



Linux Ubuntu 26.04 Manual Pages on command 'git-rev-parse.1'

\$ man git-rev-parse.1

GIT-REV-PARSE(1) Git Manual GIT-REV-PARSE(1)

NAME

git-rev-parse - Pick out and massage parameters

SYNOPSIS

git rev-parse [<options>] <arg>...

DESCRIPTION

Many Git porcelainish commands take a mixture of flags (i.e. parameters that begin with a dash -) and parameters meant for the underlying git rev-list command they use internally and flags and parameters for the other commands they use downstream of git rev-list. The primary purpose of this command is to allow calling programs to distinguish between them. There are a few other operation modes that have nothing to do with the above "help parse command line options".

Unless otherwise specified, most of the options and operation modes require you to run this command inside a git repository or a working tree that is under the control of a git repository, and will give you a fatal error otherwise.

OPTIONS

Operation Modes

Each of these options must appear first on the command line.

--parseopt

Use git rev-parse in option parsing mode (see PARSEOPT section below). The command in this mode can be used outside a repository or a working tree controlled by a repository.

--sq-quote

Use git rev-parse in shell quoting mode (see SQ-QUOTE section below). In contrast to the

--sq option below, this mode only does quoting. Nothing else is done to command input.

The command in this mode can be used outside a repository or a working tree controlled by a repository.

Options for --parseopt

--keep-dashdash

Only meaningful in --parseopt mode. Tells the option parser to echo out the first -- met instead of skipping it.

--stop-at-non-option

Only meaningful in --parseopt mode. Lets the option parser stop at the first non-option argument. This can be used to parse sub-commands that take options themselves.

--stuck-long

Only meaningful in --parseopt mode. Output the options in their long form if available, and with their arguments stuck.

Options for Filtering

--revs-only

Do not output flags and parameters not meant for git rev-list command.

--no-revs

Do not output flags and parameters meant for git rev-list command.

--flags

Do not output non-flag parameters.

--no-flags

Do not output flag parameters.

Options for Output

--default <arg>

If there is no parameter given by the user, use <arg> instead.

--prefix <arg>

Behave as if git rev-parse was invoked from the <arg> subdirectory of the working tree. Any relative filenames are resolved as if they are prefixed by <arg> and will be printed in that form.

This can be used to convert arguments to a command run in a subdirectory so that they can still be used after moving to the top-level of the repository. For example:

```
prefix=$(git rev-parse --show-prefix)
```

```
cd "$(git rev-parse --show-toplevel)"
```

rev-parse provides the -- needed for 'set'

```
eval "set $(git rev-parse --sq --prefix "$prefix" -- "$@")"
```

--verify

Verify that exactly one parameter is provided, and that it can be turned into a raw 20-byte SHA-1 that can be used to access the object database. If so, emit it to the standard output; otherwise, error out.

If you want to make sure that the output actually names an object in your object database and/or can be used as a specific type of object you require, you can add the `^{type}` peeling operator to the parameter. For example, `git rev-parse "$VAR^{commit}"` will make sure \$VAR names an existing object that is a commit-ish (i.e. a commit, or an annotated tag that points at a commit). To make sure that \$VAR names an existing object of any type, `git rev-parse "$VAR^{object}"` can be used.

Note that if you are verifying a name from an untrusted source, it is wise to use `--end-of-options` so that the name argument is not mistaken for another option.

-q, --quiet

Only meaningful in --verify mode. Do not output an error message if the first argument is not a valid object name; instead exit with non-zero status silently. SHA-1s for valid object names are printed to stdout on success.

--sq

Usually the output is made one line per flag and parameter. This option makes output a single line, properly quoted for consumption by shell. Useful when you expect your parameter to contain whitespaces and newlines (e.g. when using `pickaxe -S` with `git diff-*`). In contrast to the `--sq-quote` option, the command input is still interpreted as usual.

--short[=<length>]

Same as --verify but shortens the object name to a unique prefix with at least length characters. The minimum length is 4, the default is the effective value of the `core.abbrev` configuration variable (see `git-config(1)`).

--not

When showing object names, prefix them with ^ and strip ^ prefix from the object names that already have one.

