

# Task Guide

## Table of Contents

1. Introduction .....	3
2. Use Tasks .....	4
2.1. list-* .....	4
2.1.1. list-commands .....	5
2.1.2. list-configuration .....	5
2.1.3. list-dependencies .....	5
2.1.4. list-exitstatus .....	6
2.1.5. list-options .....	6
2.1.6. list-parameters .....	7
2.1.7. list-scenarios .....	8
2.1.8. list-tasks .....	9
2.2. describe-* .....	11
2.2.1. describe-application .....	12
2.2.2. describe-command .....	12
2.2.3. describe-dependency .....	12
2.2.4. describe-element .....	13
2.2.5. describe-exitstatus .....	13
2.2.6. describe-option .....	13
2.2.7. describe-parameter .....	14
2.2.8. describe-scenario .....	14
2.2.9. describe-task .....	15
2.3. cloc-installation .....	16
2.3.1. Options .....	16
2.3.2. Examples .....	17
2.3.3. Requirements .....	17
2.4. execute-program .....	17
2.4.1. Examples .....	18
2.4.2. Requirements .....	18
2.5. manual .....	18
2.5.1. Requirements .....	19
2.6. repeat-scenario .....	19
2.6.1. Options .....	19
2.6.2. Examples .....	20
2.7. repeat-task .....	20
2.7.1. Examples .....	20
2.8. setting .....	21

2.9. start-browser	22
2.9.1. Requirements	22
2.10. start-pdf-viewer	22
2.10.1. Requirements	22
2.11. start-xterm	22
2.11.1. Examples	23
2.11.2. Requirements	23
2.12. statistics	23
2.12.1. Filters	23
2.12.2. Examples	24
2.13. validate-installation	25
2.13.1. Targets	26
2.14. wait	27
2.14.1. Options	27
2.14.2. Examples	28
3. Build Tasks	28
3.1. build-manual	28
3.1.1. Configuration	30
3.1.2. Manual Source	30
3.1.3. General Options	31
3.1.4. Target Options	32
3.1.5. Element List Filters	32
3.1.6. Application Filters	32
3.1.7. Element Filters	33
3.1.8. Examples	33
3.1.9. Task Requirements	34
3.1.10. Notes	34
3.2. build-mvn-site	35
3.2.1. Target Options	35
3.2.2. Filter Options	36
3.2.3. Maven Options	36
3.2.4. Examples	36
3.2.5. Site Files	37
3.2.6. Requirements	38
3.3. clean	38
3.3.1. Options	38
3.4. make-target-sets	39
3.4.1. Target Options	39
3.4.2. Filter Options	40
3.4.3. Maven Options	40
3.4.4. Examples	40

3.4.5. Site Files .....	41
3.4.6. Requirements .....	42
4. Development Tasks .....	42
4.1. build-cache .....	42
4.1.1. General Target Options .....	43
4.1.2. Targets .....	43
4.1.3. Requirements .....	44
4.1.4. Notes .....	44
4.2. build-help .....	44
4.2.1. Requirements .....	45
4.2.2. Notes .....	45
4.3. download-fw-tool .....	45
4.3.1. Options .....	45
4.4. set-file-versions .....	45
4.4.1. Examples .....	46
4.4.2. Requirements .....	46
4.4.3. The Build File .....	46
4.4.4. The Macro File .....	47

for SKB-Framework v0.0.4, May 27, 2019

# 1. Introduction

This guide provides details for all tasks of the framework. The tasks are described in three categories:

- Use Tasks - these tasks are mainly directed at the application mode *use*.
- Build Tasks - these tasks are mainly directed at the application mode *builde*. They do build artifacts or compile source files to build artifacts.
- Development Tasks - these tasks are mainly directed at the application mode *dev*. They provide functionality required to develop and application or to build important runtime artifacts for an application.

A task in the framework is essentially a *bash* script that benefits from provided functions of the framework, such as loading and testing dependencies or parameters. By convention, each task should provide an argument `-h` or `--help`, which shows its arguments. For simple tasks, the description in this guide does no go beyond what this help and the online help (or manual) already provides. One of those tasks is *wait*.

For more complex tasks, with either a lot of arguments or special configuration requirements, this guide provides all details required to use the task. One of those tasks is `build-mvn-site`, which not only has a rather complex process but also requires additional metadata for sites to exist.

For details on how to develop and write a task please see the developer guide. This document

focuses only on how to use tasks.

## 2. Use Tasks

This category of tasks provides functionality for using built artifacts or functionality that does not require a build. Those tasks, if they provide important functionality, might also be available in the application modes *build* and/or *dev*.

### 2.1. list-\*

There are eight tasks that start with *list-*. Each of these tasks provides a list of framework and application features, namely:

- *list-commands* - lists shell commands
- *list-configuration* - list current configuration keys and values
- *list-dependencies* - lists dependencies
- *list-exitstatus* - lists exit status, error codes
- *list-options* - lists CLI options
- *list-parameters* - lists parameters
- *list-scenarios* - lists scenarios
- *list-tasks* - lists tasks

All of these tasks have the same default behavior, provide the same main options, and can be used with (then task specific) filters. The default behavior, when executed without any arguments, is to provide a simple list. For example, when *list-tasks* is called it provides a simple list of tasks. For each item in the list, the tasks show the identifier (or key), if available a short identifier, and a description (e.g. for tasks) or values (e.g. for *list-configuration*). The list will be empty if no items where found, e.g. in *list-scenarios* when no scenarios have been loaded.

The standard options are shown below. Each of the list tasks provides a *table* view, which provides more detailed information for their items. The list tasks can also use a print mode different than the currently configured. *print-mode* can be set to any mode that is supported by the API, namely:

- *adoc* - print in AsciiDoc format
- *ansi* - print with ANSI formatting for colors and effects
- *text* - print as plain text, no special formatting will be used
- *text-anon* - print as text with some annotations, similar to *adoc*

```
-h | --help           print help screen and exit
-P | --print-mode MODE print mode: ansi, text, adoc
-T | --table         help screen format
```

All list tasks with filters provide a standard filter for listing all items, shown below. All other filters

are task specific. If filters are used, only items that satisfy the filter will be listed. Any number of filters can be used in any combination.

```
filters
-A | --all          all settings, disables all other filters
```

### 2.1.1. list-commands

List the shell commands. This task does not provide any further options or filters.

### 2.1.2. list-configuration

Lists configurations keys and values. For print mode *ansi*, some of the values are color-coded or printed in bold or italic, for example:

- Values for application mode: *use* printed in green, *build* printed in blue, *dev* printed in yellow, *all* printed in dark red
- Values for levels: *error* printed in red, *warn* printed in yellow, *info* printed in green, *debug* and *trace* printed in blue
- Value for flavor: printed in bold
- Values for quiet settings: *off* printed in green, *on* printed in red

In table mode, a color coded status indicates from where a setting was taken.

The task offers a number of filters. They all relate to the original source from which a configuration value was set: command line, environment, configuration file, default value, or internally.

```
filters
-c | --cli          only settings from CLI options
-d | --default      only settings from default value
-e | --env          only settings from environment
-f | --file         only settings from configuration file
-i | --internal     only internal settings
```

This task can take the list of items from the framework or application cache. If *CACHE\_DIR* is set to a directory that contains cached command information (map), this information is used.

### 2.1.3. list-dependencies

Lists dependencies with name and description. Table mode shows two more aspects of dependencies: the origin of its declaration (framework or application) and the load status (color coded).

Provided filters:

- *install* - only dependencies that are only required by tasks that run in *install* application mode flavor

- *origin* - filter by declaration origin
  - framework (as: **F**, **f**, **fw**, or **framework**), or
  - application (as: **A**, **a**, **app**, or **application**)
- *requested* - show dependencies that have been requested by a loaded task
- *status* - filter by dependency status
  - success (as: **S**, **s**, or **success**),
  - warnings (as: **W**, **w**, or **warning**),
  - errors (as: **E**, **e**, or **error**), or
  - not attempted (as: **N**, **n**, or **not-attempted**)
- *tested* - filter dependencies that have been tested, ignoring untested dependencies

```
filters
-I | --install      only dependencies required only by install tasks
-o | --origin ORIGIN  only dependencies from origin: f(w), a(pp)
-r | --requested    only requested dependencies
-s | --status STATUS  only dependencies with status: (s)uccess, (w)arning, (e)rror,
(n)ot attempted
-t | --tested       only tested dependencies
```

This task can take the list of items from the framework or application cache. If `CACHE_DIR` is set to a directory that contains cached dependency information (map), this information is used.

#### 2.1.4. list-exitstatus

Lists exit status codes with their number and description. Table mode shows two more aspects: the origin (of an error: all, application, framework, loader, shell, tasks) and an indicator of the problem (internal for a bug, external for a configuration problem). Provided filters focus on the *origin* of the problem.

```
filters
  --app          only application status
-f | --fw       only framework status
-l | --loader    only loader status
-s | --shell     only shell status
-t | --task      only task status
```

This task can take the list of items from the framework or application cache. If `CACHE_DIR` is set to a directory that contains cached exit status information (map), this information is used.

#### 2.1.5. list-options

Lists command line options with name, short name, if required a parameter, and a description. Table mode also shows the type of option, being either an *exit* option or a *runtime* option. Provided

filters focus on the option type.

```
filters
-e | --exit          only exit options
-r | --run           only runtime options
```

This task can take the list of items from the framework or application cache. If `CACHE_DIR` is set to a directory that contains cached option information (map), this information is used.

