colcon Documentation

Dirk Thomas

Sep 15, 2018

User Documentation

colcon is a command line tool to improve the workflow of building, testing and using multiple software packages. It automates the process, handles the ordering and sets up the environment to use the packages.

The code is open source, and available on GitHub.

The documentation exists in two version:

- released: matching the latest released version of all packages
- latest: matching the latest state on the default branch of all packages

The documentation is organized into a few sections:

- *User Documentation*
- *Migrate from other build tools*

Information about development is also available:

- *Developer Documentation*

CHAPTER 1

Installation

The functionality of `colcon` is split over multiple Python packages. The package `colcon-core` provides the command line tool `colcon` itself as well as few fundamental extensions. Additional functionality is provided by separate packages, e.g. `colcon-cmake` adds support for packages which use CMake. The following instructions install a set of common `colcon` packages.

1.1 Using Debian packages

On platforms which support Debian packages using those is preferred since they will be updated using `apt` together with other system packages.

The Debian packages are currently hosted in apt repositories from the ROS project. You can choose either of the two following apt repositories.

- ROS 1 repository

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -cs` main
↪" > /etc/apt/sources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys␣
↪421C365BD9FF1F717815A3895523BAEEB01FA116
```

- ROS 2 repository

```
$ sudo sh -c 'echo "deb [arch=amd64,arm64] http://repo.ros2.org/ubuntu/main `lsb_
↪release -cs` main" > /etc/apt/sources.list.d/ros2-latest.list'
$ curl http://repo.ros2.org/repos.key | sudo apt-key add -
```

After that you can install the Debian package which depends on `colcon-core` as well as commonly used extension packages (see setup.cfg).

```
$ sudo apt update
$ sudo apt install python3-colcon-common-extensions
```

1.2 Using pip on any platform

On all non-Debian platforms the most common way of installation is the Python package manager `pip`. The following assumes that you are using a virtual environment with Python 3.5 or higher. If you want to install the packages globally it might be necessary to invoke `pip3` instead of `pip` and require `sudo`.

```
$ pip install -U colcon-common-extensions
```

Note: The package `colcon-common-extensions` doesn't contain any functionality itself but only depends on a set of other packages (see setup.cfg).

Note: You can find a list of released packages on PyPI using the keyword `colcon`.

1.3 Installing from source

Note: This approach is commonly only used by advanced users.

Commonly this is the case when you want to try or leverage new features or bug fixes which have been committed already but are not available in a released version yet. In order to use the latest state of any of the above packages you can invoke `pip` with a URL of the GitHub repository:

```
$ pip install -U git+https://github.com/colcon/colcon-common-extensions.git
```

1.4 Building from source

Since this is not a common use case for users you will find the documentation in the *developer section*.

1.5 Enable completion

1.5.1 Bash / zsh

On Linux / macOS the above instructions install the package `colcon-argcomplete` which offers command completion for bash and bash-like shells. To enable this feature you need to source the shell-specific script provided by that package. These scripts are named `colcon-argcomplete.bash` / `colcon-argcomplete.zsh`. For convenience you might want to source the one matching your shell in the user configuration, e.g. `~/.bashrc`:

Depending on which instructions you followed to install the packages the location will vary:

- Debian package: `/usr/share/colcon-argcomplete/hook`
- PIP - user specific: `$HOME/.local/share/colcon-argcomplete/hook`
- PIP - global: `/usr/local/share/colcon-argcomplete/hook`

CHAPTER 2

Quick start

This section gives a high-level overview of how to use the `colcon` command.

2.1 TL;DR

The following is an example workflow and sequence of commands using default settings:

```
$ mkdir -p /tmp/workspace/src      # Make a workspace directory with a src subdirectory
$ cd /tmp/workspace                # Change directory to the workspace root
$ <...>                            # Populate the `src` directory with packages
$ colcon list -g                   # List all packages in the workspace and their␣
↪dependencies
$ colcon build                     # Build all packages in the workspace
$ colcon test                      # Test all packages in the workspace
$ colcon test-result --all         # Enumerate all test results
$ . install/local_setup.bash       # Setup the environment to use the built packages
$ <...>                            # Use the built packages
```

The most commonly used arguments for the `build` and `test` verbs are to only process a specific package or a specific package including all the recursive dependencies it needs.

