"true"`, `nil` otherwise.

`magit-git-false` **&rest** *args*   [Function]
>   Executes git with ARGS and returns `t` if the first line printed by git is the string
>   `"false"`, `nil` otherwise.

`magit-git-insert` **&rest** *args*   [Function]
>   Executes git with ARGS and inserts its output at point.

`magit-git-string` **&rest** *args*   [Function]
>   Executes git with ARGS and returns the first line of its output. If there is no output
>   or if it begins with a newline character, then this returns `nil`.

`magit-git-lines` **&rest** *args*   [Function]
>   Executes git with ARGS and returns its output as a list of lines. Empty lines anywhere
>   in the output are omitted.

`magit-git-items` **&rest** *args*   [Function]
>   Executes git with ARGS and returns its null-separated output as a list. Empty items
>   anywhere in the output are omitted.
>
>   If the value of option `magit-git-debug` is non-nil and git exits with a non-zero exit
>   status, then warn about that in the echo area and add a section containing git's
>   standard error in the current repository's process buffer.

If an error occurs when using one of the above functions, then that is usually due to a
bug, i.e. using an argument which is not actually supported. Such errors are usually not
reported, but when they occur we need to be able to debug them.

`magit-git-debug`   [User Option]
>   Whether to report errors that occur when using `magit-git-insert`, `magit-git-`
>   `string`, `magit-git-lines`, or `magit-git-items`. This does not actually raise an
>   error. Instead a message is shown in the echo area, and git's standard error is insert
>   into a new section in the current repository's process buffer.

`magit-git-str` **&rest** *args*   [Function]
>   This is a variant of `magit-git-string` that ignores the option `magit-git-debug`. It
>   is mainly intended to be used while handling errors in functions that do respect that
>   option. Using such a function while handing an error could cause yet another error
>   and therefore lead to an infinite recursion. You probably won't ever need to use this
>   function.

10.1.2 Calling Git for Effect

These functions are used to run git to produce some effect. Most Magit commands that
actually run git do so by using such a function.

Because we do not need to consume git's output when using these functions, their output
is instead logged into a per-repository buffer, which can be shown using `$` from a Magit
buffer or `M-x magit-process` elsewhere.

These functions can have an effect in two distinct ways. Firstly, running git may change
something, i.e. create or push a new commit. Secondly, that change may require that
Magit buffers are refreshed to reflect the changed state of the repository. But refreshing

isn't always desirable, so only some of these functions do perform such a refresh after git has returned.

Sometimes it is useful to run git asynchronously. For example, when the user has just initiated a push, then there is no reason to make her wait until that has completed. In other cases it makes sense to wait for git to complete before letting the user do something else. For example after staging a change it is useful to wait until after the refresh because that also automatically moves to the next change.

magit-call-git &rest *args*   [Function]
> Calls git synchronously with ARGS.

magit-call-process *program* &rest *args*   [Function]
> Calls PROGRAM synchronously with ARGS.

magit-run-git &rest *args*   [Function]
> Calls git synchronously with ARGS and then refreshes.

magit-run-git-with-input *input* &rest *args*   [Function]
> Calls git synchronously with ARGS and sends it INPUT on standard input.
>
> INPUT should be a buffer or the name of an existing buffer. The content of that buffer is used as the process' standard input. After the process returns a refresh is performed.
>
> As a special case, INPUT may also be nil. In that case the content of the current buffer is used as standard input and **no** refresh is performed.
>
> This function actually runs git asynchronously. But then it waits for the process to return, so the function itself is synchronous.

magit-run-git-with-logfile *file* &rest *args*   [Function]
> Calls git synchronously with ARGS. The process' output is saved in FILE. This is rarely useful and so this function might be removed in the future.
>
> This function actually runs git asynchronously. But then it waits for the process to return, so the function itself is synchronous.

magit-git &rest *args*   [Function]
> Calls git synchronously with ARGS for side-effects only. This function does not refresh the buffer.

magit-git-wash *washer* &rest *args*   [Function]
> Execute Git with ARGS, inserting washed output at point. Actually first insert the raw output at point. If there is no output call `magit-cancel-section`. Otherwise temporarily narrow the buffer to the inserted text, move to its beginning, and then call function WASHER with ARGS as its sole argument.

And now for the asynchronous variants.

magit-run-git-async &rest *args*   [Function]
> Start Git, prepare for refresh, and return the process object. ARGS is flattened and then used as arguments to Git.
>
> Display the command line arguments in the echo area.

