# Workspace Mechanics

6.1 Workspace Configuration ........................................... 25
6.2 Workspace Anatomy ................................................. 27
6.3 Source Packages and Dependencies .............................. 30
6.4 Workspace Chaining / Extending ................................. 30

# Supported Build Types

7.1 Catkin ................................................................. 33
7.2 CMake ................................................................. 35

# Troubleshooting

8.1 Configuration Summary Warnings .............................. 37
8.2 Dependency Resolution ............................................ 37
8.3 Migration Problems ................................................ 38

# catkin build – Build Packages

9.1 Basic Usage ......................................................... 39
9.2 Building Subsets of Packages ................................. 41
9.3 Advanced Options ................................................. 43
9.4 Full Command-Line Interface ................................. 44

# catkin clean – Clean Build Products

10.1 Space Cleaning ................................................... 47
10.2 Partial Cleaning .................................................. 48
10.3 Cleaning Products from All Profiles ....................... 48
10.4 Cleaning Everything ............................................. 49
10.5 Full Command-Line Interface ............................... 49

# catkin config – Configure a Workspace

11.1 Viewing the Configuration Summary ....................... 51
11.2 Appending or Removing List-Type Arguments .......... 51
11.3 Installing Packages ............................................. 52
11.4 Explicitly Specifying Workspace Chaining ............. 52
11.5 Whitelisting and Blacklisting Packages .................. 53
11.6 Accelerated Building with Environment Caching .... 54
11.7 Full Command-Line Interface ............................. 54

# catkin create – Create Packages

12.1 Full Command-Line Interface ............................... 59

# catkin env – Environment Utility

13.1 Full Command-Line Interface ............................... 61

# catkin init – Initialize a Workspace

14.1 Full Command-Line Interface ............................... 63

# catkin list – List Package Info

15.1 Checking for Catkin Package Warnings .................. 65
15.2 Using Unformatted Output in Shell Scripts .......... 65
15.3 Full Command-Line Interface ............................. 65

# catkin locate – Locate Directories

16.1 Full Command-Line Interface ............................... 67

# catkin profile – Manage Profiles

17.1 Creating Profiles Automatically ......................... 69
Installing catkin_tools

You can install the catkin_tools package as a binary through a package manager like pip or apt-get, or from source.

Note: This project is still in beta and has not been released yet, please install from source. In particular, interface and behavior are still subject to incompatible changes. If you rely on a stable environment, please use catkin_make instead of this tool.

1.1 Installing on Ubuntu with apt-get

First you must have the ROS repositories which contain the .deb for catkin_tools:

```
$ sudo sh \
   -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -sc` main" \ 
     > /etc/apt/sources.list.d/ros-latest.list'
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

Once you have added that repository, run these commands to install catkin_tools:

```
$ sudo apt-get update
$ sudo apt-get install python3-catkin-tools
```

1.2 Installing on other platforms with pip

Simply install it with pip:

```
$ sudo pip3 install -U catkin_tools
```
1.3 Installing from source

First clone the source for catkin_tools:

```bash
$ git clone https://github.com/catkin/catkin_tools.git
$ cd catkin_tools
```

Then install the dependencies with pip:

```bash
$ pip3 install -r requirements.txt --upgrade
```

Then install with the setup.py file:

```bash
$ python3 setup.py install --record install_manifest.txt
```

**Note:** Depending on your environment/machine, you may need to use `sudo` with this command.

**Note:** If you want to perform a *local* install to your home directory, use the `install --user` option.

1.3.1 Developing

To setup catkin_tools for fast iteration during development, use the `develop` verb to `setup.py`:

```bash
$ python3 setup.py develop
```

Now the commands, like `catkin`, will be in the system path and the local source files located in the `catkin_tools` folder will be on the `PYTHONPATH`. When you are done with your development, undo this by running this command:

```bash
$ python3 setup.py develop -u
```

1.3.2 Uninstalling from Source

If you installed from source with the `--record` option, you can run the following to remove catkin_tools:

```bash
$ cat install_manifest.txt | xargs rm -rf
```
CHAPTER 2

A Brief History of Catkin

2.1 Legacy Catkin Workflow

The core Catkin meta-buildsystem was originally designed in order to efficiently build numerous inter-dependent, but separately developed, CMake projects. This was developed by the Robot Operating System (ROS) community, originally as a successor to the standard meta-buildtool rosbuild. The ROS community’s distributed development model with many modular projects and the need for building distributable binary packages motivated the design of a system which efficiently merged numerous disparate projects so that they utilize a single target dependency tree and build space.

To facilitate this “merged” build process, a workspace’s source space would contain boiler-plate “top-level” CMakeLists.txt which automatically added all of the Catkin CMake projects below it to the single large CMake project.

Then the user would build this collection of projects like a single unified CMake project with a workflow similar to the standard CMake out-of-source build workflow. They would all be configured with one invocation of cmake and subsequently targets would be built with one or more invocations of make:

```
$ mkdir build
$ cd build
$ cmake ../src
$ make
```

In order to help automate the merged build process, Catkin was distributed with a command-line tool called catkin_make. This command automated the above CMake work flow while setting some variables according to standard conventions. These defaults would result in the execution of the following commands:

```
$ mkdir build
$ cd build
$ cmake ../src -DCATKIN_DEVEL_SPACE=../devel -DCMAKE_INSTALL_PREFIX=../install
$ make -j<number of cores> -l<number of cores> [optional target, e.g. install]
```

An advantage of this approach is that the total configuration would be smaller than configuring each package individually and that the Make targets can be parallelized even among dependent packages.
In practice, however, it also means that in large workspaces, modification of the CMakeLists.txt of one package would necessitate the reconfiguration of all packages in the entire workspace.

A critical flaw of this approach, however, is that there is no fault isolation. An error in a leaf package (package with no dependencies) will prevent all packages from configuring. Packages might have colliding target names. The merged build process can even cause CMake errors to go undetected if one package defines variables needed by another one, and can depend on the order in which independent packages are built. Since packages are merged into a single CMake invocation, this approach also requires developers to specify explicit dependencies on some targets inside of their dependencies.

Another disadvantage of the merged build process is that it can only work on a homogeneous workspace consisting only of Catkin CMake packages. Other types of packages like plain CMake packages and autotools packages cannot be integrated into a single configuration and a single build step.

### 2.2 Isolated Catkin Workflow

The numerous drawbacks of the merged build process and the catkin_make tool motivated the development of the catkin_make_isolated tool. In contrast to catkin_make, the catkin_make_isolated command uses an isolated build process, wherein each package is independently configured, built, and loaded into the environment.

This way, each package is built in isolation and the next packages are built on the atomic result of the current one. This resolves the issues with target collisions, target dependency management, and other undesirable cross-talk between projects. This also allows for the homogeneous automation of other build tools like the plain CMake or autotools.

The isolated workflow also enabled the following features:

- Allowing building of part of a workspace
- Building Catkin and non-Catkin projects into a single **devel space**
- Building packages without re-configuring or re-building their dependencies
- Removing the requirement that all packages in the workspace are free of CMake errors before any packages can be built

There are, however, still some problems with catkin_make_isolated. First, it is dramatically slower than catkin_make since it cannot parallelize the building of targets or even packages which do not depend on each other. It also lacks robustness to changes in the list of packages in the workspace. Since it is a “released” tool, it also has strict API stability requirements.

### 2.3 Parallel Isolated Catkin Workflow and catkin build

The limitations of catkin_make_isolated and the need for additional high-level build tools lead to the development of a parallel version of catkin make isolated, or pcmi, as part of Project Tango. pcmi later became the build verb of the catkin command included in this project.

As such, the principle behavior of the **build verb** is to build each package in isolation and in topological order while parallelizing the building of packages which do not depend on each other.

Other functional improvements over catkin_make and catkin_make_isolated include the following:

- The use of sub-command “verbs” for better organization of build options and build-related functions
- Robustly adapting a build when packages are added to or removed from the **source space**
- Context-aware building of a given package based on the working directory
- The ability to completely clean a single package’s products from a workspace
• Utilization of persistent build metadata which catches common errors
• Support for different build “profiles” in a single workspace
• Explicit control of workspace chaining
• Additional error-checking for common environment configuration errors
• Numerous other command-line user-interface improvements
Chapter 2. A Brief History of Catkin
This chapter gives a high-level overview of how to use catkin_tools and the catkin command. This shows how to use the different command verbs to create and manipulate a workspace. For a more in-depth explanation of the mechanics of catkin workspaces, see Workspace Mechanics, and for thorough usage details see the individual verb documentation.

### 3.1 TL;DR

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

```bash
source /opt/ros/indigo/setup.bash  # Source ROS indigo to use Catkin
mkdir -p /tmp/quickstart_ws/src  # Make a new workspace and source space
cd /tmp/quickstart_ws
catkin init  # Initialize it with a hidden marker file

cd /tmp/quickstart_ws/src
catkin create pkg pkg_a  # Navigate to the source space
                        # Populate the source space with packages..