### 2.1.6. list-parameters

Lists parameters with their name and description. Table mode shows three more aspects of parameters: the origin of its declaration (framework or application), an indicator for a defined default value (red cross for not defined, green ok for defined), and a color coded load status.

The task also provides a second table. This table shows the parameter names with their declared default value.

```
options
-D | --def-table      print default value table
```

Provided filters are:

- *default* - show parameters with a set default value
- *install* - only parameters that are only required by tasks that run in `install` application mode flavor
- *origin* - filter by declaration origin
  - framework (as: `F`, `f`, `fw`, or `framework`), or
  - application (as: `A`, `a`, `app`, or `application`)
- *requested* - show parameters that have been requested by a loaded task
- *status* - filter for parameter setting status
  - not set (as `N`, `n`, or `notset`)
  - set from command line option (as: `O`, `o`, `option`)
  - set from environment (as `E`, `e`, `env`, `environment`)
  - set from configuration file (as `F`, `f`, `file`)
  - set from default value (as `D`, `d`, `default`)

```

filters
-d | --default           only parameters with a defined default value
-I | --install           only parameters required only by install tasks
-o | --origin ORIGIN    only parameters from origin: f(w), a(pp)
-r | --requested        only requested dependencies
-s | --status STATUS    only parameter for status: o, f, e, d

```

This task can take the list of items from the framework or application cache. If `CACHE_DIR` is set to a directory that contains cached parameter information (map), this information is used.

### 2.1.7. list-scenarios

Lists scenarios with their name, short name, and a description. Table mode shows a few more aspects of scenarios:

- the origin of its declaration (framework or application),
- the application mode flavor, being either **S** for **standard** or **std** or **I** for **install**,
- if the scenario is declared for application *dev*,
- if the scenario is declared for application *build*,
- if the scenario is declared for application *use*, and
- a color coded load status.

Provided filters are:

- *install* - only scenarios that are defined for the **install** application mode flavor
- *loaded* - show only scenarios currently loaded
- *mode* - show only scenarios for a specific application mode
  - for mode *all* use **A, a, All, all**
  - for mode *dev* use **D, d, Dev, dev**
  - for mode *build* use **B, b, Build, build**
  - for mode *use* use **U, u, Use, use**
- filters to exclude scenarios by name:
  - *no-a* for all these *no* filters
  - *no-b* to exclude scenarios that start with **build-**
  - *no-d* to exclude scenarios that start with **describe-**
  - *no-dl* to exclude scenarios that start with **describe-** or **list-**
  - *no-l* to exclude scenarios that start with **list-**
  - *no-s* to exclude scenarios that start with **start-**
- *origin* - \* *origin* - filter by declaration origin
  - framework (as: **F, f, fw, or framework**), or



- application (as: **A**, **a**, **app**, or **application**)
- **odl** - filter for scenarios that start with **describe-** or **list-**
- **status** - filter by scenario status
  - success (as: **S**, **s**, or **success**),
  - warnings (as: **W**, **w**, or **warning**),
  - errors (as: **E**, **e**, or **error**), or
  - not attempted (as: **N**, **n**, or **not-attempted**)
- **unloaded** - filter for scenarios that have been unloaded

```

filters
-I | --install           only scenarios for application mode flavor 'install'
-l | --loaded           only loaded scenarios
-m | --mode MODE
  --no-a                activate all '--no-' filters
  --no-b                exclude scenarios starting with 'build-'
  --no-d                exclude scenarios starting with 'describe-'
  --no-dl               exclude scenarios starting with 'describe-' or 'list-'
  --no-l                exclude scenarios starting with 'list-'
  --no-s                exclude scenarios starting with 'start-'
-o | --origin ORIGIN   only scenarios from origin: f(w), a(pp)
  --odl                 show only scenarios starting with 'describe-' or 'list-'
-s | --status STATUS   only scenarios with status: (s)uccess, (w)arning, (e)rror,
(n)ot attempted
-u | --unloaded         only unloaded scenarios

```

This task can take the list of items from the framework or application cache. If `CACHE_DIR` is set to a directory that contains cached scenario information (map), this information is used.

### 2.1.8. list-tasks

Lists tasks with their name, short name, and a description. Table mode shows a few more aspects of tasks:

- the origin of its declaration (framework or application),
- the application mode flavor, being either **S** for **standard** or **std** or **I** for **install**,
- if the task is declared for application *dev*,
- if the task is declared for application *build*,
- if the task is declared for application *use*, and
- a color coded load status.

Provided filters are:

- **install** - only tasks that are defined for the **install** application mode flavor
- **loaded** - show only tasks currently loaded

- *mode* - show only tasks for a specific application mode
  - for mode *all* use **A, a, All, all**
  - for mode *dev* use **D, d, Dev, dev**
  - for mode *build* use **B, b, Build, build**
  - for mode *use* use **U, u, Use, use**
- filters to exclude tasks by name:
  - *no-a* for all these *no* filters
  - *no-b* to exclude tasks that start with **build-**
  - *no-d* to exclude tasks that start with **describe-**
  - *no-dl* to exclude tasks that start with **describe-** or **list-**
  - *no-l* to exclude tasks that start with **list-**
  - *no-s* to exclude tasks that start with **start-**
- *origin* - \* *origin* - filter by declaration origin
  - framework (as: **F, f, fw, or framework**), or
  - application (as: **A, a, app, or application**)
- *odl* - filter for tasks that start with **describe-** or **list-**
- *status* - filter by task status
  - success (as: **S, s, or success**),
  - warnings (as: **W, w, or warning**),
  - errors (as: **E, e, or error**), or
  - not attempted (as: **N, n, or not-attempted**)
- *unloaded* - filter for tasks that have been unloaded

```

filters
-I | --install           only tasks for application mode flavor 'install'
-l | --loaded           only loaded tasks
-m | --mode MODE        only tasks for application mode: dev, build, use
  --no-a                activate all '--no-' filters
  --no-b                exclude tasks starting with 'build-'
  --no-d                exclude tasks starting with 'describe-'
  --no-dl               exclude tasks starting with 'describe-' or 'list-'
  --no-l                exclude tasks starting with 'list-'
  --no-s                exclude tasks starting with 'start-'
-o | --origin ORIGIN    only tasks from origin: f(w), a(pp)
  --odl                 show only tasks starting with 'describe-' or 'list-'
-s | --status STATUS    only tasks with status: (s)uccess, (w)arning, (e)rror, (n)ot
attempted
-u | --unloaded         only unloaded tasks

```

This task can take the list of items from the framework or application cache. If *CACHE\_DIR* is set to

a directory that contains cached task information (map), this information is used.

## 2.2. describe-\*

There are nine tasks that start with *describe-*. Each of these tasks provides a description of one or more framework and application features, namely:

- *describe-application* - describes one or more application aspects (from the manual)
- *describe-command* - describes one or more shell commands
- *describe-dependency* - describes one or more dependencies
- *describe-element* - describes one or more element types of an application
- *describe-exitstatus* - describes one or more exit status (error codes)
- *describe-option* - describes one or more CLI options
- *describe-parameter* - describes one or more parameters
- *describe-scenario* - describes one or more scenarios
- *describe-task* - describes one or more tasks

All of these tasks have the same default behavior, provide the same main options, and can be used with (then task specific) filters. The default behavior, when executed without any arguments, is to provide a list of descriptions. For example, when *describe-task* is called it provides list of task descriptions, one per declared task. The description will be empty if no items were found, e.g. in *describe-scenario* when no scenarios have been loaded.

The standard options are shown below. *print-mode* can be set to any mode that is supported by the API, namely:

- *adoc* - print in AsciiDoc format
- *ansi* - print with ANSI formatting for colors and effects
- *text* - print as plain text, no special formatting will be used
- *text-anon* - print as text with some annotations, similar to *adoc*

```
-h | --help           print help screen and exit
-P | --print-mode MODE print mode: ansi, text, adoc
```

All describe tasks with filters provide a standard filter for describing all found items, shown below. Tasks that describe elements (e.g. task, parameter, dependency) also provide a filter to only describe a single element by name (identifier). All other filters are task specific. If filters are used, only items that satisfy the filter will be described. Any number of filters can be used in any combination.

```
filters
-A | --all           all settings, disables all other filters
```

### 2.2.1. describe-application

Describe application aspects from the manual. The provided filters focus on the different aspects. If filters are used, only aspects specified will be shown. If no filters are used, all aspects are shown.

```
filters
  --app           include application description
  --authors       include authors
  --bugs          include bugs
  --copying       include copying
  --resources     include resources
  --security      include security
```

The text is taken from the manual. This `MANUAL_SRC` must point to valid manual sources.