After Git returns some buffers are refreshed: the buffer that was current when this function was called (if it is a Magit buffer and still alive), as well as the respective Magit status buffer. Unmodified buffers visiting files that are tracked in the current repository are reverted if `magit-revert-buffers` is non-nil.

`magit-run-git-with-editor` **&rest** *args*   [Function]

Export GIT_EDITOR and start Git. Also prepare for refresh and return the process object. ARGS is flattened and then used as arguments to Git.

Display the command line arguments in the echo area.

After Git returns some buffers are refreshed: the buffer that was current when this function was called (if it is a Magit buffer and still alive), as well as the respective Magit status buffer.

`magit-start-git` **&rest** *args*   [Function]

Start Git, prepare for refresh, and return the process object.

If INPUT is non-nil, it has to be a buffer or the name of an existing buffer. The buffer content becomes the processes standard input.

Option `magit-git-executable` specifies the Git executable and option `magit-git-global-arguments` specifies constant arguments. The remaining arguments ARGS specify arguments to Git. They are flattened before use.

After Git returns, some buffers are refreshed: the buffer that was current when this function was called (if it is a Magit buffer and still alive), as well as the respective Magit status buffer. Unmodified buffers visiting files that are tracked in the current repository are reverted if `magit-revert-buffers` is non-nil.

`magit-start-process` **&rest** *args*   [Function]

Start PROGRAM, prepare for refresh, and return the process object.

If optional argument INPUT is non-nil, it has to be a buffer or the name of an existing buffer. The buffer content becomes the processes standard input.

The process is started using `start-file-process` and then setup to use the sentinel `magit-process-sentinel` and the filter `magit-process-filter`. Information required by these functions is stored in the process object. When this function returns the process has not started to run yet so it is possible to override the sentinel and filter.

After the process returns, `magit-process-sentinel` refreshes the buffer that was current when `magit-start-process` was called (if it is a Magit buffer and still alive), as well as the respective Magit status buffer. Unmodified buffers visiting files that are tracked in the current repository are reverted if `magit-revert-buffers` is non-nil.

`magit-this-process`   [Variable]

The child process which is about to start. This can be used to change the filter and sentinel.

`magit-process-raise-error`   [Variable]

When this is non-nil, then `magit-process-sentinel` raises an error if git exits with a non-zero exit status. For debugging purposes.

10.2 Section Plumbing

10.2.1 Creating Sections

magit-insert-section &rest *args*   [Macro]
> Insert a section at point.
>
> TYPE is the section type, a symbol. Many commands that act on the current section behave differently depending on that type. Also if a variable `magit-TYPE-section-map` exists, then use that as the text-property `keymap` of all text belonging to the section (but this may be overwritten in subsections). TYPE can also have the form `(eval FORM)` in which case FORM is evaluated at runtime.
>
> Optional VALUE is the value of the section, usually a string that is required when acting on the section.
>
> When optional HIDE is non-nil collapse the section body by default, i.e. when first creating the section, but not when refreshing the buffer. Otherwise, expand it by default. This can be overwritten using `magit-section-set-visibility-hook`. When a section is recreated during a refresh, then the visibility of predecessor is inherited and HIDE is ignored (but the hook is still honored).
>
> BODY is any number of forms that actually insert the section's heading and body. Optional NAME, if specified, has to be a symbol, which is then bound to the struct of the section being inserted.
>
> Before BODY is evaluated the `start` of the section object is set to the value of `point` and after BODY was evaluated its `end` is set to the new value of `point`; BODY is responsible for moving `point` forward.
>
> If it turns out inside BODY that the section is empty, then `magit-cancel-section` can be used to abort and remove all traces of the partially inserted section. This can happen when creating a section by washing Git's output and Git didn't actually output anything this time around.

magit-insert-heading &rest *args*   [Function]
> Insert the heading for the section currently being inserted.
>
> This function should only be used inside `magit-insert-section`.
>
> When called without any arguments, then just set the `content` slot of the object representing the section being inserted to a marker at `point`. The section should only contain a single line when this function is used like this.
>
> When called with arguments ARGS, which have to be strings, then insert those strings at point. The section should not contain any text before this happens and afterwards it should again only contain a single line. If the `face` property is set anywhere inside any of these strings, then insert all of them unchanged. Otherwise use the `magit-section-heading` face for all inserted text.
>
> The `content` property of the section struct is the end of the heading (which lasts from `start` to `content`) and the beginning of the body (which lasts from `content` to `end`). If the value of `content` is nil, then the section has no heading and its body cannot be collapsed. If a section does have a heading then its height must be exactly one line, including a trailing newline character. This isn't enforced; you are responsible

for getting it right. The only exception is that this function does insert a newline
character if necessary.