--abbrev-ref[=(strict|loose)]

A non-ambiguous short name of the objects name. The option `core.warnAmbiguousRefs` is

used to select the strict abbreviation mode.

`--symbolic`

Usually the object names are output in SHA-1 form (with possible ^ prefix); this option makes them output in a form as close to the original input as possible.

`--symbolic-full-name`

This is similar to `--symbolic`, but it omits input that are not refs (i.e. branch or tag names; or more explicitly disambiguating "heads/master" form, when you want to name the "master" branch when there is an unfortunately named tag "master"), and shows them as full refnames (e.g. "refs/heads/master").

`--output-object-format=(sha1|sha256|storage)`

Allow oids to be input from any object format that the current repository supports.

Specifying "sha1" translates if necessary and returns a sha1 oid.

Specifying "sha256" translates if necessary and returns a sha256 oid.

Specifying "storage" translates if necessary and returns an oid in encoded in the storage hash algorithm.

Options for Objects

`--all`

Show all refs found in refs/.

`--branches[=<pattern>], --tags[=<pattern>], --remotes[=<pattern>]`

Show all branches, tags, or remote-tracking branches, respectively (i.e., refs found in refs/heads, refs/tags, or refs/remotes, respectively).

If a pattern is given, only refs matching the given shell glob are shown. If the pattern does not contain a globbing character (? , * , or [] , it is turned into a prefix match by appending /*.

`--glob=<pattern>`

Show all refs matching the shell glob pattern pattern. If the pattern does not start with refs/, this is automatically prepended. If the pattern does not contain a globbing character (? , * , or [] , it is turned into a prefix match by appending /*.

`--exclude=<glob-pattern>`

Do not include refs matching <glob-pattern> that the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with refs/heads, refs/tags, or refs/remotes when applied to --branches, --tags, or --remotes, respectively, and they must begin with refs/ when applied to --glob or --all. If a trailing /* is intended, it must be given explicitly.

--exclude-hidden=(fetch|receive|uploadpack)

Do not include refs that would be hidden by git-fetch, git-receive-pack or git-upload-pack by consulting the appropriate fetch.hideRefs, receive.hideRefs or uploadpack.hideRefs configuration along with transfer.hideRefs (see git-config(1)). This option affects the next pseudo-ref option --all or --glob and is cleared after processing them.

--disambiguate=<prefix>

Show every object whose name begins with the given prefix. The <prefix> must be at least 4 hexadecimal digits long to avoid listing each and every object in the repository by mistake.

Options for Files

--local-env-vars

List the GIT_* environment variables that are local to the repository (e.g. GIT_DIR or GIT_WORK_TREE, but not GIT_EDITOR). Only the names of the variables are listed, not their value, even if they are set.

--path-format=(absolute|relative)

Controls the behavior of certain other options. If specified as absolute, the paths printed by those options will be absolute and canonical. If specified as relative, the paths will be relative to the current working directory if that is possible. The default is option specific.

This option may be specified multiple times and affects only the arguments that follow it on the command line, either to the end of the command line or the next instance of this option.

The following options are modified by --path-format:

--git-dir

Show \$GIT_DIR if defined. Otherwise show the path to the .git directory. The path shown, when relative, is relative to the current working directory.

If \$GIT_DIR is not defined and the current directory is not detected to lie in a Git repository or work tree print a message to stderr and exit with nonzero status.

`--git-common-dir`

Show `$GIT_COMMON_DIR` if defined, else `$GIT_DIR`.

`--resolve-git-dir <path>`

Check if `<path>` is a valid repository or a gitfile that points at a valid repository, and print the location of the repository. If `<path>` is a gitfile then the resolved path to the real repository is printed.

`--git-path <path>`

Resolve "`$GIT_DIR/<path>`" and takes other path relocation variables such as `$GIT_OBJECT_DIRECTORY`, `$GIT_INDEX_FILE`... into account. For example, if `$GIT_OBJECT_DIRECTORY` is set to `/foo/bar` then `"git rev-parse --git-path objects/abc"` returns `/foo/bar/abc`.

`--show-toplevel`

Show the (by default, absolute) path of the top-level directory of the working tree. If there is no working tree, report an error.

