Borg User Manual

for version 3.3.1.50-git

Jonas Bernoulli

Table of Contents

1 Introduction

The Borg assimilate Emacs packages as Git submodules.

Borg is a bare-bones package manager for Emacs packages. It provides only a few essential features and should be combined with other tools such as Magit, `epkg`, `use-package`, and `auto-compile`.

Borg assimilates packages into the `~/.config/emacs`[1] repository as Git submodules. An assimilated package is called a drone and a borg-based `~/.config/emacs` repository is called a collective.

It is possible to clone a package repository without assimilating it. A cloned package is called a clone.

To learn more about this project, also read the blog post[2] in which it was announced.

[1] Or `~/.emacs.d` of course, if you prefer that or have to use the old location because you still have to support older Emacs releases.

[2] `https://emacsair.me/2016/05/17/assimilate-emacs-packages-as-git-submodules`.

2 Installation

Borg can be used by itself, in which case it has to be bootstrapped. This is how Borg originally was intended to be used exclusively, but nowadays you may also choose to use it merely as a secondary package manager, in which case Borg itself is installed using Package.

2.1 Use as secondary package manager

To use Borg as a secondary package manager alongside Package, begin by installing the former using the latter. Borg is only available from Melpa, so you have to add that first.

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

Then you have to `M-x package-refresh-contents RET` before you can `M-x package-install RET borg RET`. Doing that should add a verbose variant of this to your init file:

```
(custom-set-variables
 '(package-selected-packages '(borg)))
```

Then you have to make the two package managers aware of each other by replacing the call to `package-initialize` in your init file with this:

```
(if (require 'borg-elpa nil t)
    (borg-elpa-initialize)
  (package-initialize))
```

Just like Package (aka Elpa) defaults to installing packages in `elpa/`, Borg defaults to installing them in `borg/`. (When using both package managers, that is. If used by itself, then Borg defaults to using `lib/`.) If you want to use another directory, then you can do so by setting the Git variable `borg.drones-directory`.

You should also add a `Makefile` containing:

```
BORG_SECONDARY_P = true
include $(shell find -L elpa -maxdepth 1 -regex '.*/borg-[.0-9]*' |\
  sort | tail -n 1)/borg.mk
```

Despite its title, many things mentioned in the next section are relevant even when Borg was installed the way we just did. Just saying.

2.2 Bootstrapping

(As mentioned in the next section you can use a location other than `~/.config/emacs` (or `~/.emacs.d`). Nevertheless most of the subsequent examples just talk about `~/.config/emacs` and if you use something else, then obviously have to substitute that. The same applies to the string `lib` and the variables `user-init-file` and `user-emacs-directory`, which also might not be appropriate depending on your choices.)

2.2.1 Bootstrapping using a seed

To get started clone the repository of the `emacs.g` collective. A "collective" is a starter-kit and/or configuration seed that relies on Borg as the package manager. Most users end up using `emacs.g` merely as a bootstrapping seed and do not merge upstream changes after that.

This collective already assimilates a few drones in addition to `borg` itself, namely `magit`, `epkg`, `use-package`, `auto-compile`, `git-modes` and `diff-hl`, as well as their dependencies. These drones are not required by `borg` but their use is highly recommended.

Clone the `emacs.g` repository to either `~/.config/emacs`, or for testing purposes to any other location. This repository contains a `Makefile` that imports `lib/borg/borg.mk` and defines an additional target whose purpose is to make that file and `lib/borg/borg.sh` available. Run `make bootstrap-borg` to clone the `borg` repository. That does not completely setup `borg` but it makes the latest version of the mentioned files available.

Now that these files are available you can run `make bootstrap` to get and configure all submodules (including the `borg` submodule) and to build all drones.

```
git clone git@github.com:emacscollective/emacs.g.git ~/.config/emacs
cd ~/.config/emacs
make bootstrap-borg
make bootstrap
```

If you have assimilated many packages, you might want to use `make bootstrap | tee bootstrap.log`.

The last command run during bootstrap is `git submodule status`, which prints one line per module. If a line is prefixed with '`+`', that means that it was not possible to checkout the recorded commit, and `-` means that the module could not be cloned. Even if some module could not be cloned, that usually does not render a configuration unusable, so just run `emacs` now, and then investigate any issues from the comfort of Magit.

If you cloned to somewhere other than `~/.config/emacs`, then you can use that configuration using `emacs --init-directory /path/to/emacs.g/`, provided you are using Emacs 29 or later. Otherwise you have to resort to `emacs -Q -l /path/to/emacs.g/early-init.el -l /path/to/emacs.g/init.el`.