`magit-cancel-section`   [Function]
   Cancel the section currently being inserted. This exits the innermost call to
   `magit-insert-section` and removes all traces of what has already happened inside
   that call.

`magit-define-section-jumper` *sym title* **&optional** *value*   [Function]
   Define an interactive function to go to section SYM. TITLE is the displayed title of
   the section.

10.2.2 Section Selection

`magit-current-section`   [Function]
   Return the section at point.

`magit-region-sections` **&optional** *condition multiple*   [Function]
   Return a list of the selected sections.

   When the region is active and constitutes a valid section selection, then return a list
   of all selected sections. This is the case when the region begins in the heading of a
   section and ends in the heading of the same section or in that of a sibling section. If
   optional MULTIPLE is non-nil, then the region cannot begin and end in the same
   section.

   When the selection is not valid, then return nil. In this case, most commands that
   can act on the selected sections will instead act on the section at point.

   When the region looks like it would in any other buffer then the selection is invalid.
   When the selection is valid then the region uses the `magit-section-highlight` face.
   This does not apply to diffs where things get a bit more complicated, but even here
   if the region looks like it usually does, then that's not a valid selection as far as this
   function is concerned.

   If optional CONDITION is non-nil, then the selection not only has to be valid; all
   selected sections additionally have to match CONDITION, or nil is returned. See
   `magit-section-match` for the forms CONDITION can take.

`magit-region-values` **&optional** *condition multiple*   [Function]
   Return a list of the values of the selected sections.

   Return the values that themselves would be returned by `magit-region-sections`
   (which see).

10.2.3 Matching Sections

M-x `magit-describe-section-briefly`   (`magit-describe-section-briefly`)
        Show information about the section at point. This command is intended for
        debugging purposes.

`magit-section-ident` *section*   [Function]
   Return an unique identifier for SECTION. The return value has the form `((TYPE .`
   `VALUE)...)`.

`magit-get-section` *ident* **&optional** *root*   [Function]
>   Return the section identified by IDENT. IDENT has to be a list as returned by
>   `magit-section-ident`.

`magit-section-match` *condition* **&optional** *section*   [Function]
>   Return `t` if SECTION matches CONDITION. SECTION defaults to the section at
>   point. If SECTION is not specified and there also is no section at point, then return
>   `nil`.
>
>   CONDITION can take the following forms:
>
>   - `(CONDITION...)`
>
>   matches if any of the CONDITIONs matches.
>
>   - `[CLASS...]`
>
>   matches if the section's class is the same as the first CLASS or a subclass of that;
>   the section's parent class matches the second CLASS; and so on.
>
>   - `[* CLASS...]`
>
>   matches sections that match `[CLASS...]` and also recursively all their child sec-
>   tions.
>
>   - `CLASS`
>
>   matches if the section's class is the same as CLASS or a subclass of that; regard-
>   less of the classes of the parent sections.
>
>   Each CLASS should be a class symbol, identifying a class that derives from
>   `magit-section`. For backward compatibility CLASS can also be a "type symbol".
>   A section matches such a symbol if the value of its `type` slot is `eq`. If a type symbol
>   has an entry in `magit--section-type-alist`, then a section also matches that type
>   if its class is a subclass of the class that corresponds to the type as per that alist.
>
>   Note that it is not necessary to specify the complete section lineage as printed by
>   `magit-describe-section-briefly`, unless of course you want to be that precise.

`magit-section-value-if` *condition* **&optional** *section*   [Function]
>   If the section at point matches CONDITION, then return its value.
>
>   If optional SECTION is non-nil then test whether that matches instead. If there is no
>   section at point and SECTION is nil, then return nil. If the section does not match,
>   then return nil.
>
>   See `magit-section-match` for the forms CONDITION can take.

`magit-section-case` **&rest** *clauses*   [Function]
>   Choose among clauses on the type of the section at point.
>
>   Each clause looks like (CONDITION BODY...). The type of the section is com-
>   pared against each CONDITION; the BODY forms of the first match are evaluated
>   sequentially and the value of the last form is returned. Inside BODY the symbol `it` is
>   bound to the section at point. If no clause succeeds or if there is no section at point
>   return nil.
>
>   See `magit-section-match` for the forms CONDITION can take. Additionally a
>   CONDITION of t is allowed in the final clause and matches if no other CONDI-
>   TION match, even if there is no section at point.