### 2.2.2. describe-command

Describes one or more commands. The `id` filter can be used to show only a specific command. For the `id`, the long or the short form of the command can be used.

```
filters
-i | --id ID      long command identifier
```

### 2.2.3. describe-dependency

Describes one or more dependencies. Provided filters are:

- `id` - identifies a specific dependency and only this one will be described
- `install` - only dependencies that are only required by tasks that run in `install` application mode flavor
- `origin` - filter by declaration origin
  - framework (as: `F`, `f`, `fw`, or `framework`), or
  - application (as: `A`, `a`, `app`, or `application`)
- `requested` - show dependencies that have been requested by a loaded task
- `status` - filter by dependency status
  - success (as: `S`, `s`, or `success`),
  - warnings (as: `W`, `w`, or `warning`),
  - errors (as: `E`, `e`, or `error`), or
  - not attempted (as: `N`, `n`, or `not-attempted`)
- `tested` - filter dependencies that have been tested, ignoring untested dependencies

```

filters
-i | --id ID           dependency identifier
-I | --install         only dependencies required only by install tasks
-o | --origin ORIGIN  only dependencies from origin: f(w), a(pp)
-r | --requested       only requested dependencies
-s | --status STATUS  only dependencies with status: (s)uccess, (w)arning, (e)rror,
(n)ot attempted
-t | --tested         only tested dependencies

```

## 2.2.4. describe-element

Describes framework and application element types, e.g. task, parameter, and description. The text is the same as used in the manual as introduction to the elements. The provided filters focus on the different element types. If filters are used, only types specified will be shown. If no filters are used, all types are shown.

```

filters
--cmd           include commands
--dep           include dependencies
--es           include exit status
--opt          include options
--param        include parameters
--scn          include scenarios
--task         include tasks

```

The text is taken from the manual. This *MANUAL\_SRC* must point to valid manual sources.

## 2.2.5. describe-exitstatus

Describes one or more exit status codes. The *id* filter can be used to show only a specific status code.

```

filters
-i | --id ID           exit-status identifier

```

## 2.2.6. describe-option

Describes one or more command line options. The *id* filter can be used to show only a specific option. For the *id*, the long or the short form of the option can be used. The other filters focus on the option type, being either *exit* options or *runtime* options.

```

-e | --exit           only exit options
-i | --id ID         long option identifier
-r | --run           only runtime options

```

## 2.2.7. describe-parameter

Describes one or more parameters. The following filters are provided:

- *default* - describe parameters with a set default value
- *id* - only describe a specific parameter, the identifier can be given in lower-case or upper-case or mixed spelling
- *install* - only parameters that are only required by tasks that run in **install** application mode flavor
- *origin* - filter by declaration origin
  - framework (as: **F**, **f**, **fw**, or **framework**), or
  - application (as: **A**, **a**, **app**, or **application**)
- *requested* - show parameters that have been requested by a loaded task
- *status* - filter for parameter setting status
  - not set (as **N**, **n**, or **notset**)
  - set from command line option (as: **O**, **o**, **option**)
  - set from environment (as **E**, **e**, **env**, **environment**)
  - set from configuration file (as **F**, **f**, **file**)
  - set from default value (as **D**, **d**, **default**)

```
-d | --default      only parameters with a defined default value
-i | --id ID       parameter identifier
-I | --install     only parameters required only by install tasks
-o | --origin ORIGIN only parameters from origin: f(w), a(pp)
-r | --requested   only requested dependencies
-s | --status STATUS only parameter for status: o, f, e, d
```

This task can also be used to show debug information, rather than descriptions. Using the `debug` option will print all information about one or more parameters (depending on the used filters).

```
-D | --debug      print debug information instead of description
```

## 2.2.8. describe-scenario

Describes one or more scenarios. Provided filters are:

- *id* - only describe a specific scenario, the identifier can be the long name or the short name of the scenario
- *install* - only scenarios that are defined for the **install** application mode flavor
- *loaded* - show only scenarios currently loaded
- *mode* - show only scenarios for a specific application mode

- for mode *all* use **A, a, All, all**
- for mode *dev* use **D, d, Dev, dev**
- for mode *build* use **B, b, Build, build**
- for mode *use* use **U, u, Use, use**
- *origin* - \* *origin* - filter by declaration origin
  - framework (as: **F, f, fw, or framework**), or
  - application (as: **A, a, app, or application**)
- *status* - filter by scenario status
  - success (as: **S, s, or success**),
  - warnings (as: **W, w, or warning**),
  - errors (as: **E, e, or error**), or
  - not attempted (as: **N, n, or not-attempted**)
- *unloaded* - filter for scenarios that have been unloaded

```

-i | --id ID           scenario identifier
-I | --install         only scenarios declared for application mode flavor 'install'
-l | --loaded         only loaded scenarios
-m | --mode MODE      only scenarios for application mode: dev, build, use
-o | --origin ORIGIN  only scenarios from origin: f(w), a(pp)
-s | --status STATUS  only scenarios for status: (s)uccess, (w)arning, (e)rror,
(n)ot attempted
-u | --unloaded       only unloaded scenarios

```

This task can also be used to show debug information, rather than descriptions. Using the `debug` option will print all information about one or more dependencies (depending on the used filters).

```

-D | --debug          print debug information instead of description

```

### 2.2.9. describe-task

Describes one or more tasks. Provided filters are:

- *id* - only describe a specific task, the identifier can be the long name or the short name of the task
- *install* - include tasks that are defined for the `install` application mode flavor
- *loaded* - show only tasks currently loaded
- *mode* - show only tasks for a specific application mode
  - for mode *all* use **A, a, All, all**
  - for mode *dev* use **D, d, Dev, dev**
  - for mode *build* use **B, b, Build, build**

- for mode *use* use **U**, **u**, **Use**, **use**
- *origin* - \* *origin* - filter by declaration origin
  - framework (as: **F**, **f**, **fw**, or **framework**), or
  - application (as: **A**, **a**, **app**, or **application**)
- *status* - filter by task status
  - success (as: **S**, **s**, or **success**),
  - warnings (as: **W**, **w**, or **warning**),
  - errors (as: **E**, **e**, or **error**), or
  - not attempted (as: **N**, **n**, or **not-attempted**)
- *unloaded* - filter for tasks that have been unloaded

```
-i | --id ID           task identifier
-I | --install        only tasks declared for application mode flavor 'install'
-l | --loaded         only loaded tasks
-m | --mode MODE      only tasks for application mode: dev, build, use
-o | --origin ORIGIN  only tasks from origin: f(w), a(pp)
-s | --status STATUS  only tasks for status: (s)uccess, (w)arning, (e)rror, (n)ot
attempted
-u | --unloaded       only unloaded tasks
```

This task can also be used to show debug information, rather than descriptions. Using the `debug` option will print all information about one or more tasks (depending on the used filters).

```
-D | --debug          print debug information instead of description
```

## 2.3. cloc-installation

This task counts the lines of code of an installation. It is available in all application modes.

The task forces files that `cloc` identifies as `sh` files to be treated as `bash` files, using the `cloc` option `--force-lang="Bourne Again Shell",sh`. This ensures that the installation's include and application files (without the `.sh` extension) are counted as `bash` files.

More details on `cloc` can be found at the Github source repository on [Github](#).

### 2.3.1. Options

The task does not have any special options.

```
-h | --help          print help screen and exit
```



## 2.3.2. Examples

Simply running the task will count the lines of code of an installation. The following is the output of running this task on the SKB-Framework in an earlier version.

```
545 text files.
539 unique files.
222 files ignored.

github.com/AlDanial/cloc v 1.80 T=1.00 s (376.0 files/s, 23126.0 lines/s)
-----
Language                files      blank      comment      code
-----
Bourne Again Shell      226        2618        5211        12077
HTML                    1           0           1           1635
AsciiDoc                147        308         0           1202
XML                     1           7           21           28
Ant                     1           4           1           13
-----
SUM:                    376        2937        5234        14955
-----
```

The count shows the two ANT files (build file and macro file) as ANT and XML language files. All ADOC files are shown as AsciiDoc language files The HTML file is the framework's HTML manual. The top line shows the lines of code for all *bash* scripts in the installation, including all tasks and *.id* files.

## 2.3.3. Requirements

The task requires the tool `cloc` being installed. The dependency is called by the same name: `cloc`. If `cloc` is not installed, the task will print an error.

## 2.4. execute-program

This task executes an external program. The program can be started in different ways:

- Without any argument, the program is executed inside the task. This means that the task hands over execution to the program. Only when the program is finished will the task continue.
- Using *background* will start the program in the background, adding an ampersand `&` to the program. This means that the task will continue immediately after the program is started. Since the task then terminates, there is no further job control on the program. Furthermore, all program output will appear along with the task and then the shell.
- Using *xterm* will start the a new XTerm and execute the program there as a command. This means that the task will continue immediately after the program is started. Since the task then terminates, there is no further job control on the XTerm. All output will of course happen in the new XTerm.

```
-b | --background  run program in background
-h | --help        print help screen and exit
-t | --title TITLE title for the XTerm, default: program name
-x | --xterm       start a new XTerm and execute program
```

The actual program and its arguments should be provided after `--` in the command line. All characters here will be simply used to execute the program.

### 2.4.1. Examples

The following example executes `ls` in the current directory (where the application was started from).

```
execute-program — ls
```

The following example executes `ls` in the current directory (where the application was started from). It runs `ls` in the background, which causes an error since `ls` takes the added ampersand `&` as an argument.

```
execute-program --background — ls
```

The following example starts the editor `vi` with the file `build.sh` in a new XTerm.

```
execute-program --xterm — vi build.sh
```

### 2.4.2. Requirements

To start a program in a new XTerm, the task `start-xterm` must be available. This task has its own configuration settings, e.g. to set the XTerm executable.