For drones whose upstream repositories are located on Github or Gitlab the `emacs.g` collective uses the `ssh` protocol by default, which is a problem if you don't have accounts there and have not properly setup your keys. See Section 2.2.5 [Using https URLs], page 5.

During package compilation you may notice the submodules relating to those packages become dirty due to the compilation outputs not being ignored in those submodules. For this reason it is useful to ignore these outputs globally, for example in your `~/.config/git/ignore` file:

```
*.elc
*-autoloads.el
dir
```

You may discover more things that you'll want to ignore this way as you use `borg`.

2.2.2 Bootstrapping from scratch

If you don't want to base your configuration on the `emacs.g` starter-kit described in the previous section, then you have to do a few things manually.

```
git init ~/.config/emacs
cd ~/.config/emacs
```

By default Borg installs packages inside the `lib/` subdirectory, but since you are starting from scratch, you may choose something else by setting the Git variable `borg.drones-directory` locally for this repository.

Then you should add a `Makefile` containing at least:

```
DRONES_DIR = $(shell git config "borg.drones-directory" || echo "lib")

-include $(DRONES_DIR)/borg/borg.mk

bootstrap-borg:
        @git submodule--helper clone --name borg --path $(DRONES_DIR)/borg \
        --url git@github.com:emacscollective/borg.git
        @cd $(DRONES_DIR)/borg; git symbolic-ref HEAD refs/heads/main
        @cd $(DRONES_DIR)/borg; git reset --hard HEAD
```

Now you are probably tempted to run `make bootstrap-borg`, but that is for bootstrapping *from a seed*, and what we are doing right now is to bootstrap *from scratch*. In the process we are creating a seed but we are not there yet. Instead run this:

```
git submodule add --name borg git@github.com:emacscollective/borg.git lib/borg
```

Now that `borg` is available we can build all the assimilated packages (currently just `borg` itself) using `make bootstrap`.

Now it is time to tell Emacs to initialize Borg instead of Package by adding a simple `init.el` file containing at least:

```
(when (< emacs-major-version 27)
  (setq package-enable-at-startup nil))
(add-to-list 'load-path (expand-file-name "lib/borg" user-emacs-directory))
(require 'borg)
(borg-initialize)
```

Beginning with Emacs 27.1, `package-enable-at-startup` has to be disabled earlier, in `early-init.el`:

```
(setq package-enable-at-startup nil)
```

Now you could create the initial commit but you could also delay that.

```
git commit -m "Assimilate borg"
```

Now it is time to assimilate some other essential packages. You could do so using `M-x borg-assimilate`, but you would quickly notice that doing so without the help of the `epkg` package is quite cumbersome, so lets manually install that and its dependency first:

```
git submodule add --name closql git@github.com:emacscollective/closql.git lib/closql
git submodule add --name emacsql git@github.com:skeeto/emacsql.git lib/emacsql
git submodule add --name compat https://git.sr.ht/~pkal/compat lib/compat
git submodule add --name llama https://git.sr.ht/~tarsius/llama lib/llama
git submodule add --name epkg git@github.com:emacscollective/epkg.git lib/epkg
git config -f .gitmodules submodule.emacsql.no-byte-compile emacsql-pg.el
echo /epkgs/ >> .gitignore
git add .gitignore .gitmodules
make build
git commit -m "Assimilate epkg and dependencies"
```

Once you have done that and have restarted Emacs, you can install Magit using Borg, as described in Chapter 4 [Assimilation], page 7. You should also configure Magit status buffers to display submodules:

```
(with-eval-after-load 'magit
  (magit-add-section-hook 'magit-status-sections-hook
                          'magit-insert-modules
                          'magit-insert-stashes
                          'append))
```

Finally (look, nobody forced you to do this from scratch ;-) I strongly suggest that you make yourself familiar with my `auto-compile` package.

2.2.3 Migrating a legacy configuration

If you are currently using Package and want to gently ease into using Borg alongside that, then you can proceed as described in Section 2.1 [Use as secondary package manager], page 2.

If on the other hand you are already manually using Git modules, then you should proceed as described in Section 2.2.2 [Bootstrapping from scratch], page 3. Obviously "from scratch" does not apply this time around, so just skip steps like `git init`.

2.2.4 Using your configuration on another machine

Getting started using your existing configuration on another machine works the same way as described in Section 2.2.1 [Bootstrapping using a seed], page 2. The only difference is that instead of starting by cloning someone else's repository you start by cloning your own repository.

2.2.5 Using https URLs

For drones whose upstream repositories are located on Github or Gitlab the `emacs.g` collective uses the `ssh` protocol by default, which is a problem if you don't have accounts there and have not properly setup your keys.