`magit-root-section`   [Variable]
>   The root section in the current buffer. All other sections are descendants of this
>   section. The value of this variable is set by `magit-insert-section` and you should
>   never modify it.

For diff related sections a few additional tools exist.

`magit-diff-type` **&optional** *section*   [Function]
>   Return the diff type of SECTION.
>
>   The returned type is one of the symbols `staged`, `unstaged`, `committed`, or `undefined`.
>   This type serves a similar purpose as the general type common to all sections (which
>   is stored in the `type` slot of the corresponding `magit-section` struct) but takes
>   additional information into account. When the SECTION isn't related to diffs and
>   the buffer containing it also isn't a diff-only buffer, then return nil.
>
>   Currently the type can also be one of `tracked` and `untracked`, but these values are
>   not handled explicitly in every place they should be. A possible fix could be to just
>   return nil here.
>
>   The section has to be a `diff` or `hunk` section, or a section whose children are of type
>   `diff`. If optional SECTION is nil, return the diff type for the current section. In
>   buffers whose major mode is `magit-diff-mode` SECTION is ignored and the type is
>   determined using other means. In `magit-revision-mode` buffers the type is always
>   `committed`.

`magit-diff-scope` **&optional** *section strict*   [Function]
>   Return the diff scope of SECTION or the selected section(s).
>
>   A diff's "scope" describes what part of a diff is selected, it is a symbol, one of `region`,
>   `hunk`, `hunks`, `file`, `files`, or `list`. Do not confuse this with the diff "type", as
>   returned by `magit-diff-type`.
>
>   If optional SECTION is non-nil, then return the scope of that, ignoring the sections
>   selected by the region. Otherwise return the scope of the current section, or if the
>   region is active and selects a valid group of diff related sections, the type of these
>   sections, i.e. `hunks` or `files`. If SECTION (or if the current section that is nil) is a
>   `hunk` section and the region starts and ends inside the body of a that section, then
>   the type is `region`.
>
>   If optional STRICT is non-nil then return nil if the diff type of the section at point
>   is `untracked` or the section at point is not actually a `diff` but a `diffstat` section.

10.3 Refreshing Buffers

All commands that create a new Magit buffer or change what is being displayed in an
existing buffer do so by calling `magit-mode-setup`. Among other things, that function
sets the buffer local values of `default-directory` (to the top-level of the repository),
`magit-refresh-function`, and `magit-refresh-args`.

Buffers are refreshed by calling the function that is the local value of `magit-refresh-function` (a function named `magit-*-refresh-buffer`, where * may be something like
`diff`) with the value of `magit-refresh-args` as arguments.

`magit-mode-setup` *buffer switch-func mode refresh-func* **&optional**   [Macro]
    refresh-args
> This function displays and selects BUFFER, turns on MODE, and refreshes a first
> time.
>
> This function displays and optionally selects BUFFER by calling `magit-mode-`
> `display-buffer` with BUFFER, MODE and SWITCH-FUNC as arguments. Then
> it sets the local value of `magit-refresh-function` to REFRESH-FUNC and that of
> `magit-refresh-args` to REFRESH-ARGS. Finally it creates the buffer content by
> calling REFRESH-FUNC with REFRESH-ARGS as arguments.
>
> All arguments are evaluated before switching to BUFFER.

`magit-mode-display-buffer` *buffer mode* **&optional** *switch-function*   [Function]
> This function display BUFFER in some window and select it. BUFFER may be a
> buffer or a string, the name of a buffer. The buffer is returned.
>
> Unless BUFFER is already displayed in the selected frame, store the previous window
> configuration as a buffer local value, so that it can later be restored by `magit-mode-`
> `bury-buffer`.
>
> The buffer is displayed and selected using SWITCH-FUNCTION. If that is `nil` then
> `pop-to-buffer` is used if the current buffer's major mode derives from `magit-mode`.
> Otherwise `switch-to-buffer` is used.

`magit-refresh-function`   [Variable]
> The value of this buffer-local variable is the function used to refresh the current buffer.
> It is called with `magit-refresh-args` as arguments.