## 2.5. manual

This task will show the application manual. If no filters are used, i.e. no argument is provided, the text manual for the current print mode will be shown.

```

-h | --help      print help screen and exit

filters
-A | --all       all manual versions
  --adoc        ADOC manual
  --ansi        text manual with ansi colors and effects
  --html        HTML manual
  --manp        manual page skb-framework(1)
  --pdf         PDF manual
  --text        plain text manual
  --text-anon   annotated text manual

```

The filters shown above can be used to show different versions of the manual. The filter *all* will show all manual versions (it overwrites all other filters). Otherwise, any number of filters can be used in any order. The task will always show the manual in the following order:

- *adoc* - the ADOC (AsciiDoc) manual, filter *adoc*
- *anon* - the annotated text version of the manual, filter *text-anon*
- *ansi* - the ANSI formatted manual, filter *ansi*
- *text* - the plain text manual, filter *text*
- *manp* - the manual page, filter *manp*
- *html* - the HTML version in a web browser, filter *html*
- *pdf* - the PDF version in a PDF viewer, filter *pdf*

The text versions (*adoc*, *anon*, *ansi*, and *text*) are shown using the command `less` with the options `-r -C -f -M -d`. `tput` is used to save (`smcup`) and restore (`rmcup`) the terminal context.

The manual page (*manp*) is shown using the command `man`.

### 2.5.1. Requirements

The task requires the manual being build in the application home directory. If a requested manual version is not found, an error will be thrown.

To show HTML and PDF versions, this task uses the tasks `start-browser` and `start-pdf-viewer`, respectively. Both tasks have their own configuration requirements, please see their documentation for details.

## 2.6. repeat-scenario

This task repeats a scenario. It is available in all application modes.

### 2.6.1. Options

The option *scenario* should be used to identify the scenario to repeat. The scenario must be loaded. For this option, the long or short name of the scenario can be used.

The other options determine how often the scenario should be repeated (*times*) and how long the task should wait between repetitions (*wait*). Only positive integers are allowed for both options. The default value for both options is 1. So when only provided with a scenario, the task will run it once.

```
-h | --help          print help screen and exit
-s | --scenario SCENARIO  the scenario to repeat
-t | --times INT       repeat INT times
-w | --wait SEC        wait SEC seconds between repeats
```

## 2.6.2. Examples

The example below will repeat the scenario *S1* four times, waiting for 2 seconds between repetitions.

```
repeat-scenario --scenario S1 --times 4 --wait 2
```

## 2.7. repeat-task

This task will repeat a task *n* times and wait for *s* seconds between repetitions. The values for *times* and *wait* must be positive integers. The default values are repeat once (*times* = 1) and wait for 1 second (*wait* = 1).

```
-h | --help          print help screen and exit
-t | --times INT     repeat INT times
-w | --wait SEC      wait SEC seconds between repeats
```

The task and task parameters should be provided after `--` in the command line. This task assumes that the first word after `--` is the task name (or identifier, either the long or the short form) and all other characters are arguments. The task name will be used to execute the task, the arguments then will not be processed but simply handed over to the executed task.

### 2.7.1. Examples

The example below will execute the task `list-tasks` with the arguments `-T --no-a` three times and wait for two seconds between repetitions.

```
repeat-task --times 3 --wait 2 --lt -T --no-a
```

The example below will do the exact same. It simply uses the long name of the task and the long form of its arguments

```
repeat-task --times 3 --wait 2 — list-tasks --table --no-a
```

## 2.8. setting

This task alters selected settings. Changeable settings are:

- Print mode (*--pm*)
  - *ansi* for ANSI formatted text
  - *text* for plain text
  - *text-anon* for annotated text
- Levels for the shell (*--shell-level*) or tasks (*--task-level*)
  - *all* - activates all levels (except *off*)
  - *fatal* - only fatal errors
  - *error* - fatal and standard errors
  - *warn-strict* - all errors and strict warnings
  - *warn* - all errors and warnings
  - *info* - errors, warnings, and information about the progress
  - *debug* - *info* plus detailed progress information
  - *trace* - *debug* plus further details
  - *off* - no level activated, messages and shell prompt will still be printed
- Shell prompt (*--snp*)
  - alters the printing of the shell prompt between *on* and *off*
- Quiet mode for shell (*--sq*) and tasks (*--tq*)
  - alter the quiet mode between *on* and *off*

```
options
-h | --help          print help screen and exit

options for changing settings
-p | --pm MODE      print mode to: ansi, text, text-anon
-S | --shell-level LEVEL change shell level to LEVEL
  --snp            toggle shell prompt mode
  --sq            toggle shell quiet mode
-s | --strict      toggle strict mode
-T | --task-level LEVEL change task level to LEVEL
  --tq            toggle task quiet mode
```

Changes made by the task take immediate effect.

## 2.9. start-browser

This task starts web browser with a URL to show. The URL can be set with *url*. The URL will not be processed by the task.

```
-h | --help      print help screen and exit
-u | --url URL   optional URL to load in browser
```

### 2.9.1. Requirements

The actual command for starting a browser must be provided in *BROWSER*. If this parameter is not set, the task will print an error and exit. Examples for setting *BROWSER* are:

- Firefox in a new tab: `firefox --new-tab %URL%`
- Firefox in a new window: `firefox --new-window %URL%`

More information on the parameter *BROWSER* can be found in the framework manual.

## 2.10. start-pdf-viewer

This task starts a PDF viewer with a file to show in it. The file can be set with *file*.

```
-h | --help      print help screen and exit
-f | --file FILE PDF file to open in reader
```

The task will test if the file is readable, and throw an error if not. It will also use a system specific path if required (for instance on Cygwin).

### 2.10.1. Requirements

The actual command for starting a PDF viewer must be provided in *PDF\_VIEWER*. If this parameter is not set, the task will print an error and exit. Examples for setting *PDF\_VIEWER* are:

- acrobat: `acrobat %FILE%`
- evince: `evince %FILE%`

More information on the parameter *PDF\_VIEWER* can be found in the framework manual.

## 2.11. start-xterm

This task starts an X terminal, or short XTerm (or xterm), with a set title and command to run in it. The title can be set with *title*. For blanks in the title use quotes, for instance `--title "My XTERM"`.

```
-h | --help          print help screen and exit
-t | --title TITLE   title for the XTerm, default: command name
```

The command to run in the started XTerm should be provided to the task after `--`. The task assumes that the first word there is the command name (which then is used as the default title). All text after `--` will be taken as command.

Note: this task does not take any effort to *hold* the XTerm, i.e. to keep it open after the command finished. Use the *hold* option of your preferred XTerm in *XTERM* to realize this feature.

### 2.11.1. Examples

Start an XTerm, use the default title, and run `ls | more` as command.

```
start-xterm — ls | more
```

### 2.11.2. Requirements

The actual command for starting an XTerm must be provided in *XTERM*. If this parameter is not set, the task will print an error and exit. Examples for setting *XTERM* are:

- standard executable: `xterm -T %TITLE% -e %COMMAND%`
- MinTTY on Cygwin: `mintty -t %TITLE% %COMMAND%`
- XFCE 4 Terminal: `xfce4-terminal --disable-server --title='%TITLE%' -x %COMMAND%`

More information on the parameter *XTERM* can be found in the framework manual.

## 2.12. statistics

This task prints statistics about the all elements. The default, without using filters, is to print a few overview tables. The print mode can be changed using *print-mode*.

```
options
-h | --help          print help screen and exit
-P | --print-mode MODE print mode: ansi, text, adoc
```

### 2.12.1. Filters

Using filters will provide more detailed information. Any number of filters can be used in any sequence, the tasks will print the output always in the same order:

- Overview table (default, or if filter is set)
- Commands (if filter is set)
- Dependencies (if filter is set)

- Exit status (if filter is set)
- Options (if filter is set)
- Parameters (if filter is set)
- Tasks (if filter is set)
- Scenarios (if filter is set)

```
filters
-A | --all      activate all filters
-c | --cmd     for commands
-d | --dep     for dependencies
-e | --es      for exit status
-o | --opt     for options
  | --ov      overview
-p | --param   for parameters
-s | --scn    for scenarios
-t | --task    for tasks
```

### 2.12.2. Examples

The following example requests statistics for the target overview *--ov* and tasks *--task*.

```
stats --ov --task
```

The actual output depends of course on the declared and processed tasks. An earlier version of the framework shows the following statistics.



## Statistics

---

Tasks declared:	35	Scenarios declared:	1
Tasks loaded:	30	Scenarios loaded:	0

---

Dependencies declared:	15	Parameters declared:	12
Dependencies requested:	4	Parameters requested:	8
Dependencies tested:	4	Parameters w/default val:	6

---

Configuration settings:	28	Exit Status:	37
Options:	23	Commands:	11

---

## Tasks

---

Declared:	35	Not loaded:	5
- origin: framework:	35	Unloaded:	0
- origin: app:	0	Loaded:	30
- mode: dev:	35	- errors:	0
- mode: build:	32	- warnings:	2
- mode: use:	30	- success:	28

---

## 2.13. validate-installation

This task validates an installation. For each target, it will test the following conditions:

- All required or configured directories are available and readable
- All required files are available and readable
- All declarations have documentation files
- For directories with known content, e.g. task binaries, no extra files to exist

The task will issue errors for serious problems. It will issue strict warnings for problems that should not have an impact on the application at runtime (for example extra files in a directory). It can also issue warnings for less significant problems.