```
$ colcon build --packages-select <name-of-pkg>
$ colcon build --packages-up-to <name-of-pkg>
```

Note: The log files of the latest invocation can be found in the log directory which is by default in `~/.colcon/log/latest`.

Note: If you want to see the output of each package after it finished you can pass the argument `--event-handler console_cohesion+`.

2.2 Build ROS 2 packages

The process of building ROS 2 packages is described in the ROS 2 building from source instructions. Using `colcon` instead of the recommended tool `ament_tools` only changes a couple of the steps.

Instead of invoking `ament build` you can invoke `colcon`.

```
$ colcon build
```

In order to use the built packages you need to source the `install/local_setup.<ext>` script mentioned in the instructions.

For detailed information how command line arguments of `ament_tools` are mapped to `colcon` please see the *ament_tools migration guide*.

2.3 Build ROS 1 packages

The process of building ROS 1 packages is described in the distro specific building from source instructions. Using `colcon` instead of the recommended tool `catkin_make_isolated` only changes a couple of the steps.

Note: `colcon-ros` requires at least version 0.7.13 of `catkin` which provides a new CMake option the tool uses.

Instead of invoking `catkin_make_isolated --install` you can invoke `colcon`.

```
$ colcon build
```

Note: `colcon` does by design not support the concept of a "devel space" as it exists in ROS 1. Instead it requires each package to be installed so each package must declare an install step in order to work with `colcon`.

In order to use the built packages you need to source the `install/local_setup.<ext>` rather than the `setup.<ext>` script mentioned in the instructions.

For detailed information how command line arguments of `catkin_make_isolated` are mapped to `colcon` please see the *catkin_make_isolated migration guide*. For detailed information how command line arguments of `catkin_tools` are mapped to `colcon` please see the *catkin_tools migration guide*.

2.3.1 Test ROS 1 packages

Before you can run tests for ROS 1 packages, you must also build the custom `tests` target:

```
$ colcon build --cmake-target tests
```

2.4 Build Gazebo and the ignition packages

In more recent versions Gazebo has been refactored to split out a lot of the functionality into ignition libraries. While that makes the project more modular it also increases the effort necessary to build all these packages from source. `colcon` can make this process easy again.

In order to build a specific Gazebo version you need the right versions of the ignition libraries. At the time of writing Gazebo 9 is the latest release so we will use that for the purpose of this example. The following steps use a `.repos` to specify the various repositories with specific branches.

```
$ mkdir -p /tmp/gazebo/src && cd /tmp/gazebo
$ wget https://gist.githubusercontent.com/dirk-thomas/
↪6c1ca2a7f5f8c70ce7d3e1ef10a9f678/raw/490aaba72321284af956c9db12f9ef1550ef88cf/
↪Gazebo9.repos
$ vcs import src < Gazebo9.repos
```

Note: The Gist containing the repository list should be replaced with an "official" URL coming from the Gazebo project.

Before building the workspace with `colcon` the steps also fetch some additional metadata for Gazebo from a public repository.

```
$ colcon metadata add default https://raw.githubusercontent.com/colcon/colcon-
↪metadata-repository/master/index.yaml
$ colcon metadata update
$ colcon build
```

To run Gazebo which requires environment variables for e.g. the model paths the same commands as for other packages can be used. Using the additional metadata the source script will also automatically source the Gazebo specific file `share/gazebo/setup.sh` which defines these environment variables.