`magit-refresh-args`   [Variable]
> The list of arguments used by `magit-refresh-function` to refresh the current buffer.
> `magit-refresh-function` is called with these arguments.
>
> The value is usually set using `magit-mode-setup`, but in some cases it's also useful
> to provide commands that can change the value. For example, the `magit-diff-`
> `refresh` transient can be used to change any of the arguments used to display
> the diff, without having to specify again which differences should be shown, but
> `magit-diff-more-context`, `magit-diff-less-context` and `magit-diff-default-`
> `context` change just the `-U<N>` argument. In both case this is done by changing the
> value of this variable and then calling this `magit-refresh-function`.

10.4 Conventions

Also see Section 4.5.2 [Completion and Confirmation], page 24.

10.4.1 Theming Faces

The default theme uses blue for local branches, green for remote branches, and goldenrod
(brownish yellow) for tags. When creating a new theme, you should probably follow that
example. If your theme already uses other colors, then stick to that.

In older releases these reference faces used to have a background color and a box around
them. The basic default faces no longer do so, to make Magit buffers much less noisy, and
you should follow that example at least with regards to boxes. (Boxes were used in the past

to work around a conflict between the highlighting overlay and text property backgrounds. That's no longer necessary because highlighting no longer causes other background colors to disappear.) Alternatively you can keep the background color and/or box, but then have to take special care to adjust `magit-branch-current` accordingly. By default it looks mostly like `magit-branch-local`, but with a box (by default the former is the only face that uses a box, exactly so that it sticks out). If the former also uses a box, then you have to make sure that it differs in some other way from the latter.

The most difficult faces to theme are those related to diffs, headings, highlighting, and the region. There are faces that fall into all four groups - expect to spend some time getting this right.

The `region` face in the default theme, in both the light and dark variants, as well as in many other themes, distributed with Emacs or by third-parties, is very ugly. It is common to use a background color that really sticks out, which is ugly but if that were the only problem then it would be acceptable. Unfortunately many themes also set the foreground color, which ensures that all text within the region is readable. Without doing that there might be cases where some foreground color is too close to the region background color to still be readable. But it also means that text within the region loses all syntax highlighting.

I consider the work that went into getting the `region` face right to be a good indicator for the general quality of a theme. My recommendation for the `region` face is this: use a background color slightly different from the background color of the `default` face, and do not set the foreground color at all. So for a light theme you might use a light (possibly tinted) gray as the background color of `default` and a somewhat darker gray for the background of `region`. That should usually be enough to not collide with the foreground color of any other face. But if some other faces also set a light gray as background color, then you should also make sure it doesn't collide with those (in some cases it might be acceptable though).

Magit only uses the `region` face when the region is "invalid" by its own definition. In a Magit buffer the region is used to either select multiple sibling sections, so that commands which support it act on all of these sections instead of just the current section, or to select lines within a single hunk section. In all other cases, the section is considered invalid and Magit won't act on it. But such invalid sections happen, either because the user has not moved point enough yet to make it valid or because she wants to use a non-magit command to act on the region, e.g. `kill-region`.

So using the regular `region` face for invalid sections is a feature. It tells the user that Magit won't be able to act on it. It's acceptable if that face looks a bit odd and even (but less so) if it collides with the background colors of section headings and other things that have a background color.

Magit highlights the current section. If a section has subsections, then all of them are highlighted. This is done using faces that have "highlight" in their names. For most sections, `magit-section-highlight` is used for both the body and the heading. Like the `region` face, it should only set the background color to something similar to that of `default`. The highlight background color must be different from both the `region` background color and the `default` background color.

For diff related sections Magit uses various faces to highlight different parts of the selected section(s). Note that hunk headings, unlike all other section headings, by default have a background color, because it is useful to have very visible separators between hunks.

That face `magit-diff-hunk-heading`, should be different from both `magit-diff-hunk-heading-highlight` and `magit-section-highlight`, as well as from `magit-diff-context` and `magit-diff-context-highlight`. By default we do that by changing the foreground color. Changing the background color would lead to complications, and there are already enough we cannot get around. (Also note that it is generally a good idea for section headings to always be bold, but only for sections that have subsections).

When there is a valid region selecting diff-related sibling sections, i.e. multiple files or hunks, then the bodies of all these sections use the respective highlight faces, but additionally the headings instead use one of the faces `magit-diff-file-heading-selection` or `magit-diff-hunk-heading-selection`. These faces have to be different from the regular highlight variants to provide explicit visual indication that the region is active.