Luckily this can easily be fixed using the following global rules.

```
git config --global url.https://github.com/.insteadOf git@github.com:
git config --global url.https://gitlab.com/.insteadOf git@gitlab.com:
```

If you don't want to configure this globally, then you can also configure Borg itself to prefer the `https` URLS.

```
(setq borg-rewrite-urls-alist
      '(("git@github.com:" . "https://github.com/")
        ("git@gitlab.com:" . "https://gitlab.com/")))
```

This does not affect packages that have already been assimilated. During bootstrapping you have to change the URLs for packages that are assimilated by default.

```
cd ~/.config/emacs
sed -i "s|git@github.com:|https://github.com/|g" .gitmodules
sed -i "s|git@gitlab.com:|https://gitlab.com/|g" .gitmodules
git commit -m "Use https URLs for Github and Gitlab"
```

If you have already run `make bootstrap`, then you also have to edit `.git/config`.

```
cd ~/.config/emacs
sed -i "s|git@github.com:|https://github.com/|g" .git/config
sed -i "s|git@gitlab.com:|https://gitlab.com/|g" .git/config
```

3 Startup

The `user-init-file`, `~/.config/emacs/init.el`, has to contain a call to `borg-initialize`.

You should also set `package-enable-at-startup` to `nil`. If you use Emacs 27 or later, then do so in `early-init.el`, otherwise in `init.el`.

`borg-initialize` [Function]

This function initializes assimilated drones using `borg-activate`.

To skip the activation of the drone named DRONE, temporarily disable it by setting the value of the Git variable `submodule.DRONE.disabled` to true in `~/.config/emacs/.gitmodules`.

`borg-activate` *clone* [Command]

This function activates the clone named CLONE by adding the appropriate directories to the `load-path` and to `Info-directory-list`, and by loading the autoloads file, if it exists.

Unlike `borg-initialize`, this function ignores the Git variable `submodule.DRONE.disabled` and can be used to activate clones that have not been assimilated.

4 Assimilation

A third-party package is assimilated by adding it as a submodule and, if necessary, by configuring it in `~/.config/emacs/init.el`. Built-in packages are assimilated merely by configuring them.

To begin the assimilation of a third-party package use the command `borg-assimilate`, which adds the package's repository as a submodule and attempts to build the drone.

A safer alternative is to first clone the package without assimilating it, using `borg-clone`. This gives you an opportunity to inspect the cloned package for broken or malicious code, before it gets a chance to run arbitrary code. Later you can proceed with the assimilation using `borg-assimilate`, or remove the clone using `borg-remove`.

Building the drone can fail, for example due to missing dependencies. Failure to build a drone is not considered as a failure to assimilate. If a build fails, then a buffer containing information about the issue pops up. If the failure is due to unsatisfied dependencies, then assimilate those too, and then build any drone, which previously couldn't be built, by using the Emacs command `borg-build` or `make lib/DRONE`. Alternatively you can just rebuild everything using `make build`.

If you wish to avoid such complications, you should use the command `epkg-describe-package` before assimilating a package. Among other useful information, it also provides a dependency tree.

Once the packages have been added as a submodules and the drones have been built, the assimilation is completed by creating an assimilation commit.

If you assimilate a single package, then it is recommended that you use a message similar to this:

```
Assimilate foo v1.0.0
```

Or if one or more dependencies had to be assimilated, something like:

```
Assimilate foo and dependencies

Assimilate foo v1.0.0
Assimilate bar v1.1.0
Assimilate baz v0.1.0
```

It's usually a good idea not to assimilate unrelated packages in the same commit, but doing it for related packages, which do not strictly depend on one another, it might make sense:

```
Assimilate ido and extensions

Assimilate flx               v0.6.1-3-gae0981b
Assimilate ido-at-point      v1.0.0
Assimilate ido-ubiquitous    v3.12-2-g7354d98
Assimilate ido-vertical-mode v0.1.6-33-gb42e422
Assimilate smex               3.0-13-g55aaebe
```

The command `borg-insert-update-message` can be used to generate such commit messages.

`C-c C-b [in git-commit-mode buffer]` (`borg-insert-update-message`)
: This command insert information about drones that are changed in the index. Formatting is according to the commit message conventions described above.

`borg-assimilate` *package url* **&optional** *partially* [Command]
: This command assimilates the package named PACKAGE from URL.

  If `epkg` is available, then only the name of the package is read in the minibuffer and the url stored in the Epkg database is used. If `epkg` is unavailable, the package is not in the database, or if a prefix argument is used, then the url too is read in the minibuffer.