→

catkin create pkg pkg_b
catkin create pkg pkg_c --catkin-deps pkg_a
catkin create pkg pkg_d --catkin-deps pkg_a pkg_b

catkin list  # List the packages in the workspace

catkin build  # Build all packages in the workspace

source /tmp/quickstart_ws/devel/setup.bash  # Load the workspace's environment

catkin clean  # Clean all the build products
```

### 3.2 Initializing a New Workspace

While initialization of a workspace can be done automatically with catkin build, it’s good practice to initialize a catkin workspace explicitly. This is done by simply creating a new workspace with an empty source space (named src by default) and calling catkin init from the workspace root:
Now the directory `/tmp/quickstart-init` has been initialized and `catkin init` has printed the standard configuration summary to the console with the default values. This summary describes the layout of the workspace as well as other important settings which influence build and execution behavior.

Once a workspace has been initialized, the configuration summary can be displayed by calling `catkin config` without arguments from anywhere under the root of the workspace. Doing so will not modify your workspace. The `catkin` command is context-sensitive, so it will determine which workspace contains the current working directory.

An important property which deserves attention is the summary value labeled `Extending`. This describes other collections of libraries and packages which will be visible to your workspace. This is process called “workspace chaining.” The value can come from a few different sources, and can be classified in one of the three following ways:

- No chaining
- Implicit chaining via `CMAKE_PREFIX_PATH` environment or cache variable
- Explicit chaining via `catkin config --extend`

For more information on the configuration summary and workspace chaining, see `Workspace Configuration`. For information on manipulating these options, see the `config` verb.

**Note:** Calling `catkin init` “marks” a directory path by creating a hidden directory called `.catkin_tools`. This hidden directory is used to designate the parent as the root of a Catkin workspace as well as store persistent information about the workspace configuration.

### 3.3 Adding Packages to the Workspace

In order to build software with Catkin, it needs to be added to the workspace’s `source space`. You can either download some existing packages, or create one or more empty ones. As shown above, the default path for a Catkin `source space` is `.src` relative to the workspace root. A standard Catkin package is simply a directory with a `CMakeLists.txt` file and a `package.xml` file. For more information on Catkin packages see `workspace mechanics`. The shell interaction below shows the creation of four empty packages: `pkg_a`, `pkg_b`, `pkg_c`, and `pkg_d`:

```
source /opt/ros/indigo/setup.bash          # Source ROS indigo to use Catkin
mkdir -p /tmp/quickstart_ws/src           # Make a new workspace and source space
cd /tmp/quickstart_ws                    # Navigate to the workspace root
catkin init                              # Initialize it with a hidden marker file
```

```
Now the directory `/tmp/quickstart-init` has been initialized and `catkin init` has printed the standard configuration summary to the console with the default values. This summary describes the layout of the workspace as well as other important settings which influence build and execution behavior.

Once a workspace has been initialized, the configuration summary can be displayed by calling `catkin config` without arguments from anywhere under the root of the workspace. Doing so will not modify your workspace. The `catkin` command is context-sensitive, so it will determine which workspace contains the current working directory.

An important property which deserves attention is the summary value labeled `Extending`. This describes other collections of libraries and packages which will be visible to your workspace. This is process called “workspace chaining.” The value can come from a few different sources, and can be classified in one of the three following ways:

- No chaining
- Implicit chaining via `CMAKE_PREFIX_PATH` environment or cache variable
- Explicit chaining via `catkin config --extend`

For more information on the configuration summary and workspace chaining, see `Workspace Configuration`. For information on manipulating these options, see the `config` verb.

**Note:** Calling `catkin init` “marks” a directory path by creating a hidden directory called `.catkin_tools`. This hidden directory is used to designate the parent as the root of a Catkin workspace as well as store persistent information about the workspace configuration.

```
3.3 Adding Packages to the Workspace

In order to build software with Catkin, it needs to be added to the workspace’s `source space`. You can either download some existing packages, or create one or more empty ones. As shown above, the default path for a Catkin `source space` is `.src` relative to the workspace root. A standard Catkin package is simply a directory with a `CMakeLists.txt` file and a `package.xml` file. For more information on Catkin packages see `workspace mechanics`. The shell interaction below shows the creation of four empty packages: `pkg_a`, `pkg_b`, `pkg_c`, and `pkg_d`:

```
cd /tmp/quickstart_ws/src
    # Navigate to the source space
catkin create pkg pkg_a
        # Populate the source space with packages..
    ...
catkin create pkg pkg_b
    catkin create pkg pkg_c --catkin-deps pkg_a
    catkin create pkg pkg_d --catkin-deps pkg_a pkg_b
    catkin list
        # List the packages in the workspace
```

After these operations, your workspace’s local directory structure would look like the following (to two levels deep):

```
cd /tmp/quickstart_ws  # Navigate to the workspace root
tree -aL 2            # Show prebuild directory tree
```

```
| .catkin_tools
|    | CATKIN_IGNORE
```

(continues on next page)
Now that there are some packages in the workspace, Catkin has something to build.

**Note:** Catkin utilizes an “out-of-source” and “aggregated” build pattern. This means that temporary or final build products will never be placed in a package’s source directory (or anywhere in the source space). Instead all build directories are aggregated in the build space and all final build products like executables, libraries, etc., will be put in the devel space.

### 3.4 Building the Workspace

Since the catkin workspace has already been initialized, you can call `catkin build` from any directory contained within it. If it had not been initialized, then `catkin build` would need to be called from the workspace root. Based on the default configuration, it will locate the packages in the source space and build each of them.