When theming diff related faces, start by setting the option `magit-diff-refine-hunk` to `all`. You might personally prefer to only refine the current hunk or not use hunk refinement at all, but some of the users of your theme want all hunks to be refined, so you have to cater to that.

(Also turn on `magit-diff-highlight-indentation`, `magit-diff-highlight-trailing`, and `magit-diff-paint-whitespace`; and insert some whitespace errors into the code you use for testing.)

For e.g. "added lines" you have to adjust three faces: `magit-diff-added`, `magit-diff-added-highlight`, and `smerge-refined-added`. Make sure that the latter works well with both of the former, as well as `smerge-other` and `diff-added`. Then do the same for the removed lines, context lines, lines added by us, and lines added by them. Also make sure the respective added, removed, and context faces use approximately the same saturation for both the highlighted and unhighlighted variants. Also make sure the file and diff headings work nicely with context lines (e.g. make them look different). Line faces should set both the foreground and the background color. For example, for added lines use two different greens.

It's best if the foreground color of both the highlighted and the unhighlighted variants are the same, so you will need to have to find a color that works well on the highlight and unhighlighted background, the refine background, and the highlight context background. When there is an hunk internal region, then the added- and removed-lines background color is used only within that region. Outside the region the highlighted context background color is used. This makes it easier to see what is being staged. With an hunk internal region the hunk heading is shown using `magit-diff-hunk-heading-selection`, and so are the thin lines that are added around the lines that fall within the region. The background color of that has to be distinct enough from the various other involved background colors.

Nobody said this would be easy. If your theme restricts itself to a certain set of colors, then you should make an exception here. Otherwise it would be impossible to make the diffs look good in each and every variation. Actually you might want to just stick to the default definitions for these faces. You have been warned. Also please note that if you do not get this right, this will in some cases look to users like bugs in Magit - so please do it right or not at all.

Appendix A FAQ

The next two nodes lists frequently asked questions. For a list of frequently **and recently** asked questions, i.e. questions that haven't made it into the manual yet, see `https://github.com/magit/magit/wiki/FAQ`.

Please also use the Chapter 11 [Debugging Tools], page 133.

A.1 FAQ - How to . . . ?

A.1.1 How to show git's output?

To show the output of recently run git commands, press `$` (or, if that isn't available, `M-x magit-process-buffer`). This will show a buffer containing a section per git invocation; as always press `TAB` to expand or collapse them.

By default, git's output is only inserted into the process buffer if it is run for side-effects. When the output is consumed in some way, also inserting it into the process buffer would be too expensive. For debugging purposes, it's possible to do so anyway by setting `magit-git-debug` to `t`.

A.1.2 How to install the gitman info manual?

Git's manpages can be exported as an info manual called `gitman`. Magit's own info manual links to nodes in that manual instead of the actual manpages because Info doesn't support linking to manpages.

Unfortunately some distributions do not install the `gitman` manual by default and you will have to install a separate documentation package to get it.

Magit patches Info adding the ability to visit links to the `gitman` Info manual by instead viewing the respective manpage. If you prefer that approach, then set the value of `magit-view-git-manual-method` to one of the supported packages `man` or `woman`, e.g.:

```
(setq magit-view-git-manual-method 'man)
```

A.1.3 How to show diffs for gpg-encrypted files?

Git supports showing diffs for encrypted files, but has to be told to do so. Since Magit just uses Git to get the diffs, configuring Git also affects the diffs displayed inside Magit.

```
git config --global diff.gpg.textconv "gpg --no-tty --decrypt"
echo "*.gpg filter=gpg diff=gpg" > .gitattributes
```

A.1.4 How does branching and pushing work?

Please see Section 6.6 [Branching], page 71, and `http://emacsair.me/2016/01/17/magit-2.4`

A.1.5 Can Magit be used as `ediff-version-control-package`?

No, it cannot. For that to work the functions `ediff-magit-internal` and `ediff-magit-merge-internal` would have to be implemented, and they are not. These two functions are only used by the three commands `ediff-revision`, `ediff-merge-revisions-with-ancestor`, and `ediff-merge-revisions`.

These commands only delegate the task of populating buffers with certain revisions to the "internal" functions. The equally important task of determining which revisions are to be compared/merged is not delegated. Instead this is done without any support whatsoever from the version control package/system - meaning that the user has to enter the revisions explicitly. Instead of implementing `ediff-magit-internal` we provide `magit-ediff-compare`, which handles both tasks like it is 2005.