When the option *strict* is used, all strict warnings become errors. The *strict* mode is useful to validate an installation before packaging an application.

```
-h | --help      print help screen and exit
-s | --strict    run in strict mode
```

### 2.13.1. Targets

By default, the task will validate all targets. If one or more targets are requested, only those targets will be validated.

```
targets
-A | --all      set all targets
  --msrc       target: manual source
  --cmd        target: commands
  --dep        target: dependencies
  --es         target: exit-status
  --opt        target: options
  --param      target: parameters
  --scn        target: scenarios
  --task       target: tasks
```

The following conditions are validated:

- *msrc* - Manual Source
  - Test directory `application/` with files for all application aspects for the manual in ADOC and text version
  - Test directory `elements/` with files for all element types for the manual in ADOC and text version
  - Test directory `tags/` exists with files `name.txt` and `authors.txt`
- *cmd* - Commands
  - Test that the command declaration directory exists
  - For each declared command, test the ADOC and text documentation files exist
  - Test that no extra files are in the command declaration directory
  - Note: commands are only declared in the framework, not in an application
- *dep* - Dependencies
  - Test that the dependency declaration directory exists
  - For each declared dependency, test the ADOC and text documentation files exist
  - Test that no extra files are in the dependency declaration directory
  - Note: dependency declaration directories are tested in the framework and an application
- *es* - Exit Status
  - Test that the exit-status declaration directory exists
  - For each declared exit-status, test the ADOC and text documentation files exist

- Test that no extra files are in the exit-status declaration directory
- Note: exit-status are only declared in the framework, not in an application
- *opt* - Options
  - Test that the option declaration directory exists
  - For each declared option, test the ADOC and text documentation files exist
  - Test that no extra files are in the option declaration directory
  - Note: options are only declared in the framework, not in an application
- *param* - Parameters
  - Test that the parameter declaration directory exists
  - For each declared parameter, test the ADOC and text documentation files exist
  - Test that no extra files are in the parameter declaration directory
  - Note: parameter declaration directories are tested in the framework and an application
- *scn* - Scenarios
  - Test that the scenario declaration directory exists
  - For each declared scenario, test the ADOC and text documentation files exist
  - For each declared scenario, test that the scenario script file exists
  - Test that no extra files are in the scenario declaration directory
  - Note: scenario declaration directories are tested in the framework, an application, and for each directory in *SCENARIO\_PATH*
- *task* - Tasks
  - Test that the task declaration directory exists
  - For each declared task, test the ADOC and text documentation files exist
  - Test that no extra files are in the task declaration directory
  - For each declared task, test that the task script file exists
  - Test that no extra files are in the task script directory
  - Note: task declaration directories are tested in the framework and an application

## 2.14. wait

This task waits for a given amount of seconds. It can be used to *slow down* task execution, for instance in a scenario. It is available in all application modes.

### 2.14.1. Options

The option *seconds* takes a positive integer as argument for the number of seconds to wait. The default, without this option, is 1.

The actual wait time depends on the underlying system. Since *wait* is a task, a new *bash* instance is created to execute it. This creation does take time, less on powerful UNIX hosts than for instance on

Cygwin or a Raspberry PI. On native UNIX systems the creation time should not be significant. The actual wait time will be printed when the task finishes.

Since *bash* does not support floating point or double integer values, only positive integers can be used.

```
-h | --help          print help screen and exit
-s | --seconds SEC  wait SEC seconds, default is 1
```

### 2.14.2. Examples

The example below will wait for 3 seconds, plus the time it takes to execute the *wait* task itself.

```
wait --seconds 3
```

## 3. Build Tasks

This category of tasks either *builds* artifacts or *compiles* source files to create artifacts. Those tasks should be available in the application mode *build*, but no *use*. They might be available in the application mode *dev* if required. The exception to this general rule is the task *build-manual*, since it can be used to build an application-mode-specific manual and might thus be required in all application modes.

By conventions, all *build* and *compile* tasks should provide an argument *-c* or *--clean*. This argument should clean (remove) all built or compiled artifacts (and directories if applicable).

### 3.1. build-manual

This task builds the application manual. It uses the metadata provided by the element definitions (e.g. for a task in the task's *.adoc* file) and the general text from the application manual folder in *etc/manual*. The actual manual text comes from the AsciiDoctor files with the extension *adoc*. Element lists (e.g. for tasks or parameters) also use the description from the elements definition (in the *id* file).

Building the manual involves different steps, depending on the selected target:

- **ADOC**: generate an aggregated ADOC file with all text and lists.
- **TEXT**: convert the ADOC text into well-formatted paragraphs of plain or ANSI text, then build aggregated documents for text or ANSI formatted text. The converted text is saved in the *txt* file along with the original *adoc* file, since it is also used for the online help.
- **MANP, PDF, HTML**: use the aggregated ADOC file and the AsciiDoctor tool chain to generate a manpage, a single HTML file, or a PDF file. These targets require AsciiDoctor (*manp*, *html*) and AsciiDoctor-PDF (*pdf*) installed.

For all targets, the task will validate the installation to ensure that all required manual source files

(*adoc* and *txt* files) are accessible. If the validation does not pass successfully, no manual artifact will be build.

The general structure is the same for all targets, i.e. the task will create the same manual just for different output formats. The structure is similar to other manual pages:

- *Name* and *Synopsis*: name, tag line, general description on how to start the application
- *Description*: a description of the application
- *Options*: a description of command line (CLI) options, including a list of options Options are further categorized as *exit options* (application will process option and exit) and *runtime options* (directing runtime behavior).
- *Parameters*: a description and list of parameters that can be used to configure the application and its tasks.
- *Tasks*: a description and list of tasks provided by the application
- *Dependencies*: a description and list of dependencies (that tasks might require)
- *Shell Commands*: a description and list of commands the interactive shell provides
- *Exit Status*: a description and list of the different exit status codes and messages.
- *Scenarios*: a description and list of scenarios provided by the application
- *Security Concerns*: remarks on potential or actual security concerns when running the application
- *Bugs*: notes on known bugs
- *Authors*: list of authors
- *Resources*: list of resources
- *Copying*: notes on copyright and other related aspects

The task then provides negative filters to exclude parts of this general structure (with the exception of *name* and *synopsis*). This allows to generate tailored manuals for any application need.

The generated targets can also be tested, i.e. shown using external applications. All text formats (*adoc*, plain text, ANSI formatted text, annotated text) are shown using `less`. The manpage is shown using the command `man`. HTML and PDF files are shown with a web browser and a PDF reader, respectively. Note: the parameters `BROWSER` and `PDF_READER` must be set to test these targets.

Depending on the selected targets, the task generates the following output files:

- `$APP_HOME/doc/manual/{app-name}.adoc` - aggregated ADOC file with all text for the manual
- `$APP_HOME/doc/manual/{app-name}.text` - plain text file manual
- `$APP_HOME/doc/manual/{app-name}.text-anon` - annotated plain text manual
- `$APP_HOME/doc/manual/{app-name}.ansi` - ANSI formatted text manual
- `$APP_HOME/doc/manual/{app-name}.html` - single file HTML manual
- `$APP_HOME/doc/manual/{app-name}.pdf` - single file PDF manual
- `$APP_HOME/man/man1/{app-name}.1` - the generated manpage

In *warning* and *info* level, the task does not output any information (except errors). In *debug* level, the task provides detailed information about the progress. Build all targets, including the ADOC to text transformation, can take a few minutes even on a powerful host.

### 3.1.1. Configuration

The task can be configured with two parameters:

- *SKB\_FW\_TOOL* - optional, to find the tool for ADOC to text conversion, if not set, the target *src* cannot be build. This also requires the dependency *jre* to execute the tool.
- *MANUAL\_SRC* - optional, to set the source directory for the application related parts of the manual. If used, it must be a directory, readable, and with the correct source files (to pass validation).

### 3.1.2. Manual Source

The general text for the manual is located in the directory pointed to by the parameter *MANUAL\_SRC*. This directory must have the following layout and contents:



① Directory with general application text, the *.adoc* source must be present, the *.txt* files can be generated using the target *src*. For multi-paragraph text, use an empty line to separate paragraphs. To add a list, add an empty line and then each list element in a single line starting with an asteriks *\**. Finish the list with an empty line. Nested lists are not supported.

② List of authors.

③ Statements on bugs, known problems, etc.

- ④ Statements on copyright, licenses, etc.
- ⑤ A description of the application.
- ⑥ Links to resources, for instance a source repository or issue management or a web site.
- ⑦ Statements on security concerns when using the application.
- ⑧ Directory with text for the framework elements, the `.adoc` source must be present, the `.txt` files can be generated using the target `src`. For the ADOC source, the same rules as for the *application* directory apply.
- ⑨ Introduction to shell commands.
- ⑩ Introduction and text for dependencies.
- ⑪ Text for *exit* command line options.
- ⑫ Text for exit status codes and error messages.
- ⑬ Introduction and text for command line options.
- ⑭ Introduction and text for parameters.
- ⑮ Text for *runtime* command line options.
- ⑯ Introduction and text for scenarios.
- ⑰ Introduction and text for tasks.
- ⑱ Directory with tags, these files are used as plain text files.
- ⑲ A list of authors, used in the ADOC file header.
- ⑳ A tag line for the application, used in the *name* and *synopsis* sections.