  If a negative prefix argument is used, then the submodule is added but the build and activation steps are skipped. This is useful when assimilating a package that requires special build steps. After configuring the build steps use `borg-build` to complete the assimilation.

`borg-clone` *package url* [Command]
: This command clones the package named PACKAGE from URL, without assimilating it. This is useful when you want to inspect the package before potentially executing malicious or broken code.

  Interactively, when the `epkg` package is available, then the name is read in the minibuffer and the url stored in the Epkg database is used. If `epkg` is unavailable, the package is unknown, or when a prefix argument is used, then the url is also read in the minibuffer.

`borg-remove` *clone* [Command]
: This command removes the cloned or assimilated package named CLONE, by removing the working tree from `borg-drones-directory`, regardless of whether that repository belongs to an assimilated package or a package that has only been cloned for review using `borg-clone`. The Git directory is not removed.

`borg-build` *clone* **&optional** *activate* [Command]
: This command builds the clone named CLONE. Interactively, or when optional ACTIVATE is non-nil, then also activate the drone using `borg-activate`.

`borg-update-autoloads` *clone* **&optional** *path* [Function]
: This function updates the autoload file for the libraries belonging to the clone named CLONE in the directories in PATH. PATH can be omitted or contain file-names that are relative to the top-level of CLONE's repository.

`borg-compile` *clone* **&optional** *path* [Function]
: This function compiles the libraries for the clone named CLONE in the directories in PATH. PATH can be omitted or contain file-names that are relative to the top-level of CLONE's repository.

`borg-maketexi` *clone* **&optional** *files* [Function]
: This function generates Texinfo files from certain Org files for the clone named CLONE. Org files that are located on `borg-info-path` are exported if their names match `borg-maketexi-filename-regexp` and the `TEXINFO_DIR_HEADER` export keyword is set in their content.

`borg-makeinfo` *clone* [Function]
This function generates the Info manuals and the Info index for the clone named CLONE.

`borg-batch-rebuild` **&optional** *quick* [Function]
This function rebuilds all assimilated drones in alphabetic order, except for Org which is rebuilt first. After that it also builds the user init files using `borg-batch-rebuild-init`.

This function is not intended for interactive use; instead it is used by the Make target `build` described in the following section.

When optional QUICK is non-nil, then this function does not build drones for which `submodule.DRONE.build-step` is set, assuming that those are the drones that take longer to be built.

`borg-batch-rebuild-init` [Function]
This function builds the init files specified by the Make variable `INIT_FILES`, or if that is unspecified `init.el` and `LOGIN.el`, where `LOGIN` is the value of the variable `user-real-login-name`. If a file does not exist, then it is silently ignored.

This function is not intended for interactive use; instead it is used by the Make targets `build-init` and (indirectly) `build`, which are described in Chapter 7 [Make targets], page 12.

5 Updating drones

Borg does not provide an update command. By not doing so, it empowers you to update to exactly the commit you wish to update to, instead of to "the" new version.

To determine the drones that you *might* want to update, visit the Magit status buffer of the `~/.config/emacs` repository and press `f m` to fetch inside all submodules. After you have done so, and provided there actually are any modules with new upstream commits, a section titled "Modules unpulled from @{upstream}" appears.

Each subsection of that section represents a submodule with new upstream commits. Expanding such a subsection lists the new upstream commits. These commits can be visited by pressing `RET`, and the status buffer of a submodule can be visited by pressing `RET` while point is inside the heading of the respective submodule section. To return to the status buffer of `~/.config/emacs` press `q`.

Inside the status buffer of a submodule, you can pull the upstream changes as usual, using `F u`. If you wish you can inspect the changes before doing so. And you can also choose to check out another commit instead of the upstream `HEAD`.

Once you have "updated" to a new commit, you should also rebuild the drone using the command `borg-build`. This may fail, e.g., due to new dependencies.

Once you have resolved all issues, you should create an "update commit". You can either create one commit per updated drone or you can create a single commit for all updated drones, which ever you find more appropriate. However it is recommended that you use a message similar to:

```
Update foo to v1.1.0
```

Or for multiple packages:

```
Update 2 drones

Update foo to v1.1.0
Update bar to v1.2.1
```

The command `borg-insert-update-message` can be used to generate such commit messages.

To update the Epkg package database use the command `epkg-update`.

6 Patching drones

By using Borg you can not only make changes to assimilated packages, you can also keep track of those patches and share them with others.

If you created some commits in a drone repository and are the maintainer of the respective package, then you can just push your changes to the "origin" remote.