`--show-superproject-working-tree`

Show the absolute path of the root of the superproject's working tree (if exists) that uses the current repository as its submodule. Outputs nothing if the current repository is not used as a submodule by any project.

`--shared-index-path`

Show the path to the shared index file in split index mode, or empty if not in split-index mode.

The following options are unaffected by `--path-format`:

`--absolute-git-dir`

Like `--git-dir`, but its output is always the canonicalized absolute path.

`--is-inside-git-dir`

When the current working directory is below the repository directory print `"true"`, otherwise `"false"`.

`--is-inside-work-tree`

When the current working directory is inside the work tree of the repository print `"true"`, otherwise `"false"`.

`--is-bare-repository`

When the repository is bare print `"true"`, otherwise `"false"`.

`--is-shallow-repository`

When the repository is shallow print "true", otherwise "false".

--show-cdup

When the command is invoked from a subdirectory, show the path of the top-level directory relative to the current directory (typically a sequence of "../", or an empty string).

--show-prefix

When the command is invoked from a subdirectory, show the path of the current directory relative to the top-level directory.

--show-object-format[=(storage|input|output|compat)]

Show the object format (hash algorithm) used for the repository for storage inside the .git directory, input, output, or compatibility. For input, multiple algorithms may be printed, space-separated. If compat is requested and no compatibility algorithm is enabled, prints an empty line. If not specified, the default is "storage".

--show-ref-format

Show the reference storage format used for the repository.

Other Options

--since=<datestring>, --after=<datestring>

Parse the date string, and output the corresponding --max-age= parameter for git rev-list.

--until=<datestring>, --before=<datestring>

Parse the date string, and output the corresponding --min-age= parameter for git rev-list.

<arg>...

Flags and parameters to be parsed.

SPECIFYING REVISIONS

A revision parameter <rev> typically, but not necessarily, names a commit object. It uses what is called an extended SHA-1 syntax. Here are various ways to spell object names. The ones listed near the end of this list name trees and blobs contained in a commit.

Note

This document shows the "raw" syntax as seen by git. The shell and other UIs might require additional quoting to protect special characters and to avoid word splitting.

<sha1>, e.g. dae86e1950b1277e545cee180551750029cfe735, dae86e

The full SHA-1 object name (40-byte hexadecimal string), or a leading substring that is

unique within the repository. E.g. dae86e1950b1277e545cee180551750029cfe735 and dae86e both name the same commit object if there is no other object in your repository whose object name starts with dae86e.

<describeOutput>, e.g. v1.7.4.2-679-g3bee7fb

Output from git describe; i.e. a closest tag, optionally followed by a dash and a number of commits, followed by a dash, a g, and an abbreviated object name.

<refname>, e.g. master, heads/master, refs/heads/master

A symbolic ref name. E.g. master typically means the commit object referenced by refs/heads/master. If you happen to have both heads/master and tags/master, you can explicitly say heads/master to tell Git which one you mean. When ambiguous, a <refname> is disambiguated by taking the first match in the following rules:

1. If \$GIT_DIR/<refname> exists, that is what you mean (this is usually useful only for

HEAD, FETCH_HEAD, ORIG_HEAD, MERGE_HEAD, REBASE_HEAD, REVERT_HEAD, CHERRY_PICK_HEAD, BISECT_HEAD and AUTO_MERGE);

2. otherwise, refs/<refname> if it exists;
3. otherwise, refs/tags/<refname> if it exists;
4. otherwise, refs/heads/<refname> if it exists;
5. otherwise, refs/remotes/<refname> if it exists;
6. otherwise, refs/remotes/<refname>/HEAD if it exists.

HEAD

names the commit on which you based the changes in the working tree.

FETCH_HEAD

records the branch which you fetched from a remote repository with your last git fetch invocation.

ORIG_HEAD

is created by commands that move your HEAD in a drastic way (git am, git merge, git rebase, git reset), to record the position of the HEAD before their operation, so that you can easily change the tip of the branch back to the state before you ran them.

MERGE_HEAD

records the commit(s) which you are merging into your branch when you run git merge.

REBASE_HEAD

during a rebase, records the commit at which the operation is currently stopped, either because of conflicts or an edit command in an interactive rebase.

REVERT_HEAD

records the commit which you are reverting when you run `git revert`.