### 3.1.3. General Options

Following the SKB-Framework convention, the task has two main options: *clean* to remove built manual artifacts and *build* to build them. When *build* is used, other general options and filters can be used to direct the build:

- *all* - build everything (*src*, primary targets, secondary targets)
- *primary* - build all primary targets, i.e. *src* and *adoc*
- *secondary* - build all secondary targets, i.e. *text* (plain, ANSI, annotated), *manp*, *html*, and *pdf*

```
-A | --all      set all targets, overwrites other options
-b | --build   builds a manual (manpage), requires a target
-c | --clean   removes all target artifacts
-h | --help    print help screen and exit
-p | --primary set all primary targets
-s | --secondary set all secondary targets
-t | --test    test a manual (show results), requires a target
```

### 3.1.4. Target Options

Targets can also be selected individually. The target options can be used in any sequence in the command line, the task will automatically generate all manual artifacts in the correct order. For the secondary targets that require *adoc* to be build, the task will also automatically generate *adoc* if the file does not exist. Text sources (target *src*) are not created automatically, only on request.

```
--adoc      secondary target: text versions: ansi, text
--html      secondary target: HTML file
--manp      secondary target: man page file
--pdf       secondary target: PDF file)
--text      secondary target: text versions: ansi, text
--src       primary target: text source files from ADOC
```

### 3.1.5. Element List Filters

Some parts of the manual list application elements. For selected element types, the element list filters can be used to direct what these lists contain:

- *loaded* - applies to task lists and scenario lists. If not used, all tasks and all scenarios will be listed. If this option is used, only the loaded tasks and scenarios are listed. Loaded here means that the elements are defined for mode the application was started in *and* have been successfully loaded.
- *install* - do list tasks and scenarios that are defined for the application mode flavor *install* By default, tasks and scenarios defined for the application mode flavor *install* are not listed in the manual. If used, any task or scenario defined for the flavor *install* will be listed. This filter also extends to parameters and dependencies. If used, parameters and dependencies that are *only* required by one or more *install* tasks or scenarios will be listed.
- *requested* - applies to dependency lists and parameter lists. If not used, all dependencies and all parameters will be listed. If this option is used, only the requested dependencies and parameters are listed. Requested here means any dependency or parameter requested by a loaded task.

```
-I | --install    do list 'install' app mode flavor tasks and scenarios
-l | --loaded     list only loaded tasks and scenarios
-r | --requested  list only requested dependencies and parameters
```

### 3.1.6. Application Filters

These filters are negative filters to exclude general (application related) parts of the manual. The option name corresponds to the heading in the general manual structure described above. The default behavior is to include all general parts.



```
--no-authors      do not include authors
--no-bugs         do not include bugs
--no-copying     do not include copying
--no-resources   do not include resources
--no-security    do not include security
```

### 3.1.7. Element Filters

These filters are negative filters to exclude element description or element lists. By default, all element descriptions and all element lists are included in the manual. To not show a list, use the *no-\*list* options. To not show any description for an element type, use the *no-\** options (without *-list*).

To give an example: to show all information about tasks do not use any of these filters. To show the general text, but no task list, use *--no-task-list*. To not show any information about tasks, use *--no-tasks*.

```
--no-commands      do not include commands
--no-command-list  include command text, but no list
--no-deps          include dependency text, but no list
--no-dep-list      do not include dependencies
--no-exitstatus    do not include exit status
--no-exitstatus-list include exit status text, but no list
--no-options       do not include options
--no-option-list   include option text, but no list
--no-params        do not include parameters
--no-param-list    include parameter text, but no list
--no-scenarios     do not include scenarios
--no-scenario-list include scenario text, but no list
--no-tasks         do not include tasks
--no-task-list     include task text, but no list
```

### 3.1.8. Examples

The following example will use the framework tool to convert *adoc* sources into well-formatted plain text.

```
build-manual --build --src
```

The following examples build the targets *adoc*, *text*, *manp*, *html*, and *pdf*. All tasks and scenarios will be listed. Only requested dependencies and parameters will be listed.

```
build-manual --build --requested --adoc --text --manp --html --pdf
```

### 3.1.9. Task Requirements

The task has the following requirements:

- *SKB\_FW\_TOOL* - optional, to find the tool for ADOC to text conversion, if not set, the target *src* cannot be build. This also requires the dependency *jre* to execute the tool.
- *MANUAL\_SRC* - optional, to set the source directory for the application related parts of the manual. If used, it must be a directory, readable, and with the correct source files (to pass validation).
- *asciidocctor* - optional dependency required to generate *manp* and *html* targets. If it does not exist, these targets cannot be generated.
- *asciidocctor-pdf* - optional dependency required to generate the *pdf* target. If it does not exist, this target cannot be generated.
- *start-browser* - optional task to start a web browser testing the generated *html* target. If not present, not successfully loaded, or has missing parameters, the target *html* cannot be tested.
- *start-pdf-viewer* - optional task to start a PDF reader testing the generated *pdf* target. If not present, not successfully loaded, or has missing parameters, the target *pdf* cannot be tested.

The task will automatically test if the required directories exist. If not, they need to be created manually, since the task does not create any directories:

- `$APP_HOME/man/man1` - for the *manp* target
- `$APP_HOME/doc/manual` - for all other targets

A few standard framework tasks are also required (all of them are mandatory and included in a standard framework installation):

- *describe-option* - this task is used to generate option lists.
- *describe-parameter* - this task is used to generate the parameter list.
- *describe-task* - this task is used to generate the task list it is a mandatory.
- *describe-dependency* - this task is used to generate the dependency list.
- *describe-existatus* - this task is used to generate the exit status list.
- *describe-command* - this task is used to generate the command list.
- *describe-scenario* - this task is used to generate the scenario list.
- *validate-installation* - this task is used to validate all input required for the manual, i.e. *adoc* and *txt* files

### 3.1.10. Notes

This task will change directories and files in the application (or framework) installation. Sufficient permissions must exist to run this task successfully.

## 3.2. build-mvn-site

This task can be used to build Maven sites. Beside calling Apache Maven, it also provides functionality to run scripts before building and before staging a site. Those scripts can take full advantage of the framework and its API. This allows to build even very complex sites in an automated way, with features realized outside the Maven build process but fully integrated into the overall site building.

The task is not specific to one site, but can build one or more sites, even within one execution. The parameter `MVN_SITES` is used to identify the sites that can be build. It should contain a set of paths to the site directories. Each of these directories should contain a *pom* file (`pom.xml`) with the Maven specifications for the site. It also should contain a framework-specific metadata file (`skb-site.id`). It might contain a file `skb-site-scripts.skb` with scripts to be run before the site build and before the site staging.

The general options, shown below, allow to direct the behavior of the task.

- *build* - request one or more sites to be built
- *clean* - requests to clean one or more sites. Clean will call `mvn clean` to do the job
- *list* - will list all defined and loaded sites, which then can be build or cleaned.
- *test* - once a site is successfully build, it can be tested. Testing here means to start a web browser with the site's `index.html` file.

```
-b | --build      builds site(s), requires a target and site ID or all
-c | --clean     cleans all site(s)
-h | --help      print help screen and exit
-l | --list      list sites
-T | --test      test sites, open in browser
```

### 3.2.1. Target Options

The task can build a Maven site for several targets:

- *targets* - builds all targets
- *ad* - calls Maven with the argument `site:attach-descriptor`. This argument is important for multi-module sites with inherited descriptors
- *site* - calls Maven to build the site, but not stage it
- *stage* - calls Maven to stage a site that has been prior built successfully

```
-t | --targets    mvn: all targets
  --ad           mvn: site:attach-descriptor
  --site         mvn: site
  --stage        mvn: site:stage
```

There is no default target.

### 3.2.2. Filter Options

Filters allow to specify which of the loaded (and available) sites should be built:

- *all* - will build all sites, in the order they are listed (alphabetically)
- *id* - can be used to build a specific site

```
-A | --all          all sites
-i | --id ID       site identifier for building
```

There is no default filter.

### 3.2.3. Maven Options

Further options exist to configure the runtime behavior of Maven:

- *profile* - call Maven with a specific profile setting, e.g. when the site definitions in the POM file are within a profile

```
--profile PROFILE mvn: use profile PROFILE
```

### 3.2.4. Examples

The framework's own site can be built using this task. When correctly configured, the site can be listed

```
build-mvn-site --list
```

This will show (with a real path instead of *SOME\_PATH*)

```
Sites
fw - SKB Framework site - SOME_PATH/dev/skb/bash/framework
```

The site can then be built using

```
build-mvn-site --build --id fw
```

The task and Maven output will be similar to:

```

[INFO] Scanning for projects...
[WARNING] The project de.vandermeer:skb-framework:pom:0.0.1 uses prerequisites which
is only intended for maven-plugin projects but not for non maven-plugin projects. For
such purposes you should use the maven-enforcer-plugin. See
https://maven.apache.org/enforcer/enforcer-rules/requireMavenVersion.html
[INFO]
[INFO] -----< de.vandermeer:skb-framework >-----
[INFO] Building skb-framework 0.0.1
[INFO] -----[ pom ]-----
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.079 s
[INFO] Finished at: 2018-11-15T21:08:19Z
[INFO] -----
-> building user guide (html, pdf)
-> building task guide (html, pdf)
-> building developer guide (html, pdf)
-> building implementation doc (html, pdf)

```

Setting the task level to *info* will provide more information on the task's progress.

### 3.2.5. Site Files

As mentioned above, a site suitable for this task should provide a file with its metadata and might provide a file with scripts to be used in the build. Both files must be located in the same directory as the POM file.