You don't have to do this every time you created some commits, but at important checkpoints, such as after creating a release, you should record the changes in the `~/.config/emacs` repository. To do so proceed as described in Chapter 5 [Updating drones], page 10.

But for most packages you are not the maintainer and if you create commits for such drones, then you have to create a fork and push there instead. You should configure that remote as the push-remote using `git config remote.pushDefault FORK`, or by pressing `b C M-p` in Magit. After you have done that, you can continue to pull from the upstream using `F u` in Magit and you can also push to your fork using `P p`.

Of course you should also occasionally record the changes in the `~/.config/emacs` repository. Additionally, and ideally when you first fork a drone, you should also record information about your personal remote in the super-repository by setting `submodule.DRONE.remote` in `~/.config/emacs/.gitmodules`.

`submodule.DRONE.remote "`*NAME URL*`"` [Variable]

This variable specifies an additional remote named NAME that is fetched from URL. This variable can be specified multiple times. Note that "NAME URL" is a single value and that the two parts of that value are separated by a single space.

`make bootstrap` automatically adds all remotes that are specified like this to the DRONE repository by setting `remote.NAME.url` to URL and using the standard value for `remote.NAME.fetch`.

`borg.pushDefault` *FORK* [Variable]

This variable specifies a name used for push-remotes. Because this variable can only have one value it is recommended that you use the same name, FORK, for your personal remote in all drone repositories in which you have created patches that haven't been merged into the upstream repository (yet). A good value may be your username.

For all DRONES for which one value of `submodule.DRONE.remote` specifies a remote whose NAME matches FORK, `make bootstrap` automatically configures FORK to be used as the push-remote by setting `remote.pushDefault` to FORK.

7 Make targets

The following `make` targets are available by default. To use them you have to be in `~/.config/emacs` in a shell. They are implemented in `borg.mk`, which is part of the `borg` package.

Help targets

`help` [Command]

This target prints information about most of the following targets.

This is the default target, unless the user sets `.DEFAULT_GOAL`.

`helpall` [Command]

This target prints information about all of the following targets.

Batch targets

`clean` [Command]

This target removes all byte-code and native files of all drones and config files.

To ensure a clean build, this target should always be run before `build`, so you might want to add a default target that does just that. To do so, add this to `~/.config/emacs/etc/borg/config.mk`:

```
.DEFAULT_GOAL := all
all: clean build
```

`clean-force` [Command]

This target removes all byte-code files using `find`. The `clean` target on the other hand uses the lisp function `borg--batch-clean`. Byte-code isn't always compatible between Emacs releases and this target makes it possible to recover from such an incompatibility.

`build` [Command]

This target byte-compiles Borg and Compat first, followed by all other drones in alphabetic order. After that it also byte-compiles the user init files, like `build-init` does.

`native` [Command]

This target byte-compiles and natively compiles Borg and Compat first, followed by all other drones in alphabetic order. After that it also byte-compiles the user init files, like `build-init` does.

Quick batch targets

These targets act on most drones but exclude those for which the Git variable `submodule.DRONE.build-step` is set. The assumption is that those are the drones that take longer to build.

`quick` [Command]
: This target cleans and builds most drones.

  It also cleans and builds the init files as described as for `build`.

`quick-clean` [Command]
: This target removes all byte-code and native files of most drones It also remove the byte-code files of the config files.

`quick-build` [Command]
: This target builds *most* drones and the config files

  It also builds the init files as described as for `build`.

Drone targets

`clean/DRONE` [Command]
: This target removes all byte-code and native files belonging to the drone named DRONE.

`build/DRONE` [Command]
: This target byte-compiles the drone named DRONE.

  `lib/DRONE` is an alias for this target; or rather `DIR/DRONE`, where `DIR` is directory containing the drone submodules.

`native/DRONE` [Command]
: This target byte-compiles and natively-compiles the drone named DRONE.

Init file targets

`init-clean` [Command]
: This target removes byte-code files for init files.

`init-tangle` [Command]
: This target tangles (creates) `init.el` from `init.org`. You obviously don't have to use such a file if you don't want to.

`init-build` [Command]
: This target byte-compiles the init files specified by the make variable `INIT_FILES`; or if that is unspecified `init.el` and `LOGIN.el` (where `LOGIN` is the value of the variable `user-real-login-name`). If an init file does not exist, then that is silently ignored.

  If you publish your `~/.config/emacs` repository but would like to keep some settings private, then you can do so by putting them in a file `~/.config/emacs/LOGIN.el`. The downside of this approach is that you will have to somehow synchronize that file between your machines without checking it into Git.

Bootstrap targets

`bootstrap-borg` [Command]

This target bootstraps `borg` itself.