```
catkin build
```

Calling `catkin build` will generate build and devel directories (as described in the config summary above) and result in a directory structure like the following (up to one level deep):

```
cd /tmp/quickstart_ws
```
```
tree -aL 2
```

(continues on next page)
Intermediate build products (CMake cache files, Makefiles, object files, etc.) are generated in the build directory, or build space and final build products (libraries, executables, config files) are generated in the devel directory, or devel space. For more information on building and customizing the build configuration see the build verb and config verb documentation.

3.5 Loading the Workspace Environment

In order to properly “use” the products of the workspace, its environment needs to be loaded. Among other environment variables, sourcing a Catkin setup file modifies the CMAKE_PREFIX_PATH environment variable, which will affect workspace chaining as described in the earlier section.

Setup files are located in one of the result spaces generated by your workspace. Both the devel space or the install space are valid result spaces. In the default build configuration, only the devel space is generated. You can load the environment for your respective shell like so:

```bash
source /tmp/quickstart_ws/devel/setup.bash # Load the workspace's environment
```

At this point you should be able to use products built by any of the packages in your workspace.

**Note:** Any time the member packages change in your workspace, you will need to re-run the source command.

Loading the environment from a Catkin workspace can set arbitrarily many environment variables, depending on which “environment hooks” the member packages define. As such, it’s important to know which workspace environment is loaded in a given shell.

It’s not unreasonable to automatically source a given setup file in each shell for convenience, but if you do so, it’s good practice to pay attention to the Extending value in the Catkin config summary. Any Catkin setup file will modify the CMAKE_PREFIX_PATH environment variable, and the config summary should catch common inconsistencies in the environment.
3.6 Cleaning Workspace Products

Instead of using dangerous commands like `rm -rf build devel` in your workspace when cleaning build products, you can use the `catkin clean` command. Just like the other verbs, `catkin clean` is context-aware, so it only needs to be called from a directory under the workspace root.

In order to clean the **build space** and **devel space** for the workspace, you can use the following command:

```
catkin clean  # Clean all the build products
```

For more information on less aggressive cleaning options see the `clean verb` documentation.
This is a non-exhaustive list of some common and useful invocations of the `catkin` command. All of the commands which do not explicitly specify a workspace path (with `--workspace`) are assumed to be run from within a directory contained by the target workspace. For thorough documentation, please see the chapters on each verb.

### 4.1 Initializing Workspaces

Initialize a workspace with a default layout (`src/build/devel`) in the current directory:

- `catkin init`
- `catkin init --workspace .`
- `catkin config --init`
- `mkdir src && catkin build`

... with a default layout in a different directory:

- `catkin init --workspace /tmp/path/to/my_catkin_ws`

... which explicitly extends another workspace:

- `catkin config --init --extend /opt/ros/indigo`

Initialize a workspace with a source space called `other_src`:

- `catkin config --init --source-space other_src`

... or a workspace with build, devel, and install space ending with the suffix `_alternate`:

- `catkin config --init --space-suffix _alternate`

### 4.2 Configuring Workspaces

View the current configuration:
• catkin config

**Setting and unsetting CMake options:**
• catkin config --cmake-args -DENABLE_CORBA=ON -DCORBA_IMPLEMENTATION=OMNIORB
• catkin config --no-cmake-args

**Toggle installing to the specified install space:**
• catkin config --install

### 4.3 Building Packages

**Build all the packages:**
• catkin build

... one at a time, with additional debug output:
• catkin build -p 1

... and force CMake to re-configure for each one:
• catkin build --force-cmake

**Build a specific package and its dependencies:**
• catkin build my_package

... or ignore its dependencies:
• catkin build my_package --no-deps

**Build the package containing the current working directory:**
• catkin build --this

... but don’t rebuild its dependencies:
• catkin build --this --no-deps

**Build packages with additional CMake args:**
• catkin build --cmake-args -DCMAKE_BUILD_TYPE=Debug

... and save them to be used for the next build:
• catkin build --save-config --cmake-args -DCMAKE_BUILD_TYPE=Debug

**Build all packages in a given directory:**
• catkin build $(catkin list -u -d /path/to/folder)

... or in the current folder:
• catkin build $(catkin list -u -d .)

### 4.4 Testing Packages

**Test all the packages:**
• catkin test

... one at a time, with live output:
• `catkin build -p 1 -i`

Test a specific package:

• `catkin test my_package`

... or a specific test target of a package

• `catkin test -t my_target my_package`

### 4.5 Cleaning Build Products

Blow away the build, devel, and install spaces (if they exist):

• `catkin clean`

... or just the build space:

• `catkin clean --build`

... or just clean a single package:

• `catkin clean PKGNAME`

... or just delete the build directories for packages which have been disabled or removed:

• `catkin clean --orphans`

### 4.6 Controlling Color Display

Disable colors when building in a shell that doesn’t support it (like IDEs):

• `catkin --no-color build`

... or enable it for shells that don’t know they support it:

• `catkin --force-color build`

### 4.7 Profile Cookbook

Create “Debug” and “Release” profiles and then build them in independent build and devel spaces:

```bash
catkin config --profile debug -x _debug --cmake-args -DCMAKE_BUILD_TYPE=Debug
catkin config --profile release -x _release --cmake-args -DCMAKE_BUILD_TYPE=Release
catkin build --profile debug
catkin build --profile release
```

Quickly build a package from scratch to make sure all of its dependencies are satisfied, then clean it:

```bash
catkin config --profile my_pkg -x _my_pkg_test
catkin build --profile my_pkg my_pkg
catkin clean --profile my_pkg --all
```
4.8 Manipulating Workspace Chaining

Change from implicit to explicit chaining:

```
catkin clean
catkin config --extend /opt/ros/indigo
```

Change from explicit to implicit chaining:

```
catkin clean
catkin config --no-extend
```

4.9 Building With Other Job Servers

Build with `distcc`:

```
CC="distcc gcc" CXX="distcc g++" catkin build -p$(distcc -j) -j$(distcc -j) --no-jobserver
```

4.10 Changing Package's Build Type

Set the build type to `cmake` in the `package.xml` file's `<export/>` section:

```
<export>
  <build_type>cmake</build_type>
</export>
```
CHAPTER 5

Migrating from catkin_make

5.1 Important Distinctions between catkin_make and catkin build

Unlike catkin_make, the catkin command-line tool is not just a thin wrapper around the cmake and make commands. The catkin build command builds each package in a workspace’s source space in isolation in order to prevent build-time cross-talk. As such, in its simplest use, catkin build behaves similarly to a parallelized version of catkin_make_isolated.

While there are many more features in catkin_tools described in the rest of the documentation, this chapter provides details on how to switch from using catkin_make and catkin_make_isolated. This chapter does not describe advanced features that catkin_tools provides over catkin_make and catkin_make_isolated. For a quick overview of what you can do with catkin build, see the Cheat Sheet.

5.1.1 Implications of Isolation

Build isolation has the following implications for both catkin_make_isolated and catkin build:

- There is no “top-level” CMakeLists.txt file in the source space.
- Each package in a catkin_tools workspace has its own isolated build space.
- Packages built with catkin build can not access variables defined in other Catkin packages in the same workspace.
- All targets in each of a package’s dependencies are guaranteed to have been built before the current package.
- Packages do not need to define target dependencies on ROS messages built in other packages.
- It passes the same CMake command line arguments to multiple packages.
- Plain CMake packages can be built if they each have a package.xml file with the appropriate <build_type> tag.
5.1.2 Additional Differences with `catkin build`

In addition to the differences due to isolation, `catkin build` is also different from `catkin_make_isolated` in the following ways:

- It builds packages in parallel, using an internal job server to distribute load.
- It puts products into hidden directories, and then symbolically links them into the `devel space` (by default).
- It stores persistent configuration options in a `.catkin_tools` directory at the root of your workspace.
- It passes `--no-warn-unused-cli` to the `cmake` command since not all packages accept the same CMake arguments.
- It generates `.catkin` files where each source package is listed, individually, instead of just listing the source space for the workspace. This leads to similar `ROS_PACKAGE_PATH` variables which list each package source space.

5.2 Step-by-Step Migration

Most problems users will encounter when migrating from `catkin_make` to `catkin build` are due to hidden bugs in packages which previously relied on side-effects from their dependencies to build. The best way to debug these problems before switching to the entirely new tool, is to use `catkin_make_isolated` first. Note that all three of these tools can share `source spaces`, but they must use their own build, devel, and install spaces.

5.2.1 1. Verify that your packages already build with `catkin_make`:

To make iterating easier, use `catkin_make` with build and devel spaces with the suffix `_cm` so that they do not collide with the other build tools:

```
cd /path/to/ws
catkin_make --build build_cm --cmake-args -DCATKIN_DEVEL_PREFIX=devel_cm -DCMAKE_INSTALL_PREFIX=install_cm [CMAKE_ARGS...] --make-args [MAKE_ARGS...]
```

If your packages build and other appropriate tests pass, continue to the next step.

5.2.2 2. Verify that your packages build in isolation:

Use `catkin_make_isolated` with build and devel spaces with the suffix `_cmi`, and make sure your packages build in isolation. This is where you are most likely to discover bugs in your packages' `CMakeLists.txt` files. Fix each problem, using the troubleshooting advice later in this chapter.