```
$ . install/local_setup.bash
$ gazebo
```

CHAPTER 3

Configuration

Configuration files can provide additional metadata for packages as well as define default command line arguments. All files described below are using the YAML format. Note that all strings are case sensitive.

3.1 colcon.pkg files

A `colcon.pkg` file must be placed in the root directory of a package.

The first level of the configuration file is a dictionary. The key can contain any of the following:

- `name` (string) to declare the package name
- `type` (string) to explicitly declare which colcon extension should process the package.
- `dependencies` (list of strings) to declare additional dependencies on other packages. For more fine grain control it is also possible to set `build-dependencies`, `run-dependencies`, and `test-dependencies`.
- `hooks` (list of relative paths within the install prefix) to declare additional scripts to be sourced.
- Any command line argument. The leading single or double dash must be omitted.

3.1.1 Values for command line arguments

The value type depends on the kind of command line argument:

- For flags which are not followed by a value the value can be either `true` or `false`.
- For options followed by a single decimal / float the value must be a decimal / float.
- For options followed by a single value the value must be a string.
- For options followed by one or more values the value must be a list where each item can be any of the mentioned types.

An example declaring an environment hook which should be sourced for a package:

```
{
    "hooks": ["share/pkgname/hook/something.sh"]
}
```

3.2 .meta files

The first level of the configuration file is a dictionary. The only two supported keys are: * `names` to provide settings based on the package name. * `paths` to provide settings based on the package path.

3.2.1 By package name

Note: Providing metadata based on the package name only works if the package can be identified and the name can be determined. Otherwise using a *colcon.pkg* file or the *By package path* configuration is necessary.

The value under the `names` key is again a dictionary.

The key is the name of the package. The value can contain the same package specific settings as described in the *colcon.pkg files* section above. The only except is that specifying a package name is not supported.

3.2.2 By package path

The value under the `paths` key is again a dictionary.

The key is the path of the package. It can be either absolute or relative to the `.meta` file. The value can contain the same package specific settings as described in the *colcon.pkg files* section above. This can be used if the package name can't be determined automatically and placing a `colcon.pkg` file into the package directory is undesired.

3.2.3 Package specific configuration

The package specific part under the package name or package path has the same content as the package specific configuration files described in the *colcon.pkg files* section above.

3.2.4 Using .meta files

Some configuration files are being picked up by default. The following are a few examples (see e.g. `colcon build --help`):

- When `--ignore-user-meta` is not passed any file ending with `.meta` in any recursive subdirectory of `$COLCON_HOME/metadata` is being used.
- When `--metas` is not passed and a file `./colcon.meta` exists it is being used.
- Any file passed with `--metas <path/to/file>` is being used.

Note: The default value for the environment variable `COLCON_HOME` is pointing to the directory `.colcon` within the users home directory.

3.3 defaults.yaml

If the configuration file `$COLCON_HOME/defaults.yaml` exists it is used to customize the default behavior of the CLI. The location can also be modified using the environment variable `COLCON_DEFAULTS_FILE` (see `colcon --help`).

The first level of the configuration file is a dictionary. The key is the `verb` name. The value is another dictionary containing the verb specific configuration.

3.3.1 Verb specific configuration

The key can contain any command line argument. The leading single or double dash must be omitted. The value type depends on the command line argument as mentioned in the *Values for command line arguments* section above.

An example to use the symlink install option by default:

```
{
    "build": {
        "symlink-install": true
    }
}
```

CHAPTER 4

How to

This section describe how to perform common tasks.

4.1 Show all output immediately on the console

```
$ colcon <verb> --event-handlers console_direct+
```

Note: If you use the parallel executor (which is the default when that extension is installed) the output of packages processed in parallel will be interleaved.

4.2 Show all output on the console after a package has finished

```
$ colcon <verb> --event-handlers console_cohesion+
```

Note: While this delays the output until a package has finished, it avoids interleaving the output when using the parallel executor.

4.3 Build only a single package (or selected packages)