`bootstrap` [Command]

This target attempts to bootstrap the drones. To do so it runs `git submodule init`, `borg.sh` (which see), and `make build`.

If an error occurs during the `borg.sh` phase, then you can just run that command again to process the remaining drones. The drones that have already been bootstrapped or that have previously failed will be skipped. If a drone cannot be cloned from any of the known remotes, then you should temporarily remove it using `git submodule deinit lib/DRONE`. When done with `borg.sh` also manually run `make build` again.

8 Variables

`borg.drones-directory` *DIRECTORY* [Variable]

This Git variable can be used to override the name of the directory that contains the drone submodules. If specified, the value has to be relative to the top-level directory of the repository.

Note that if you change the value of this variable, then you might have to additionally edit `~/.config/emacs/Makefile`.

The values of the following variables are set at startup and should not be changed by the user.

`borg-drones-directory` [Variable]

The value of this constant is the directory beneath which drone submodules are placed. The value is set based on the location of the `borg` library and the Git variable `borg.drones-directory` if set, and should not be changed.

`borg-user-emacs-directory` [Variable]

The value of this constant is the directory beneath which additional per-user Emacs-specific files are placed. The value is set based on the location of the `borg` library and should not be changed. The value is usually the same as that of `user-emacs-directory`, except when Emacs is started with `emacs -q -l /path/to/init.el`.

`borg-gitmodules-file` [Variable]

The value of this constant is the `.gitmodules` file of the super-repository.

The value of this Make variable has to be set in `~/.config/emacs/Makefile`.

`INIT_FILES` [Variable]

A list of init files to be build by the Make targets `build` and `build-init`. See Chapter 7 [Make targets], page 12.

The values of these borg-specific Git variables have to be set in the file `~/.config/emacs/.gitmodules`. The variables `borg.pushDefault` and `submodule.DRONE.remote` are described in Chapter 6 [Patching drones], page 11.

Because most repositories used to maintain Emacs packages follow some common-sense conventions, Borg usually does not have to be told how to build a given drone. Building is done using `borg-build`, which in turn usually does its work using `borg-update-autoloads`, `borg-compile`, `borg-maketexi`, and `borg-makeinfo`.

However some packages don't follow the conventions either because they are too complex to do so, or for the sake of doing it differently. But in either case resistance is futile; by using the following variables you can tell Borg how to build such packages.

`submodule.DRONE.build-step` *COMMAND* [Variable]

By default drones are built using the lisp functions `borg-update-autoloads`, `borg-compile`, `borg-maketexi`, and `borg-makeinfo`, but if this variable has one or more values, then DRONE is built using these COMMANDs **instead**.

Each COMMAND can be one of the default steps, an S-expression, or a shell command. The COMMANDs are executed in the specified order.

If a COMMAND matches one of default steps, then it is evaluated with the appropriate arguments.

If a COMMAND begins with a parenthesis, then it is evaluated as a lisp expression. In that case the variable `borg-clone` holds the name of the package that is being build. To make a function available in this context, you usually have to define or load it in '`etc/borg/config.el`', relative to `borg-user-emacs-directory`.

Otherwise COMMAND is assumed to be a shell command and is executed with `shell-command`.

```
[submodule "mu4e"]
        path = lib/mu4e
        url = git@github.com:djcb/mu.git
        build-step = test -e ./configure || autoreconf -i
        build-step = ./configure
        build-step = make -C mu4e > /dev/null
        build-step = borg-update-autoloads
        load-path = mu4e
```

To skip generating "autoloads" (e.g., using `use-package` to create "autoloads" on the fly), just provide the required build steps to build the package, omitting `borg-update-autoloads`. Borg silently ignores a missing "autoloads" file during initialization (`borg-initialize`).

```
[submodule "multiple-cursors"]
        path = lib/multiple-cursors
        url = git@github.com:magnars/multiple-cursors.el.git
        build-step = borg-compile
```

Note that just because a package provides a `Makefile`, you do not necessarily have to use it.

Even if `make` generates the Info file, you might still have to add `borg-makeinfo` as an additional build-step because the former might not generate an Info index file (named `dir`), which Borg relies on.

Also see `borg.extra-build-step` below.

`borg-build-shell-command` [Variable]

This variable can be used to change how shell commands specified by `submodule.DRONE.build-step` are run. The default value is `nil`, meaning that each build step is run unchanged using `shell-command`.

If the value is a string, then that is combined with each build step in turn and the results are run using `shell-command`. This string must contain either %s, which is replaced with the unchanged build step, or %S, which is replaced with the result of quoting the build step using `shell-quote-argument`.