CHERRY_PICK_HEAD

records the commit which you are cherry-picking when you run `git cherry-pick`.

BISECT_HEAD

records the current commit to be tested when you run `git bisect --no-checkout`.

AUTO_MERGE

records a tree object corresponding to the state the `ort` merge strategy wrote to the working tree when a merge operation resulted in conflicts.

Note that any of the `refs/*` cases above may come either from the `$GIT_DIR/refs` directory or from the `$GIT_DIR/packed-refs` file. While the ref name encoding is unspecified, UTF-8 is preferred as some output processing may assume ref names in UTF-8.

@

@ alone is a shortcut for HEAD.

[<refname>]@{<date>}, e.g. `master@{yesterday}`, `HEAD@{5 minutes ago}`

A ref followed by the suffix @ with a date specification enclosed in a brace pair (e.g. `{yesterday}`, `{1 month 2 weeks 3 days 1 hour 1 second ago}` or `{1979-02-26 18:30:00}`) specifies the value of the ref at a prior point in time. This suffix may only be used immediately following a ref name and the ref must have an existing log (`$GIT_DIR/logs/<ref>`). Note that this looks up the state of your local ref at a given time; e.g., what was in your local master branch last week. If you want to look at commits made during certain times, see `--since` and `--until`.

<refname>@{<n>}, e.g. `master@{1}`

A ref followed by the suffix @ with an ordinal specification enclosed in a brace pair (e.g. `{1}`, `{15}`) specifies the n-th prior value of that ref. For example `master@{1}` is the immediate prior value of master while `master@{5}` is the 5th prior value of master. This suffix may only be used immediately following a ref name and the ref must have an existing log (`$GIT_DIR/logs/<refname>`).

@{<n>}, e.g. `@{1}`

You can use the @ construct with an empty ref part to get at a reflog entry of the current branch. For example, if you are on branch `blabla` then `@{1}` means the same as

blabla@{1}.

@{-<n>}, e.g. @{-1}

The construct @{-<n>} means the <n>th branch/commit checked out before the current one.

[<branchname>]@{upstream}, e.g. master@{upstream}, @{u}

A branch B may be set up to build on top of a branch X (configured with branch.<name>.merge) at a remote R (configured with the branch X taken from remote R, typically found at refs/remotes/R/X.

[<branchname>]@{push}, e.g. master@{push}, @{push}

The suffix @{push} reports the branch "where we would push to" if git push were run while branchname was checked out (or the current HEAD if no branchname is specified).

Like for @{upstream}, we report the remote-tracking branch that corresponds to that branch at the remote.

Here's an example to make it more clear:

```
$ git config push.default current
$ git config remote.pushdefault myfork
$ git switch -c mybranch origin/master
$ git rev-parse --symbolic-full-name @{upstream}
refs/remotes/origin/master
$ git rev-parse --symbolic-full-name @{push}
refs/remotes/myfork/mybranch
```

Note in the example that we set up a triangular workflow, where we pull from one location and push to another. In a non-triangular workflow, @{push} is the same as @{upstream}, and there is no need for it.

This suffix is also accepted when spelled in uppercase, and means the same thing no matter the case.

<rev>^{<n>}, e.g. HEAD^, v1.5.1^0

A suffix ^ to a revision parameter means the first parent of that commit object. ^<n> means the <n>th parent (i.e. <rev>^ is equivalent to <rev>^1). As a special rule, <rev>^0 means the commit itself and is used when <rev> is the object name of a tag object that refers to a commit object.

<rev>~[<n>], e.g. HEAD~, master~3

A suffix ~ to a revision parameter means the first parent of that commit object. A suffix ~<n> to a revision parameter means the commit object that is the <n>th generation

ancestor of the named commit object, following only the first parents. I.e. `<rev>~3` is equivalent to `<rev>^^` which is equivalent to `<rev>^1^1^1`. See below for an illustration of the usage of this form.

`<rev>^{<type>}`, e.g. `v0.99.8^{commit}`

A suffix `^` followed by an object type name enclosed in brace pair means dereference the object at `<rev>` recursively until an object of type `<type>` is found or the object cannot be dereferenced anymore (in which case, barf). For example, if `<rev>` is a commit-ish, `<rev>^{commit}` describes the corresponding commit object. Similarly, if `<rev>` is a tree-ish, `<rev>^{tree}` describes the corresponding tree object. `<rev>^0` is a short-hand for `<rev>^{commit}`.