The metadata file must be named `skb-site.id`. It must contain the following information:

- An identifier. This identifier can be any string that does not contain whitespaces or *bash* special characters (such as `&` or `*`). The identifier must be unique for all site in *MVN\_SITES*.
- A description. The description should be short, like a tag line, to fit into the listing of sites.

The source block below shows the file for the frameworks' site.

```

ID=fw
DESCRIPTION="the SKB Framework site"

```

The file `skb-site-scripts.skb` could be provided. If it does not exist, the task will build the site purely using Maven. If it does exist, the task will source the file and call two functions the file must provide. This means that the file can

- Use anything a normal *bash* script can do at any place in the file
- Use anything a normal *bash* script can do inside the two required functions
- Use the full framework settings and API at any place in the file

The following code block shows the standard contents of the script file with the two required functions:

```
MvnSitePreScript() {
    # any code run before Maven is called to build the site
}

MvnSitePostScript() {
    # any code run `mvn site` but before `mvn site:stage`
}
```

To see an example have a look at the framework's script file. The current file can be found in the Github repository: [skb-site-scripts.skb](#). Here, the function `MvnSitePreScript` executes framework tasks to create files for the site. The function `MvnSitePostScript` calls tools from the AsciiDoctor tool chain to create documents. It also uses the framework's API function `MvnSiteFixAdoc`, which helps to fix some AsciiDoctor problem in generated HTML files in the site.

### 3.2.6. Requirements

The task takes the sites it should load and build from the parameter `MVN_SITES`. If no sites are provided, or no sites could be loaded (due to a missing side identifier file), the task will throw an error.

Building a Maven site of course requires Maven (dependency *maven*), which in turn requires a JDK to be installed on the host as well.

Testing a site is done by starting a web browser with the site's `index.html` file. This requires the task `start-browser` to be loaded and well-configured. Without this task be available, sites cannot be tested.

## 3.3. clean

This tasks cleans, i.e. removes, all artifacts that have been build by other tasks. In addition to that, the directory set by the parameter `TARGET` will be removed.

The task does not actually remove any built artifacts itself. It uses the `-c` (or `--clean`) option of all tasks that do build artifacts. By convention, these are tasks whose name starts with *build-* or *compile-*.

So *clean* will lookup the loaded task map, find all tasks that start with *build-* or *compile-* and execute them with the `--clean` option. If any of these tasks fails (for instance due to missing parameters), *clean* will also fail.

### 3.3.1. Options

The option *simulate* can be used to simulate a clean. Here, not task will be executed and the directory `TARGET` will not be removed either. Instead, the task will simply print what commands it would run.

The default command to remove the *TARGET* directory is `rm -frI`. This is an additional safety feature to prevent accidental removal of the directory. The option *force* can be used to remove the directory forced, i.e. using `rm -fr` instead. *simulate* will overwrite *force*.

```
-f | --force      force mode, not questions asked
-h | --help      print help screen and exit
-s | --simulate  print only, removes nothing, overwrites force
```

## 3.4. make-target-sets

This task can be used to build Maven sites. Beside calling Apache Maven, it also provides functionality to run scripts before building and before staging a site. Those scripts can take full advantage of the framework and its API. This allows to build even very complex sites in an automated way, with features realized outside the Maven build process but fully integrated into the overall site building.

The task is not specific to one site, but can build one or more sites, even within one execution. The parameter *MVN\_SITES* is used to identify the sites that can be build. It should contain a set of paths to the site directories. Each of these directories should contain a *pom* file (`pom.xml`) with the Maven specifications for the site. It also should contain a framework-specific metadata file (`skb-site.id`). It might contain a file `skb-site-scripts.skb` with scripts to be run before the site build and before the site staging.

The general options, shown below, allow to direct the behavior of the task.

- *build* - request one or more sites to be built
- *clean* - requests to clean one or more sites. Clean will call `mvn clean` to do the job
- *list* - will list all defined and loaded sites, which then can be build or cleaned.
- *test* - once a site is successfully build, it can be tested. Testing here means to start a web browser with the site's `index.html` file.

```
-b | --build      builds site(s), requires a target and site ID or all
-c | --clean     cleans all site(s)
-h | --help      print help screen and exit
-l | --list      list sites
-T | --test      test sites, open in browser
```

### 3.4.1. Target Options

The task can build a Maven site for several targets:

- *targets* - builds all targets
- *ad* - calls Maven with the argument `site:attach-descriptor`. This argument is important for multi-module sites with inherited descriptors
- *site* - calls Maven to build the site, but not stage it

- *stage* - calls Maven to stage a site that has been prior built successfully

```
-t | --targets      mvn: all targets
  --ad             mvn: site:attach-descriptor
  --site           mvn: site
  --stage          mvn: site:stage
```

There is no default target.

### 3.4.2. Filter Options

Filters allow to specify which of the loaded (and available) sites should be built:

- *all* - will build all sites, in the order they are listed (alphabetically)
- *id* - can be used to build a specific site

```
-A | --all          all sites
-i | --id ID       site identifier for building
```

There is no default filter.

### 3.4.3. Maven Options

Further options exist to configure the runtime behavior of Maven:

- *profile* - call Maven with a specific profile setting, e.g. when the site definitions in the POM file are within a profile

```
--profile PROFILE mvn: use profile PROFILE
```

### 3.4.4. Examples

The framework's own site can be build using this task. When correctly configured, the site can be listed

```
build-mvn-site --list
```

This will show (with a real path instead of *SOME\_PATH*)

```
Sites
fw - SKB Framework site - SOME_PATH/dev/skb/bash/framework
```

The site can then be build using



```
build-mvn-site --build --id fw
```

The task and Maven output will be similar to:

```
[INFO] Scanning for projects...
[WARNING] The project de.vandermeer:skb-framework:pom:0.0.1 uses prerequisites which
is only intended for maven-plugin projects but not for non maven-plugin projects. For
such purposes you should use the maven-enforcer-plugin. See
https://maven.apache.org/enforcer/enforcer-rules/requireMavenVersion.html
[INFO]
[INFO] -----< de.vandermeer:skb-framework >-----
[INFO] Building skb-framework 0.0.1
[INFO] -----[ pom ]-----
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.079 s
[INFO] Finished at: 2018-11-15T21:08:19Z
[INFO] -----
-> building user guide (html, pdf)
-> building task guide (html, pdf)
-> building developer guide (html, pdf)
-> building implementation doc (html, pdf)
```

Setting the task level to *info* will provide more information on the task's progress.

### 3.4.5. Site Files

As mentioned above, a site suitable for this task should provide a file with its metadata and might provide a file with scripts to be used in the build. Both files must be located in the same directory as the POM file.

The metadata file must be named `skb-site.id`. It must contain the following information:

- An identifier. This identifier can be any string that does not contain whitespaces or *bash* special characters (such as `&` or `*`). The identifier must be unique for all site in *MVN\_SITES*.
- A description. The description should be short, like a tag line, to fit into the listing of sites.

The source block below shows the file for the frameworks' site.

```
ID=fw
DESCRIPTION="the SKB Framework site"
```

The file `skb-site-scripts.skb` could be provided. If it does not exist, the task will build the site purely using Maven. If it does exist, the task will source the file and call two functions the file must provide. This means that the file can

- Use anything a normal *bash* script can do at any place in the file
- Use anything a normal *bash* script can do inside the two required functions
- Use the full framework settings and API at any place in the file

The following code block shows the standard contents of the script file with the two required functions:

```
MvnSitePreScript() {  
    # any code run before Maven is called to build the site  
}  
  
MvnSitePostScript() {  
    # any code run `mvn site` but before `mvn site:stage`  
}
```

To see an example have a look at the framework's script file. The current file can be found in the Github repository: [skb-site-scripts.skb](#). Here, the function `MvnSitePreScript` executes framework tasks to create files for the site. The function `MvnSitePostScript` calls tools from the AsciiDoctor tool chain to create documents. It also uses the framework's API function `MvnSiteFixAdoc`, which helps to fix some AsciiDoctor problem in generated HTML files in the site.

### 3.4.6. Requirements

The task takes the sites it should load and build from the parameter `MVN_SITES`. If no sites are provided, or no sites could be loaded (due to a missing side identifier file), the task will throw an error.

Building a Maven site of course requires Maven (dependency *maven*), which in turn requires a JDK to be installed on the host as well.

Testing a site is done by starting a web browser with the site's `index.html` file. This requires the task `start-browser` to be loaded and well-configured. Without this task be available, sites cannot be tested.

## 4. Development Tasks

This category of tasks provides functionality required to develop an application or the framework. This can include the creation of required or optional runtime artifacts. Tasks that start with `build-` or `compile-` should provide an argument `-c` or `--clean` to clean (i.e. remove) artifacts.

Most *dev* tasks should only be available in the *dev* application mode.

### 4.1. build-cache

This task creates cached information about the application (or framework). This cache is not required to run an application. All functionality can be loaded at startup without a cache. However,

caches can speedup the application load as well as some runtime behavior namely the `list-*` tasks and the help on all tasks.

On powerful hosts the cache will not provide any advantage. On less powerful hosts (for instance a Raspberry PI) or on systems with slower output (e.g. using ANSI formatted text in Cygwin), a cache can significantly improve performance.