If the value is a function, then that is called once with the drone as argument and must return either a string or a function. If the returned value is a string, then that is used as described above.

If the value returned by the first function is another function, then this second function is called for each build step with the drone and the build step as arguments. It must

return a string or `nil`. If the returned value is a string, then that is used as described above.

Finally the second function may execute the build step at its own discretion and return `nil` to indicate that it has done so.

Notice that if the value of this variable is a function, this function must a) be defined in a drone; and b) be registered as an autoload. This is because build happens in a separate Emacs process started with `-Q --batch`, which only receives the name of the function.

`submodule.DRONE.load-path` *PATH* [Variable]

This variable instructs `borg-activate` to add PATH to the `load-path` instead of the directory it would otherwise have added. Likewise it instructs `borg-compile` to compile the libraries in that directory. PATH has to be relative to the top-level of the repository of the drone named DRONE. This variable can be specified multiple times.

Normally Borg uses `elisp/` as the drone's `load-path`, if that exists; otherwise `lisp/`, if that exists; or else the top-level directory. If this variable is set, then it *overrides* the default location. Therefore, to *add* an additional directory, you also have to explicitly specify the default location.

```
[submodule "org"]
        path = lib/org
        url = git://orgmode.org/org-mode.git
        build-step = make
        load-path = lisp
        load-path = contrib/lisp
        info-path = doc
```

`submodule.DRONE.no-byte-compile` *PATH* [Variable]

This variable instructs `borg-compile` to not compile the library at PATH. PATH has to be relative to the top-level of the repository of the drone named DRONE. This variable can be specified multiple times.

Sometimes a drone comes with an optional library which adds support for some other third-party package, which you don't want to use. For example `emacsql` comes with a PostgreSQL back-end, which is implemented in the library `emacsql-pg.el`, which requires the `pg` package. The standard Borg collective `emacs.g` assimilates `emacsql`, for the sake of the `epkg` drone, which only requires the SQLite back-end. To avoid an error about `pg` not being available, `emacs.g` instructs Borg to not compile `emacsql-pg.el`. (Of course if you want to use the PostgreSQL back-end and assimilate `pg`, then you should undo that.)

`submodule.DRONE.recursive-byte-compile` *BOOLEAN* [Variable]

Setting this variable to `true` instructs `borg-compile` to compile DRONE's directories recursively. This isn't done by default because there are more repositories in which doing so would cause issues than there are repositories that would benefit from doing so.

Unfortunately many packages put problematic test files or (usually outdated) copies of third-party libraries into subdirectories. The latter is a highly questionable thing

to do, but the former would be perfectly fine, if only the non-library lisp files did not provide a feature (which effectively turns them into libraries) and/or if a file named `.nosearch` existed in the subdirectory. That file tells functions such as `normal-top-level-add-subdirs-to-load-path` and `borg-compile` to ignore the containing directory.

`borg-compile-function` [Variable]
The function used to compile each individual library. One of `byte-compile-file`, `borg-byte+native-compile` or `borg-byte+native-compile-async`.

To enable native compilation when running `make`, use one of the respective make targets, as described in Chapter 7 [Make targets], page 12.

`borg-compile-recursively` [Variable]
Setting this variable to a non-nil value instructs `borg-compile` to compile all drones recursively. Doing so is discouraged.

`borg-native-compile-deny-list` [Variable]
This variable lists the names of files to be excluded from native compilation.

`borg.extra-build-step` *COMMAND* [Variable]
This variable instructs Borg to execute COMMAND after the default build-steps for each DRONE (or after `submodule.DRONE.build-step`, if that specified). It has to be set in `borg-gitmodules-file` and can have multiple values.

If a COMMAND begins with a parenthesis, then it is evaluated as a lisp expression. In that case the variable `borg-clone` holds the name of the package that is being build. To make a function available in this context, you usually have to define or load it in '`etc/borg/config.el`', relative to `borg-user-emacs-directory`.

Otherwise COMMAND is assumed to be a shell command and is executed with `shell-command`.

`submodule.DRONE.info-path` *PATH* [Variable]
This variable instructs `borg-initialize` to add PATH to `Info-directory-list`. PATH has to be relative to the top-level of the repository of the drone named DRONE.

`submodule.DRONE.no-maketexi` *PATH* [Variable]
This variable instructs `borg-maketexi` to not create a Texinfo file for the Org file at PATH. PATH has to be relative to the top-level of the repository of the drone named DRONE. This variable can be specified multiple times.

`submodule.DRONE.no-makeinfo` *PATH* [Variable]
This variable instructs `borg-makeinfo` to not create an Info file for the Texinfo file at PATH. PATH has to be relative to the top-level of the repository of the drone named DRONE. This variable can be specified multiple times.