`<rev>^{object}` can be used to make sure `<rev>` names an object that exists, without requiring `<rev>` to be a tag, and without dereferencing `<rev>`; because a tag is already an object, it does not have to be dereferenced even once to get to an object.

`<rev>^{tag}` can be used to ensure that `<rev>` identifies an existing tag object.

`<rev>^{}`, e.g. `v0.99.8^{}`

A suffix `^` followed by an empty brace pair means the object could be a tag, and dereference the tag recursively until a non-tag object is found.

`<rev>^{/<text>}`, e.g. `HEAD^{/fix nasty bug}`

A suffix `^` to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the `:/fix nasty bug` syntax below except that it returns the youngest matching commit which is reachable from the `<rev>` before `^`.

`:/<text>`, e.g. `:/fix nasty bug`

A colon, followed by a slash, followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from any ref, including HEAD. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. `:/^foo`. The special sequence `:/!` is reserved for modifiers to what is matched. `:/!-foo` performs a negative match, while `:/!!foo` matches a literal `!` character, followed by `foo`. Any other sequence beginning with `:/!` is reserved for now. Depending on the given text, the shell's word splitting rules might require additional quoting.

`<rev>:<path>`, e.g. `HEAD:README`, `master:./README`

A suffix `:` followed by a path names the blob or tree at the given path in the tree-ish object named by the part before the colon. A path starting with `./` or `../` is relative to

the current working directory. The given path will be converted to be relative to the working tree's root directory. This is most useful to address a blob or tree from a commit or tree that has the same tree structure as the working tree.

:<n>:<path>, e.g. :0:README, :README

A colon, optionally followed by a stage number (0 to 3) and a colon, followed by a path, names a blob object in the index at the given path. A missing stage number (and the colon that follows it) names a stage 0 entry. During a merge, stage 1 is the common ancestor, stage 2 is the target branch's version (typically the current branch), and stage 3 is the version from the branch which is being merged.

Here is an illustration, by Jon Loeliger. Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.



$$A = A^0$$

$$B = A^1 = A_{\sim 1}$$

$$C = A^2$$

$$D = A^{11} = A_{\sim 2}$$

$$E = B^2 = A^{22}$$

$$F = B^3 = A^{33}$$

$$G = A^{111} = A_{\sim 3}$$

$$H = D^2 = B^{22} = A^{222} = A_{\sim 2}^2$$

$$I = F^1 = B^{31} = A^{331}$$

$$J = F^2 = B^{32} = A^{332}$$

SPECIFYING RANGES

History traversing commands such as `git log` operate on a set of commits, not just a single

commit.

For these commands, specifying a single revision, using the notation described in the previous section, means the set of commits reachable from the given commit.

Specifying several revisions means the set of commits reachable from any of the given commits.

A commit's reachable set is the commit itself and the commits in its ancestry chain.

There are several notations to specify a set of connected commits (called a "revision range"), illustrated below.

Commit Exclusions

`^<rev>` (caret) Notation

To exclude commits reachable from a commit, a prefix `^` notation is used. E.g. `^r1 r2` means commits reachable from `r2` but exclude the ones reachable from `r1` (i.e. `r1` and its ancestors).

Dotted Range Notations

The `..` (two-dot) Range Notation

The `^r1 r2` set operation appears so often that there is a shorthand for it. When you have two commits `r1` and `r2` (named according to the syntax explained in SPECIFYING REVISIONS above), you can ask for commits that are reachable from `r2` excluding those that are reachable from `r1` by `^r1 r2` and it can be written as `r1..r2`.

The `...` (three-dot) Symmetric Difference Notation

A similar notation `r1...r2` is called symmetric difference of `r1` and `r2` and is defined as `r1 r2 --not $(git merge-base --all r1 r2)`. It is the set of commits that are reachable from either one of `r1` (left side) or `r2` (right side) but not from both.