```
$ colcon build --packages-select <name-of-pkg>
$ colcon build --packages-select <name-of-pkg> <name-of-another-pkg>
```

Note: This assumes that you have built dependencies of the selected packages within the workspace before.

4.4 Build selected packages including their dependencies

```
$ colcon build --packages-up-to <name-of-pkg>
```

4.5 Rebuild packages which depend on a specific package

Assuming you have built the whole workspace before and then made changes to one package. In order to rebuild this package as well as all packages which (recursively) depend on this package invoke:

```
$ colcon build --packages-above <name-of-pkg>
```

4.6 Test selected packages as well as their dependents

If you have built the relevant packages before you can run the tests the same way as described in the previous section:

```
$ colcon test --packages-above <name-of-pkg>
```

If you haven't built the relevant packages before you can do that by using one invocation to determine all dependents and a second invocation to invoke the actual build:

```
$ colcon list -n --packages-above <name-of-pkg>
$ colcon build --packages-up-to <copy-n-paste-output-previous-command>
```

4.7 Run specific tests

Depending on the type of the package a different tool is being used to run tests.

4.7.1 Python packages using pytest

```
$ colcon test --packages-select <name-of-pkg> --pytest-args ...
```

Pytest provides multiple ways to select individual tests:

- Tests can be identified by their name:

  ```
  $ ... --pytest-args -k name_of_the_test_function
  ```

- Tests can be identified using markers if the tests have been decorated with markers before:

  ```
  $ ... --pytest-args -m marker_name
  ```

Both approaches also support logical expressions like `or` and `not`. For more information see the pytest documentation.

4.7.2 CMake packages using CTest

```
$ colcon test --packages-select <name-of-pkg> --ctest-args ...
```

CTest provides multiple ways to select individual tests:

- Tests can be selected / excluded using a regular expression matching their name:

```
$ ... --ctest-args -R regex
$ ... --ctest-args -E regex
```

- Tests can be selected / excluded using a regular expression matching their label (which have to be assigned to each test when adding the test in the CMake code):

```
$ ... --ctest-args -L regex
$ ... --ctest-args -LE regex
```

For more information see the CTest documentation.

4.8 Build CMake packages without configuring tests

For CMake packages which use the CMake option `BUILD_TESTING` (which is the standard in the CTest module) you can skip configuring and building tests to improve the build time: .. code-block:: bash

    $ colcon build –cmake-args -DBUILD_TESTING=OFF

4.9 Enable additional output for debugging

Beside the output of the actually invoked commands to build or test packages the tool by default only outputs warning or error messages. For debugging purposes you can enable logging messages with other levels (e.g. `info`, `debug`).