`submodule.DRONE.disabled` *true*|*false* [Variable]
If the value of this variable is `true`, then it is skipped by `borg-initialize`.

`borg-rewrite-urls-alist` [Variable]
An alist that can optionally be used to rewrite certain URLs. Each element has the form `(ORIG . BASE)`. Each URL that starts with ORIG is rewritten to start with BASE instead. See Section 2.2.5 [Using https URLs], page 5.

`borg-maketexi-filename-regexp` [Variable]

A regexp matching Org files that may be exported to Texinfo by `borg-maketexi`. The name of the clone is substituted for `%s`. Setting this to `nil` disables the export of any Org files.

9 Low-level functions

You normally should not have to use the following low-level functions directly. That being said, you might want to do so anyway if you build your own tools on top of Borg.

`borg-worktree` *clone* [Function]
This function returns the top-level of the working tree of the clone named CLONE.

`borg-gitdir` *clone* [Function]
This function returns the Git directory of the clone named CLONE.

It always returns `BORG-USER-EMACS-DIRECTORY/.git/modules/CLONE`, even when CLONE's Git directory is actually located inside the working tree.

`borg-do-drones` (*var* [*result*]) *body...*) [Macro]
This macro loop over drones. BODY is evaluated with VAR bound to each drone, in turn. Inside BODY variables set in `.gitmodules` are cached. Then RESULT is evaluated to get the return value, defaulting to nil.

`borg-get` *clone variable* **&optional** *all* [Function]
This function returns the value of the Git variable `submodule.CLONE.VARIABLE` defined in `~/.config/emacs/.gitmodules`. If optional ALL is non-nil, then it returns all values as a list.

`borg-get-all` *clone variable* [Function]
This function returns all values of the Git variable `submodule.CLONE.VARIABLE` defined in `~/.config/emacs/.gitmodules` as a list.

`borg-load-path` *clone* [Function]
This function returns the `load-path` for the clone named CLONE.

`borg-info-path` *clone* **&optional** *setup* [Function]
This function returns the `Info-directory-list` for the clone named CLONE.

If optional SETUP is non-nil, then it returns a list of directories containing `texi` and/or `info` files. Otherwise it returns a list of directories containing a file named `dir`.

`borg-dronep` *name* [Function]
This function returns non-nil if a drone named NAME exists.

If that is set in `.gitmodules`, then it returns the value of `submodule.NAME.path`, nil otherwise.

`borg-drones` **&optional** *include-variables* [Function]
This function returns a list of all assimilated drones.

The returned value is a list of the names of the assimilated drones, unless optional INCLUDE-VARIABLES is non-nil, in which case elements of the returned list have the form `(NAME . ALIST)`.

ALIST is an association list. Property names are symbols and correspond to a VARIABLE defined in the Borg repository's `.gitmodules` file as `submodule.NAME.VARIABLE`.

Each property value is either a string or a list of strings. If INCLUDE-VARIABLES is `raw` then all values are lists. Otherwise a property value is only a list if the corresponding property name is a member of `borg--multi-value-variables`. If a property name isn't a member of `borg--multi-value-variables` but it does have multiple values anyway, then it is undefined with value is included in the returned value.

`borg-clones` [Function]

This function returns a list of all cloned packages.

The returned value includes the names of all drones, as well as the names of all other repositories that are located directly inside `borg-drones-directory` but aren't tracked as submodules.

`borg-read-package` *prompt* **&optional** *edit-url* [Function]

This function reads a package name and the url of its upstream repository from the user, and returns them as a list.

When the `epkg` package is available, then the user is only prompted for the name of the package, and the upstream url is retrieved from the Epkg database. If the package isn't in the database then the url has to be provided by the user. If optional EDIT-URL is non-nil, then the url from the database, if any, is provided as initial input for the user to edit.

PROMPT is used when prompting for the package name.

`borg-read-clone` *prompt* [Function]

This function reads the name of a cloned package from the user.

There exist a few more functions, but those are considered to be internal and might therefore change in incompatible ways without that being noted in the change log.

`borg--maybe-absorb-gitdir` *pkg* [Function]

`borg--maybe-reuse-gitdir` *pkg* [Function]

`borg--restore-worktree` *pkg* [Function]

`borg--call-git` *pkg* **&rest** *args* [Function]

`borg--expand-load-path` *drone path* [Function]

`borg--sort-submodule-sections` [Function]

Appendix A Function and Command Index

B

C

H

I

N

Q

Appendix B Variable Index

B

I

S