```
cd /path/to/ws
catkin_make_isolated --build build_cmi --devel devel_cmi --merge --cmake-args [CMAKE_ARGS...] --make-args [MAKE_ARGS...]
```

Once your packages build (and other appropriate tests pass), continue to the next step.

5.2.3 3. Build with `catkin build`:

Finally, you can verify that your packages build with `catkin build`, using build and devel spaces with the suffix `_cb`. Since `catkin build` stores build configuration, you only need to set your CMake and Make args once:
Then you can build with `catkin build`. If issues arise, try to use the troubleshooting advice later in this chapter and in the main Troubleshooting chapter.

Once the build succeeds and your appropriate tests pass, you can go on to continue using `catkin build`!

## 5.3 Migration Troubleshooting

When migrating from `catkin_make` to `catkin build`, the most common problems come from Catkin packages taking advantage of package cross-talk in the CMake configuration stage.

Many Catkin packages implicitly rely on other packages in a workspace to declare and find dependencies. When switching from `catkin_make`, users will often discover these bugs.

### 5.3.1 Common Issues

**Unknown CMake command “catkin_package”**

If `find_package(catkin REQUIRED ...)` isn’t called, then the `catkin_package()` macro will not be available. If such a package builds with `catkin_make`, it’s because it’s relying on another package in the same workspace to do this work.

**Compilation Errors (Missing Headers)**

Compilation errors can occur if required headers are not found. If your package includes headers from `$(catkin_INCLUDE_DIRS)`, make sure that package is finding the right Catkin packages in `find_package(catkin COMPONENTS ...)`.

If your package includes headers from other libraries, make sure those libraries are found and those CMake variables are defined.

**Linker Errors (Undefined References)**

Linker errors are due to targets not being linked to required libraries. If your target links against `$(catkin_LIBRARIES)`, make sure that package is finding the right Catkin packages in `find_package(catkin COMPONENTS ...)`.

If your target links against other libraries, make sure those libraries are found and those CMake variables are defined.

- https://github.com/catkin/catkin_tools/issues/228
Targets Not Being Built

It is critical for Catkin-based packages to call \texttt{catkin\_package()} before any targets are defined. Otherwise your targets will not be built into the \texttt{devel space}. Previously with \texttt{catkin\_make}, as long as some package called \texttt{catkin\_package()} before your package was configured, the appropriate target destinations were defined.

- https://github.com/catkin/catkin\_tools/issues/220

Compiler Options Aren’t Correct

Your program might fail to build or fail to run due to incorrect compiler options. Sometimes these compiler options are needed to use a dependency, but aren’t made available to the dependent package.

With \texttt{catkin\_make}, if a package sets certain compiler options, such as:

```cpp
set(CMAKE_CXX_FLAGS "-std=c++ ${CMAKE_CXX_FLAGS}"
```

These options will be set for every package in the topological sort which is built after it, even packages which don’t depend on it.

With \texttt{catkin\_build}, however, these effects are isolated, so even the packages that need these options will not get them. The \texttt{catkin\_package()} macro already provides options for exporting libraries and include directories, but it does not have an option for CMake variables.

To export such settings (or even execute code), the \texttt{CFG\_EXTRAS} option must be used with an accompanying CMake file. For more information on this option, see the \texttt{catkin\_package()} documentation.

- https://github.com/catkin/catkin\_tools/issues/210
- https://github.com/carpe-noctem-cassel/cnc\_msl/pull/1

5.3.2 Uncommon Issues

Exporting Build Utilities

Some Catkin packages provide build tools at configuration time, like scripts for generating code or downloading resources from the internet. These packages need to export absolute paths to such tools both when used in a workspace and when installed.

For example, when using in a source space, the build tools from package \texttt{my\_build\_util} would be found at \texttt{${CMAKE\_CURRENT\_SOURCE\_DIR}/cmake}, but when installed, they would be found in \texttt{${my\_build\_util\_DIR}}.

With \texttt{catkin\_make}, the path to these tools could be set to either the source or install space in the provider package just by setting a CMake variable, which would be “leaked” to all subsequently built packages.

With \texttt{catkin\_build}, these paths need to be properly exported with \texttt{CFG\_EXTRAS}. A way to do this that works both out of a workspace and install is shown below:

**Listing 1: my\_build\_util-extras.cmake.em**

```cmake
# generated from stdr\_common/cmake/stdr\_common-extras.cmake.em
@if DEVELSPACE@
# set path to source space
set(my\_build\_util\_EXTRAS\_DIR "@\(CMAKE\_CURRENT\_SOURCE\_DIR\)/cmake")
@end
```

(continues on next page)
Exporting Non-Standard Library Output Locations or Prefixes

Some users may choose to build library targets with non-standard output locations or prefixes. However, the normal `catkin_package()` macro cannot export libraries with such paths across packages.

Again, we can use the `CFG_EXTRAS` option to append the special library to the `${PROJECT_NAME}_LIBRARIES` variable that `catkin_package()` exports to other packages.

Listing 2: CMakeLists.txt
```cmake
# NOTE: Not specified here, but in extras file
CFG_EXTRAS my-extras.cmake