```
$ colcon --log-level info <verb> ...
```

4.10 Log files of past invocations

By default the `log` directory is created as a sibling to the `src` directory. Some verbs (e.g. `build`, `test`, `test-result`) generate log files in a subdirectory which is named following the pattern `<verb>_<timestamp>`. For the latest invocation of a specific verb there is a symlink named `latest_<verb>` (on platforms which support symbolic links). For the latest invocation there is another symlink just named `latest` (on platforms which support symbolic links).

Each log directory contains a couple of files in the root:

- `events.log` contains all internal events dispatched. This file is mostly for debugging purposes.
- `logger_all.log` contains all logging messages even though the invocation didn't show them on the console. This is helpful to see log message with a different level after a command was run. The first line of this file contains the exact command line invocation including all the arguments passed.

For each package additional files are being created in a subdirectory named after the package:

- `command.log` contains the commands which have been invoked for the package, e.g. calls to `python setup.py`.

- `stdout.log` contains all the output the invoked commands printed to `stdout`.
- `stderr.log` contains all the output the invoked commands printed to `stderr`.
- `stdout_stderr.log` contains all the output the invoked commands printed to either of the two pipes in the order they appeared.
- `streams.log` combines the output of all the other log files in the order they appeared.

Note: While `colcon` is doing its best to read concurrently from the `stdout` and `stderr` pipes to preserve the order of output it can't guarantee the correctness of the order in all cases.

CHAPTER 5

Design

5.1 Goals for colcon

A few high level goals are used to guide the overall development.

- The tool should make building, testing and using multiple packages easy.
- It should be possible to add support for any kind of build system using extensions. `colcon-core` only bundles Python support in order to bootstrap itself.
- It should be possible to build any set of packages without requiring changes to their sources. If necessary missing information can be provided externally.
- After building packages they must be immediately usable which includes setting up necessary environment variables etc.

5.1.1 Explicitly out of scope

The tool does not aim to address any of the following tasks. Those should be left for other tools to take care of them.

- Fetch the source of the packages which should be processed by `colcon`.
- Install dependencies of the packages which should be processed by `colcon`.
- Perform packaging tasks like creating Debian packages.

Note: While these items are specifically not targeted by `colcon` it is still possible to implement support for any of these features (or helpful functionality to integration with existing tools) in an extension.

5.2 Software design

Additionally some software design goals are stated:

- All the functionality provided should be exposed in a way that it can be reused by other extensions.
- The separation into multiple Python packages is being used to encourage modularity and loose coupling (Law of Demeter). It is also used to demonstrate extensibility and show that certain features are not "special" but can be contributed externally.
- Each component should have responsibility over a single part of the software (Single responsibility principle).
- Each functionality added should follow the principle "you don't pay for what you don't use".

CHAPTER 6

Bootstrap from source

When developing `colcon` you want to have a local checkout of all involved packages.

Note: The following steps us the command line tool vcstool to fetch a set of repositories. You can e.g. install it using `pip install vcstool`.

Note: While the following instructions use a Linux shell the same can be done on other platforms like Windows with slightly adjusted commands.

6.1 Virtual environment

While not strictly necessary it is recommended to use a virtual environment for developing Python packages.

```
$ mkdir colcon-venv
$ python3 -m venv colcon-venv
$ . colcon-venv/bin/activate
```

Note: On Windows the Python 3 executable is likely named `python` and the activation script is invoked with `colcon-venv\Scripts\activate`

You might want to make sure that the venv is using up-to-date versions of the some foundational packages.

```
$ pip install -U pip setuptools
```

6.2 Fetch the sources

```
$ mkdir colcon-from-source && cd colcon-from-source
$ curl --output colcon.repos https://raw.githubusercontent.com/colcon/colcon.
↪readthedocs.org/master/colcon.repos
$ mkdir src
$ vcs import src < colcon.repos
```

Note: Depending on your platform you might not want to use all cloned packages. On Windows you must ignore or remove `colcon-argcomplete`, and may want to do the same for `colcon-bash`. If you don't use PowerShell you might want to ignore / remove the package `colcon-powershell`. To ignore a package add an empty file named COLCON_IGNORE to the folder.

Ignore `colcon-argcomplete` and `colcon-bash` on Windows.

```
> type nul > src\colcon-argcomplete\COLCON_IGNORE
> type nul > src\colcon-bash\COLCON_IGNORE
```

6.3 Dependencies

Make sure the dependencies are available:

```
$ curl --output requirements.txt https://raw.githubusercontent.com/colcon/colcon.
↪readthedocs.org/master/requirements.txt
$ pip install -r requirements.txt
```

6.4 Build the sources - first time

In the first build we will use the minimal features provided by `colcon-core` to build the set of cloned packages.