In these two shorthand notations, you can omit one end and let it default to `HEAD`. For example, `origin..` is a shorthand for `origin..HEAD` and asks "What did I do since I forked from the origin branch?" Similarly, `..origin` is a shorthand for `HEAD..origin` and asks "What did the origin do since I forked from them?" Note that `..` would mean `HEAD..HEAD` which is an empty range that is both reachable and unreachable from `HEAD`.

Commands that are specifically designed to take two distinct ranges (e.g. "git range-diff R1 R2" to compare two ranges) do exist, but they are exceptions. Unless otherwise noted, all "git" commands that operate on a set of commits work on a single revision range. In other words, writing two "two-dot range notation" next to each other, e.g.

```
$ git log A..B C..D
```

does not specify two revision ranges for most commands. Instead it will name a single connected set of commits, i.e. those that are reachable from either B or D but are reachable from neither A or C. In a linear history like this:

```
---A---B---o---o---C---D
```

because A and B are reachable from C, the revision range specified by these two dotted ranges is a single commit D.

Other <rev>^ Parent Shorthand Notations

Three other shorthands exist, particularly useful for merge commits, for naming a set that is formed by a commit and its parent commits.

The `r1^@` notation means all parents of `r1`.

The `r1^!` notation includes commit `r1` but excludes all of its parents. By itself, this notation denotes the single commit `r1`.

The `<rev>^[<n>]` notation includes `<rev>` but excludes the `<n>`th parent (i.e. a shorthand for `<rev>^<n>..<rev>`), with `<n> = 1` if not given. This is typically useful for merge commits where you can just pass `<commit>^-` to get all the commits in the branch that was merged in merge commit `<commit>` (including `<commit>` itself).

While `<rev>^<n>` was about specifying a single commit parent, these three notations also consider its parents. For example you can say `HEAD^2^@`, however you cannot say `HEAD^@^2`.

REVISION RANGE SUMMARY

`<rev>`

Include commits that are reachable from `<rev>` (i.e. `<rev>` and its ancestors).

`^<rev>`

Exclude commits that are reachable from `<rev>` (i.e. `<rev>` and its ancestors).

`<rev1>..<rev2>`

Include commits that are reachable from `<rev2>` but exclude those that are reachable from `<rev1>`. When either `<rev1>` or `<rev2>` is omitted, it defaults to `HEAD`.

`<rev1>...<rev2>`

Include commits that are reachable from either `<rev1>` or `<rev2>` but exclude those that are reachable from both. When either `<rev1>` or `<rev2>` is omitted, it defaults to `HEAD`.

`<rev>^@`, e.g. `HEAD^@`

A suffix `^` followed by an at sign is the same as listing all parents of `<rev>` (meaning, include anything reachable from its parents, but not the commit itself).

`<rev>^!`, e.g. `HEAD^!`

A suffix ^ followed by an exclamation mark is the same as giving commit <rev> and all its parents prefixed with ^ to exclude them (and their ancestors).

<rev>^-<n>, e.g. HEAD^-, HEAD^-2

Equivalent to <rev>^<n>..<<rev>, with <n> = 1 if not given.

Here are a handful of examples using the Loeliger illustration above, with each step in the notation's expansion and selection carefully spelt out:

Args	Expanded arguments	Selected commits
------	--------------------	------------------

D	G H D	
---	-------	--

D F	G H I J D F	
-----	-------------	--

^G D	H D	
------	-----	--

^D B	E I J F B	
------	-----------	--

^D B C	E I J F B C	
--------	-------------	--

C	I J F C	
---	---------	--

B..C = ^B C	C	
-------------	---	--

B...C = B ^F C	G H D E B C	
----------------	-------------	--

B^- = B^..B		
-------------	--	--

= ^B^1 B	E I J F B	
----------	-----------	--

C^@ = C^1		
-----------	--	--

= F	I J F	
-----	-------	--

B^@ = B^1 B^2 B^3		
-------------------	--	--

= D E F	D G H E F I J	
---------	---------------	--

C^! = C ^C^@		
--------------	--	--

= C ^C^1		
----------	--	--

= C ^F	C	
--------	---	--

B^! = B ^B^@		
--------------	--	--

= B ^B^1 ^B^2 ^B^3		
--------------------	--	--

= B ^D ^E ^F	B	
--------------	---	--

F^! D = F ^I ^J D	G H D F	
-------------------	---------	--

PARSEOPT

In --parseopt mode, git rev-parse helps massaging options to bring to shell scripts the same facilities C builtins have. It works as an option normalizer (e.g. splits single switches aggregate values), a bit like getopt(1) does.