The other commands `ediff-merge-revisions` and `ediff-merge-revisions-with-ancestor` are normally not what you want when using a modern version control system like Git. Instead of letting the user resolve only those conflicts which Git could not resolve on its own, they throw away all work done by Git and then expect the user to manually merge all conflicts, including those that had already been resolved. That made sense back in the days when version control systems couldn't merge (or so I have been told), but not anymore. Once in a blue moon you might actually want to see all conflicts, in which case you **can** use these commands, which then use `ediff-vc-merge-internal`. So we don't actually have to implement `ediff-magit-merge-internal`. Instead we provide the more useful command `magit-ediff-resolve` which only shows yet-to-be resolved conflicts.

A.2 FAQ - Issues and Errors

A.2.1 Magit is slow

See Section 9.2.2 [Performance], page 113.

A.2.2 I changed several thousand files at once and now Magit is unusable

Magit is **currently** not expected to work under such conditions. It sure would be nice if it did, and v2.5 will hopefully be a big step into that direction. But it might take until v3.1 to accomplish fully satisfactory performance, because that requires some heavy refactoring.

But for now we recommend you use the command line to complete this one commit. Also see Section 9.2.2 [Performance], page 113.

A.2.3 I am having problems committing

That likely means that Magit is having problems finding an appropriate emacsclient executable. See Section "Configuring With-Editor" in `with-editor` and Section "Debugging" in `with-editor`.

A.2.4 I am using MS Windows and cannot push with Magit

It's almost certain that Magit is only incidental to this issue. It is much more likely that this is a configuration issue, even if you can push on the command line.

Detailed setup instructions can be found at `https://github.com/magit/magit/wiki/Pushing-with-Magit-from-Windows`.

A.2.5 I am using OS X and SOMETHING works in shell, but not in Magit

This usually occurs because Emacs doesn't have the same environment variables as your shell. Try installing and configuring `https://github.com/purcell/`

exec-path-from-shell. By default it synchronizes $PATH, which helps Magit find the same `git` as the one you are using on the shell.

If SOMETHING is "passphrase caching with gpg-agent for commit and/or tag signing", then you'll also need to synchronize `$GPG_AGENT_INFO`.

A.2.6 Diffs contain control sequences

This happens when you configure Git to always color diffs and/or all of its output. The valid values for relevant Git variables `color.ui` and `color.diff` are `false`, `true` and `always`, and the default is `true`. You should leave it that way because then you get colorful output in terminals by default but when git's output is consumed by something else, then no color control sequences are used.

If you actually use some other tool that requires setting `color.ui` and/or `color.diff` to `always` (which is highly unlikely), then you can override these settings just for Magit by using:

```
(setq magit-git-global-arguments
      (nconc magit-git-global-arguments
             '("-c" "color.ui=false"
               "-c" "color.diff=false")))
```

A.2.7 Expanding a file to show the diff causes it to disappear

This is probably caused by a change of a `diff.*` Git variable. You probably set that variable for a reason, and should therefore only undo that setting in Magit by customizing `magit-git-global-arguments`.

A.2.8 Point is wrong in the `COMMIT_EDITMSG` buffer

Neither Magit nor 'git-commit' fiddle with point in the buffer used to write commit messages, so something else must be doing it.

You have probably globally enabled a mode which does restore point in file-visiting buffers. It might be a bit surprising, but when you write a commit message, then you are actually editing a file.

So you have to figure out which package is doing. `saveplace`, `pointback`, and `session` are likely candidates. These snippets might help:

```
(setq session-name-disable-regexp "\\(?:\\'’\\.git/[A-Z_]+\\'\\)")

(with-eval-after-load 'pointback
  (lambda ()
    (when (or git-commit-mode git-rebase-mode)
      (pointback-mode -1))))
```

A.2.9 The mode-line information isn't always up-to-date

Magit is not responsible for the version control information that is being displayed in the mode-line and looks something like `Git-master`. The built-in "Version Control" package, also known as "VC", updates that information, and can be told to do so more often:

```
(setq auto-revert-check-vc-info t)
```

But doing so isn't good for performance. For more (overly optimistic) information see Section "VC Mode Line" in `emacs`.

If you don't really care about seeing that information in the mode-line, but just don't want to see *incorrect* information, then consider disabling VC when using Git:

```
(setq vc-handled-backends (delq 'Git vc-handled-backends))
```

Or to disable it completely:

```
(setq vc-handled-backends nil)
```

A.2.10 A branch and tag sharing the same name breaks SOMETHING

Or more generally, ambiguous refnames break SOMETHING.