Without any arguments, the standard cache (for all declarations) is being build. `clean` will clean all cached information.

```
-b | --build      builds cache, requires a target
-c | --clean     removes all cached maps and screens
-h | --help     print help screen and exit
```

### 4.1.1. General Target Options

Target options can be used to set specific cache targets.

- `all` - for all targets, except individual tasks
- `decl` - to cache element declarations. Except for parameters, all other elements will be cached: options, commands, dependencies, tasks, and scenarios. For scenarios, only the standard directories of application and framework are cached, additional scenario directories (from `SCENARIPO_PATH`) will not be cached.
- `full` - cache everything, including individual tasks. This will cache declarations, tables, and the help screen of every loaded task.
- `tab` - cache table representations of all elements. This cache is used by the `list-*` tasks at runtime.

```
target options
-A | --all      set all targets, except tasks
-d | --decl    set all declaration targets
-f | --full    set all targets, including tasks
-t | --tab     set all table targets
```

### 4.1.2. Targets

Beside the general targets, the task can also be run with very specific targets, shown below. Here, declaration and table caches can be requested for each element type.

## targets

<code>--cmd-decl</code>	target: command declarations
<code>--cmd-tab</code>	target: command table
<code>--es-decl</code>	target: exit-status declarations
<code>--es-tab</code>	target: exit-status table
<code>--opt-decl</code>	target: option declarations
<code>--opt-tab</code>	target: option table
<code>--dep-decl</code>	target: dependency declarations
<code>--dep-tab</code>	target: dependency table
<code>--param-tab</code>	target: parameter table
<code>--task-decl</code>	target: task declarations
<code>--task-tab</code>	target: task table
<code>--tasks</code>	target: help screens for all(!) tasks

### 4.1.3. Requirements

This task requires the parameter `CACHE_DIR` to be set. Since this parameter has the default value of `/var/cache/$APP_NAME` it is always set.

### 4.1.4. Notes

This task will change directories and files in the cache directory. Sufficient permissions must exist to run this task successfully.

Care should be taken when using specific configurations for `CACHE_DIR`. Since there can be any number of SKB application installed on a single system, the cache directory should be different per application. Otherwise there can be unexpected behavior, especially for the declaration caches.

When declarations of cached elements (for instance a task) are changed, the cache is not automatically changed. This means, the changes will have no effect on the application load. This can lead to unexpected behavior. If caches are used, they should be cleared and rebuild whenever declarations change.

## 4.2. build-help

This task builds the help files for command line options and shell commands. These help files are used by the framework when

- An application is started with the help argument `-h` or `--help` and
- When help is requested in the interactive shell using the shell command `h`, `?`, or `help`.

In a standard installation, these help files should already exist. For example, the DEB and RPM distributions should create those files during the installation process (actually using this task).

The option `clean` is added by convention. It does not actually remove the help files, since this would break the installation of an application.

```
-c | --clean      added by convention, does nothing
-h | --help      print help screen and exit
```

### 4.2.1. Requirements

The task requires an installation of the framework. It also requires the tasks `list-options` and `list-commands` being loaded (this should be the default).

### 4.2.2. Notes

This task will change directories and files in the framework installation. Sufficient permissions must exist to run this task successfully.

## 4.3. download-fw-tool

This task uses CURL to download the SKB Framework Tool. The framework tool is an executable Java JAR file, including all dependencies. It is published on Bintray using the same version as the SKB Framework. The URL for download (without the filename) is: <https://dl.bintray.com/vdmeer/generic/>.

The framework tool provides mechanisms to convert ADOC text into formatted plain text. It is used for instance by the task `build-manual` to create formatted text versions of the manual.

The task uses the setting of the parameter `SKB_FRAMEWORK_TOOL` to determine which JAR file it should download. The output directory will be created if it does not exist. If the JAR file already exists, no download will be started (use the force option to force a download).

### 4.3.1. Options

The option `simulate` can be used to simulate all task actions. In this mode, the task will only print what it would do. The option `force` can be used to force a download even if the JAR file already exists. `simulate` will overwrite `force`.

```
-f | --force      force mode, not questions asked
-h | --help      print help screen and exit
-s | --simulate   print only, downloads nothing, overwrites force
```

## 4.4. set-file-versions

This task changes version information in source file headers. It runs Apache ANT using a simple build script, which in turn calls a macro that changes the version line. The default build and macro files change java files and all relevant framework files. These files can be used as template for writing other substitutions, if required.

```
-b | --build-file FILE    ANT build file
-d | --directory DIR     start directory with source files
-h | --help              print help screen and exit
-m | --macro-file FILE   ANT macro file
-v | --version VERSION   new version string
```

### 4.4.1. Examples

The following example will change the version in all files in the directory `src` to `0.1.0`. It will use the default build and macro file.

```
set-file-versions --version 0.1.0 --directory ./src
```

### 4.4.2. Requirements

The task requires Apache ANT (dependency `ant`). It also needs the parameters `VERSIONS_BUILD_FILE` and `VERSIONS_MACRO_FILE` to be set. Both parameters come with default values (the build and macro file provided by the framework). Settings these parameters will make the task using different build or macro files.

### 4.4.3. The Build File

The build file is a file called `build.xml` with information for ANT on what to build, and how. The default build file provided by the framework should be sufficient for all use cases. The XML below shows the default build file.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="pm" default="skb-set-versions">
  <!-- ant -f ant/build.xml -DmoduleVersion=VERSION_STRING -DmoduleDir=../
-DmacroFile=SOME_FILE -->
  <property name="module.version" value="${moduleVersion}"/>
  <condition property="macroFile" value="macro.xml">
    <not>
      <isset property="macroFile"/>
    </not>
  </condition>
  <import file="${macroFile}"/>
  <target name="skb-set-versions">
    <push-version module.dir="${moduleDir}" sourceEncoding="UTF-8" />
  </target>
</project>
```

The build file is kept very simple. The version string is provided by the setting `moduleVersion`, which is translated into the property `module.version`. The macro file is either provided by a setting `macroFile` or as the default value `macro.xml`. If the default value is used, the macro file must be in the same directory as the build file.

The build file then define its only target `skb-set-versions`. For the actual substitution of strings, the build file calls a macro `push-version`. The start directory is provided by the setting `moduleDir`, which is translated into a property `module.dir` for the macro.

Line 3 shows an example use of the build file from the command line.

#### 4.4.4. The Macro File

The macro file is called `macro.xml` (default) or defined in the build file (see above). The actual macro then must be called `push-version`. The following source block shows the default macro file provided by the framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="skb-set-versions">
  <macrodef name="push-version" description="Updates the internal version string of
all SKB-Framework source files.">
    <attribute name="module.dir"/>
    <sequential>
      <replaceregexp byline="false" encoding="UTF-8">
        <!-- sed "s/^ \* @version.*$/ \* @version    ${version}/" -->
        <regexp pattern=" \* @version(.*)" />
        <substitution expression=" \* \@version    ${module.version}" />
        <fileset dir="@{module.dir}" >
          <include name="src/**/*.java" />
        </fileset>
      </replaceregexp>

      <replaceregexp byline="false" encoding="UTF-8">
        <!-- sed "s/^## @version.*$/ \*## @version    ${version}/" -->
        <regexp pattern="## @version(.*)" />
        <substitution expression="## @version    ${module.version}" />
        <fileset dir="@{module.dir}" >
          <include name="src/main/bash/**/*.sh" />
          <include name="src/main/bash/**/*.id" />
          <include name="src/main/bash/scenarios/**/*.scn" />
          <include name="src/main/bash/bin/skb-framework" />
          <include name="src/main/bash/bin/**/_include" />
          <exclude name="**/set-file-versions.sh" />
        </fileset>
      </replaceregexp>
    </sequential>
  </macrodef>
</project>
```

The main functionality is a sequential execution of a regular expression replacement (`replaceregexp`), for all files satisfying the given filters. Replacements are done en block (i.e. not by line). The encoding is always `UTF-8`. For each replacement:

- `regexp` - defines the search pattern. This pattern is essentially a comment (in the respective

source file language), followed by the string `@version` and the rest of the actual line.

- **substitution** - defines the replacement string This string starts with the same comment, `@version` string, then followed by the new version `${module.version}`.
- **fileset** - defines which files should be processed. Files are taken from the start directory. **include** defines include patterns. **exclude** defines exclude patterns. Globbing is used to catch all files recursively. For instance `src/**/*.java` will process all files with the extension `.java` in the folder `@{module.dir}/src`.

The following examples show a number of standard patterns and substitution expressions.

#### *JDOC style comments*

```
sed "s/^ \* @version.*$/ \* @version    ${version}/"
regexp pattern=" \* @version(.*)"
substitution expression=" \* \@version    ${module.version}"
files: **/*.java
```

#### *C/C++/Java single line comment style*

```
sed "s/^ \\/\ @version.*$/ \* @version    ${version}/"
regexp pattern="// @version(.*)"
substitution expression="// \@version    ${module.version}"
files: **/*.java, files: **/*.cpp
```

#### *BASH single hash comment style*

```
sed "s/^# @version.*$/ \*# @version    ${version}/"
regexp pattern=" # @version(.*)"
substitution expression=" # \@version    ${module.version}"
files: **/*.sh
```

#### *BASH double hash comment style*

```
sed "s/^## @version.*$/ \*## @version    ${version}/"
regexp pattern="## @version(.*)"
substitution expression="## @version    ${module.version}"
files: **/*.sh
```