```
$ ./src/colcon-core/bin/colcon build --paths src/*
```

Note: On Windows the command needs to be prefixed with `python`.

The build of the packages will run sequentially and for each package the output will be printed directly to the console. The install directory will contain a `local_setup.sh` (or `.bat` on Windows).

In order to generate scripts for additional shells the set of packages have to be built a second time but this time using all extension provided by the various cloned packages.

6.5 Build the sources - second time

```
$ . install/local_setup.sh
$ colcon build
```

Note: On Windows the setup file ends with `.bat` and is just being called. Also the `colcon` executable can't be invoked directly here since while it is being used it can't be overwritten by the build. Instead invoke the following command: `python install\colcon-core\Scripts\colcon-script.py build`.

Note: The second build will process packages in parallel as long as their dependencies are finished. Also the output of all packages is not shown on the console (until there are errors) but is being redirected to log files. Depending on the platform you might also notice a status line during the build, a continuously updated title of the shell windows, and a desktop notification at the end of the build.

To use the full functionality you can source the generated script for your shell:

```
$ . install/local_setup.bash
```

Note: With bash you should now also have completion for all arguments if you have the Python package argcomplete installed. Try typing `colcon <tab>` to see the completion of global options and verbs.

CHAPTER 7

Contributions

There are already many great contribution guidelines available online. Therefore only a few important bullets are enumerated here. Please read for example the Open Source Guide [How to contribute](https://opensource.guide/how-to-contribute/) for more detailed information which was created and is curated by GitHub.

7.1 Bug reports

- Make sure you are using the latest version.
- Search the project's issue tracker and/or the internet for similar reports.
- Perform basic troubleshooting steps:
  - Try to reproduce the problem "from scratch".
  - If you are deviating from any instructions try to following the instructions and see if the problem persists.
  - If it seems to work for others ask yourself what is different in your case.
- Consider to provide a pull request if possible.
- And as a last step report a bug you can't solve yourself:
  - Describe the expected as well as the actual behavior.
  - Give enough context (e.g. platform, versions, environment).
  - Provide easily reproducible steps and/or a [SSCCE](http://sscce.org/).

7.2 Pull requests

- Keep each pull request focused on one aspect, create separate ones for separate aspects.
- For bug fixes make sure to reproduce the problem before and after applying the patch ensure that the problem has been addressed.

- For larger patches consider to create a feature request ticket to discuss the proposal ahead of time.
- Ensure that new code is covered by tests to prevent regressions in the future.
- Make sure that the code changes pass the existing tests including the linters.
- And as a last step create a pull request.

7.3 New packages / extensions

Using Python entry points it is easy to contribute extensions in separate packages. To ease discoverability and ensure long term maintenance if individual maintainers move on it is encouraged to host the code in a repository under the *colcon* organization unit on GitHub. Please open a ticket to either ask for the creation of a repository which you will have *admin* level access to or for moving an existing repository to this organization unit.

7.3.1 Use keyword in package metadata

When creating a package containing `colcon` extensions please consider declaring a keyword to help discovering extensions through e.g. PyPI. When using a `setup.cfg` file for the metadata of the package it is as simple as including these lines:

```
[metadata]
keywords = colcon
```

CHAPTER 8

ament_tools

The following describes the mapping of some `ament_tools` options and arguments to the `colcon` command line interface.

8.1 ament build | test

[BASEPATH] `--base-paths BASEPATH`

--build-space PATH `--build-base PATH`

--install-space PATH `--install-base PATH`

--build-tests CMake configures tests by default. To skip configuring tests use `--cmake-args -DBUILD_TESTING=OFF`.

-s, --symlink-install `--symlink-install`

--isolated The colcon option `--merge-install` has the inverse logic.

--start-with PKGNAME `--packages-start PKGNAME`

--end-with PKGNAME `--packages-end PKGNAME`

--only-packages PKGNAME1 ... PKGNAMEn `--packages-select PKGNAME1 ... PKGNAMEn`

--skip-packages PKGNAME1 ... PKGNAMEn `--packages-skip PKGNAME1 ... PKGNAMEn`

--parallel colcon uses the parallel execution by default. To build packages sequentially use `--executor sequential`.

8.2 ament build

`colcon build ...`

--cmake-args -D... -- `--cmake-args -D...` The closing double dash is not necessary anymore. Any CMake arguments which match colcon arguments need to be prefixed with a space. This can be done by quoting each argument with a leading space.

--force-cmake-configure `--cmake-force-configure`

--use-ninja `--cmake-args -G Ninja`

8.3 ament test

`colcon test ...`

--ctest-args ... -- `--ctest-args ...` Any CTest arguments which start with a dash need to be prefixed with a space (see `--cmake-args`).

--retest-until-fail N `--retest-until-fail N`

--retest-until-pass N `--retest-until-pass N`

--abort-on-test-error `--abort-on-error`

8.4 ament test_results

`colcon test-result ...`

[BASEPATH] `--build-base BASEPATH`

--verbose `--all`

8.5 Behavioral changes

The colcon `test` verb performs only the action of running tests. It does not build any packages.

`--retest-until-fail` with `colcon` uses pytest-repeat which runs individual tests of a package N+1 times each (the first test N+1 times, then the second test N+1 times, etc). With `ament_tools` the entire test suite of a package was run up to N+1 times. As a consequence `colcon` provides a more accurate result since each test that passed has actually run N times. Note that with `pytest-repeat`, `pytest` tests are repeated N times regardless of the result of the previous runs; if a test fails it will be repeated N times anyway. This is different from the behavior of a CTest test that will stop being repeated as soon as it fails once.

The location of JUnit test results file for `ament_python` packages tested with `colcon` is in `<pkg-build>/pytest.xml`, whereas with `ament_tools` it is in `<pkg-build>/test_results/<pkgname>/pytest.xunit.xml`.

CHAPTER 9

catkin_make_isolated

The following describes the mapping of some `catkin_make_isolated` options and arguments to the `colcon` command line interface.

`--source PATH` `--base-paths BASEPATH`

`--build PATH` `--build-base PATH`

`--devel PATH` colcon doesn't support the concept of a "devel" space. Instead you can choose the path of the devel space as the install base and perform an normal installation.

`--install-space PATH` `--install-base PATH`

`--merge` `--merge-install`

`--use-ninja` `--cmake-args -G Ninja`

`--use-nmake` `--cmake-args -G "NMake Makefiles"`

`--install` colcon always performs an installation. It doesn't support the concept of a "devel" space.

`--cmake-args ...` `--cmake-args ...` The closing double dash is not necessary anymore. Any CMake arguments which match colcon arguments need to be prefixed with a space. This can be done by quoting each argument with a leading space.

`--force-cmake` `--cmake-force-configure`

`--pkg PKGNAME1 ... PKGNAMEn` `--packages-select PKGNAME1 ... PKGNAMEn`

`--from-pkg PKGNAME` `--packages-start PKGNAME`

`--only-pkg-with-deps PKGNAME1 ... PKGNAMEn` `--packages-up-to PKGNAME1 ... PKGNAMEn`

CHAPTER 10

catkin_tools

The following describes the mapping of some `catkin_tools` options and arguments to the `colcon` command line interface.

10.1 catkin build

`[PKGNAME1 ... PKGNAMEn]` `--packages-up-to PKGNAME1 ... PKGNAMEn`

`--no-deps` `--packages-select PKGNAME1 ... PKGNAMEn`

`--start-with PKGNAME` `--packages-start PKGNAME`

`--force-cmake` `--cmake-force-configure`

`--cmake-args ... --` `--cmake-args ...` The closing double dash is not necessary anymore. Any CMake arguments which match colcon arguments need to be prefixed with a space. This can be done by quoting each argument with a leading space.

`-v, --verbose` `--event-handler console_cohesion+`

`-i, --interleave-output` `--event-handler console_direct+`

`--no-status` `--event-handler status-`

`--no-summarize, --no-summary` `--event-handler summary-`

`--no-notify` `--event-handler desktop_notification-`