Magit assumes that refs are named non-ambiguously across the "refs/heads/", "refs/tags/", and "refs/remotes/" namespaces (i.e., all the names remain unique when those prefixes are stripped). We consider ambiguous refnames unsupported and recommend that you use a non-ambiguous naming scheme. However, if you do work with a repository that has ambiguous refnames, please report any issues you encounter so that we can investigate whether there is a simple fix.

A.2.11 My Git hooks work on the command-line but not inside Magit

When Magit calls `git` it adds a few global arguments including `--literal-pathspecs` and the `git` process started by Magit then passes that setting on to other `git` process it starts itself. It does so by setting the environment variable `GIT_LITERAL_PATHSPECS`, not by calling subprocesses with the `--literal-pathspecs`. You can therefore override this setting in hook scripts using `unset GIT_LITERAL_PATHSPECS`.

A.2.12 `git-commit-mode` isn't used when committing from the command-line

The reason for this is that `git-commit.el` has not been loaded yet and/or that the server has not been started yet. These things have always already been taken care of when you commit from Magit because in order to do so, Magit has to be loaded and doing that involves loading `git-commit` and starting the server.

If you want to commit from the command-line, then you have to take care of these things yourself. Your `init.el` file should contain:

```
(require 'git-commit)
(server-mode)
```

Instead of '(require 'git-commit)' you may also use:

```
(load "/path/to/magit-autoloads.el")
```

You might want to do that because loading `git-commit` causes large parts of Magit to be loaded.

There are also some variations of (`server-mode`) that you might want to try. Personally I use:

```
(use-package server
  :config (or (server-running-p) (server-mode)))
```

Now you can use:

```
$ emacs&
$ EDITOR=emacsclient git commit
```

However you cannot use:

```
$ killall emacs
$ EDITOR="emacsclient --alternate-editor emacs" git commit
```

This will actually end up using `emacs`, not `emacsclient`. If you do this, then can still edit the commit message but `git-commit-mode` won't be used and you have to exit `emacs` to finish the process.

Tautology ahead. If you want to be able to use `emacsclient` to connect to a running `emacs` instance, even though no `emacs` instance is running, then you cannot use `emacsclient` directly.

Instead you have to create a script that does something like this:

Try to use `emacsclient` (without using `--alternate-editor`). If that succeeds, do nothing else. Otherwise start `emacs &` (and `init.el` must call `server-start`) and try to use `emacsclient` again.

A.2.13 Point ends up inside invisible text when jumping to a file-visiting buffer

This can happen when you type `RET` on a hunk to visit the respective file at the respective position. One solution to this problem is to use `global-reveal-mode`. It makes sure that text around point is always visible. If that is too drastic for your taste, then you may instead use `magit-diff-visit-file-hook` to reveal the text, possibly using `reveal-post-command` or for Org buffers `org-reveal`.

11 Debugging Tools

Magit and its dependencies provide a few debugging tools, and we appreciate it very much if you use those tools before reporting an issue. Please include all relevant output when reporting an issue.

M-x magit-version   (magit-version)

> This command shows the currently used versions of Magit, Git, and Emacs in the echo area. Non-interactively this just returns the Magit version.

M-x magit-emacs-Q-command   (magit-emacs-Q-command)

> This command shows a debugging shell command in the echo area and adds it to the kill ring. Paste that command into a shell and run it.
>
> This shell command starts `emacs` with only `magit` and its dependencies loaded. Neither your configuration nor other installed packages are loaded. This makes it easier to determine whether some issue lays with Magit or something else.
>
> If you run Magit from its Git repository, then you should be able to use `make emacs-Q` instead of the output of this command.

M-x magit-debug-git-executable   (magit-debug-git-executable)

> This command displays a buffer containing information about the available and used `git` executable(s), and can be useful when investigating `exec-path` issues.
>
> Also see Section 4.6.4 [Git Executable], page 29.

M-x with-editor-debug   (with-editor-debug)

> This command displays a buffer containing information about the available and used ˜emacsclient executable(s), and can be useful when investigating why Magit (or rather `with-editor`) cannot find an appropriate `emacsclient` executable.
>
> Also see Section "Debugging" in `with-editor`.

Please also see the Appendix A [FAQ], page 128.

Appendix B Keystroke Index

R

S

T

U

V

W

X

Y

Z

Appendix C Command Index

S

W

Appendix D Function Index

W

Appendix E Variable Index

N

P

R