It takes on the standard input the specification of the options to parse and understand, and

echoes on the standard output a string suitable for `sh(1)` `eval` to replace the arguments with normalized ones. In case of error, it outputs usage on the standard error stream, and exits with code 129.

Note: Make sure you quote the result when passing it to `eval`. See below for an example.

Input Format

`git rev-parse --parseopt` input format is fully text based. It has two parts, separated by a line that contains only `--`. The lines before the separator (should be one or more) are used for the usage. The lines after the separator describe the options.

Each line of options has this format:

`<opt-spec><flags>*<arg-hint>? SP+ help LF`

`<opt-spec>`

its format is the short option character, then the long option name separated by a comma. Both parts are not required, though at least one is necessary. May not contain any of the `<flags>` characters. `h,help`, `dry-run` and `f` are examples of correct

`<opt-spec>`.

`<flags>`

`<flags>` are of `*`, `=`, `?` or `!`.

? Use `=` if the option takes an argument.

? Use `?` to mean that the option takes an optional argument. You probably want to use the `--stuck-long` mode to be able to unambiguously parse the optional argument.

? Use `*` to mean that this option should not be listed in the usage generated for the `-h` argument. It's shown for `--help-all` as documented in `gitcli(7)`.

? Use `!` to not make the corresponding negated long option available.

`<arg-hint>`

`<arg-hint>`, if specified, is used as a name of the argument in the help output, for options that take arguments. `<arg-hint>` is terminated by the first whitespace. It is customary to use a dash to separate words in a multi-word argument hint.

The remainder of the line, after stripping the spaces, is used as the help associated with the option.

Blank lines are ignored, and lines that don't match this specification are used as option group headers (start the line with a space to create such lines on purpose).

Example

```
OPTS_SPEC="\
```

some-command [<options>] <args>...

some-command does foo and bar!

--

-h,help! show the help

foo some nifty option --foo

bar= some cool option --bar with an argument

baz=arg another cool option --baz with a named argument

qux?path qux may take a path argument but has meaning by itself

An option group Header

C? option C with an optional argument"

eval "\$(echo "\$OPTS_SPEC" | git rev-parse --parseopt -- "\$@" || echo exit \$?)"

Usage text

When "\$@" is -h or --help in the above example, the following usage text would be shown:

usage: some-command [<options>] <args>...

some-command does foo and bar!

-h, --help show the help

--[no-]foo some nifty option --foo

--[no-]bar ... some cool option --bar with an argument

--[no-]baz <arg> another cool option --baz with a named argument

--[no-]qux[=<path>] qux may take a path argument but has meaning by itself

An option group Header

-C[...] option C with an optional argument

SQ-QUOTE

In --sq-quote mode, git rev-parse echoes on the standard output a single line suitable for sh(1) eval. This line is made by normalizing the arguments following --sq-quote. Nothing other than quoting the arguments is done.

If you want command input to still be interpreted as usual by git rev-parse before the output is shell quoted, see the --sq option.

Example

```
$ cat >your-git-script.sh <<\EOF
```

```
#!/bin/sh
```

```
args=$(git rev-parse --sq-quote "$@") # quote user-supplied arguments
```

```
command="git froz -n24 $args" # and use it inside a handcrafted
```

command line

```
eval "$command"
```

EOF

```
$ sh your-git-script.sh "a b'c"
```

EXAMPLES

? Print the object name of the current commit:

```
$ git rev-parse --verify HEAD
```

? Print the commit object name from the revision in the \$REV shell variable:

```
$ git rev-parse --verify --end-of-options $REV^{commit}
```

This will error out if \$REV is empty or not a valid revision.

? Similar to above:

```
$ git rev-parse --default master --verify --end-of-options $REV
```

but if \$REV is empty, the commit object name from master will be printed.

GIT

Part of the git(1) suite

Git 2.53.0

03/02/2026

GIT-REV-PARSE(1)