set_target_properties(
  ${PROJECT_NAME} PROPERTIES
  PREFIX ""
  LIBRARY_OUTPUT_DIRECTORY ${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE_PYTHON_DESTINATION}
)
```

Listing 3: my.cmake.in
```cmake
find_library(${PROJECT_NAME}_LIBRARY
  NAMES ${PROJECT_NAME}
  PATHS "${PROJECT_NAME}_DIR"/../../../@CATKIN_GLOBAL_LIB_DESTINATION@/
  NO_DEFAULT_PATH)

if(${PROJECT_NAME}_LIBRARY)
  # Multiple CMake projects case (i.e. 'catkin build'):
  # - The target has already been built when its dependencies require it
  # - Specify full path to found library
  list(APPEND ${PROJECT_NAME}_LIBRARIES ${PROJECT_NAME}_LIBRARY)
else()
  # Single CMake project case (i.e. 'catkin_make'):
  # - The target has not been built when its dependencies require it
  # - Specify target name only
  list(APPEND ${PROJECT_NAME}_LIBRARIES ${PROJECT_NAME})
endif()
```

- https://github.com/catkin/catkin_tools/issues/128

Controlling Python Version

On some platforms, there are multiple versions of Python, and Catkin’s internal setup file generation might pick the wrong one. For `catkin_make`, this is sometimes solved on a given platform by creating a shell alias which sets the `PYTHON_EXECUTABLE` CMake variable.

5.3. Migration Troubleshooting
For `catkin build`, however, you can create a `verb alias` like the one below, which overrides the default behavior of `catkin build` even in new workspaces.

```
build: build -DPYTHON_EXECUTABLE=/usr/bin/python2.7
```

See `Verb Aliasing` for more details.

- https://github.com/catkin/catkin_tools/issues/166

## 5.4 IDE Integration

Since all packages are built in isolation with `catkin build`, you can’t rely on CMake’s IDE integration to generate a single project for your entire workspace.

## 5.5 CLI Comparison with `catkin_make` and `catkin_make_isolated`

Below are tables mapping `catkin_make` and `catkin_make_isolated` arguments into `catkin` arguments. Note that some `catkin_make` options can only be achieved with the `catkin config` verb.

<table>
<thead>
<tr>
<th>catkin_make ...</th>
<th>catkin ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-C PATH</code></td>
<td>`-w PATH [build</td>
</tr>
<tr>
<td><code>--source PATH</code></td>
<td><code>config --source-space PATH</code></td>
</tr>
<tr>
<td><code>--build PATH</code></td>
<td><code>config --build-space PATH</code></td>
</tr>
<tr>
<td><code>--use-ninja</code></td>
<td><code>not yet available</code></td>
</tr>
<tr>
<td><code>--force-cmake</code></td>
<td><code>build --force-cmake</code></td>
</tr>
<tr>
<td><code>--pkg PKG [PKG ...]</code></td>
<td><code>build --no-deps PKG [PKG ...]</code></td>
</tr>
<tr>
<td><code>--only-pkg-with-deps PKG [PKG ...]</code></td>
<td><code>build PKG [PKG ...]</code></td>
</tr>
<tr>
<td><code>--cmake-args ARG [ARG ...]</code></td>
<td><code>build --cmake-args ARG [ARG ...]</code></td>
</tr>
<tr>
<td><code>--make-args ARG [ARG ...]</code></td>
<td><code>build --make-args ARG [ARG ...]</code></td>
</tr>
<tr>
<td><code>--override-build-tool-check</code></td>
<td><code>build --override-build-tool-check</code></td>
</tr>
<tr>
<td><code>ARG [ARG ...]</code></td>
<td><code>build --make-args ARG [ARG ...]</code></td>
</tr>
<tr>
<td><code>install</code></td>
<td><code>config --install</code></td>
</tr>
<tr>
<td><code>-DCATKIN_DEVEL_PREFIX=PATH</code></td>
<td><code>config --devel-space PATH</code></td>
</tr>
<tr>
<td><code>-DCATKIN_INSTALL_PREFIX=PATH</code></td>
<td><code>config --install-space PATH</code></td>
</tr>
<tr>
<td><code>-DCATKIN_WHITELIST_PACKAGES=&quot;PKG[;PKG ...]&quot;</code></td>
<td><code>config --whitelist PKG [PKG ...]</code></td>
</tr>
</tbody>
</table>

1 These options require a subsequent call to `catkin build`, and the options will continue to persist until changed.
2 These options, if passed to `catkin build` only affect that invocation. If passed to `catkin config`, they will persist to subsequent calls to `catkin build`.
<table>
<thead>
<tr>
<th>catkin_make_isolated...</th>
<th>catkin...</th>
</tr>
</thead>
<tbody>
<tr>
<td>-C PATH</td>
<td>-w PATH [build</td>
</tr>
<tr>
<td>--source PATH</td>
<td>config --source-space PATH</td>
</tr>
<tr>
<td>--build PATH</td>
<td>config --build-space PATH</td>
</tr>
<tr>
<td>--devel PATH</td>
<td>config --devel-space PATH</td>
</tr>
<tr>
<td>--merge PATH</td>
<td>config --devel-layout merged</td>
</tr>
<tr>
<td>--install-space PATH</td>
<td>config --install-space PATH</td>
</tr>
<tr>
<td>--use-ninja</td>
<td>not yet available</td>
</tr>
<tr>
<td>--install</td>
<td>config --install</td>
</tr>
<tr>
<td>--force-cmake</td>
<td>build --force-cmake</td>
</tr>
<tr>
<td>--no-color</td>
<td>build --no-color</td>
</tr>
<tr>
<td>--pkg PKG [PKG ...]</td>
<td>build --no-deps PKG [PKG ...]</td>
</tr>
<tr>
<td>--from-pkg PKG</td>
<td>build --start-with PKG</td>
</tr>
<tr>
<td>--only-pkg-with-deps PKG [PKG ...]</td>
<td>build PKG [PKG ...]</td>
</tr>
<tr>
<td>--cmake-args ARG [ARG ...]</td>
<td>build --cmake-args ARG [ARG ...]</td>
</tr>
<tr>
<td>--make-args ARG [ARG ...]</td>
<td>build --make-args ARG [ARG ...]</td>
</tr>
<tr>
<td>--catkin-make-args ARG [ARG ...]</td>
<td>build --catkin-make-args ARG [ARG ...]</td>
</tr>
<tr>
<td>--override-build-tool-check</td>
<td>build --override-build-tool-check</td>
</tr>
</tbody>
</table>
This chapter defines the organization, composition, and use of Catkin workspaces. Catkin workspaces enable rapid simultaneous building and executing of numerous interdependent projects. These projects do not need to share the same build tool, but they do need to be able to either build or install to a FHS tree.

Unlike integrated development environments (IDEs) which normally only manage single projects, the purpose of Catkin is to enable the simultaneous compilation of numerous independently-authored projects.

### 6.1 Workspace Configuration

Most `catkin` commands which modify a workspace’s configuration will display the standard configuration summary, as shown below:

```bash
$ cd /tmp/path/to/my_catkin_ws
$ catkin config
--------------------------------------------------------------
Profile:         default
Extending:       None
Workspace:       /tmp/path/to/my_catkin_ws
--------------------------------------------------------------
Source Space:    [exists] /tmp/path/to/my_catkin_ws/src
Log Space:       [missing] /tmp/path/to/my_catkin_ws/logs
Build Space:     [missing] /tmp/path/to/my_catkin_ws/build
Devel Space:     [missing] /tmp/path/to/my_catkin_ws/devel
Install Space:   [unused] /tmp/path/to/my_catkin_ws/install
DESTDIR:         [unused] None
--------------------------------------------------------------
Devel Space Layout:     linked
Install Space Layout:   merged
--------------------------------------------------------------
Additional CMake Args: None
Additional Make Args:  None
Additional catkin Make Args: None
```

(continues on next page)
This summary describes the layout of the workspace as well as other important settings which influence build and execution behavior. Each of these options can be modified either with the `config` verb’s options described in the full command-line usage or by changing environment variables. The summary is composed of the following sections:

### 6.1.1 Overview Section

- **Profile** – The name of this configuration.
- **Extending** – Describes if your current configuration will extend another Catkin workspace, and through which mechanism it determined the location of the extended workspace:
  - *No Chaining*
  - *Implicit Chaining* – Derived from the `CMAKE_PREFIX_PATH` environment variable.
  - *Cached Implicit Chaining* – Derived from the `CMAKE_PREFIX_PATH` CMake cache variable.
  - *Explicit Chaining* – Specified by `catkin config --extend`
- **Workspace** – The path to the workspace.
- **Source Space** – The subdirectory containing the source packages.
- **Build Space** – The subdirectory containing the intermediate build products for each package.
- **Devel Space** – The subdirectory containing the final build products which can be used to run code, but relies on the presence of the source space.
- **Install Space** – The subdirectory containing the final build products which can be used to run code, but is entirely self-contained.
- **DESTDIR** – An optional prefix to the install space as defined by GNU Standards

### 6.1.2 Build Product Layout Section

- **Devel Space Layout** – The organization of the *devel space*.
  - *Linked* – Write products from each package into independent isolated FHS trees, and symbolically link them into a merged FHS tree. For more details, see *Linked Devel Space*.
  - *Merged* – Write products from all packages to a single FHS tree. This is most similar to the behavior of `catkin_make`.
  - *Isolated* – Write products from each package into independent isolated FHS trees. This is most similar to the behavior of `catkin_make_isolated`.
- **Install Packages** – Enable creating and installation into the install space.
- **Isolate Installs** – Installs products into individual FHS subdirectories in the install space.
6.1.3 Build Tool Arguments Section

- **Additional CMake Args** – Arguments to be passed to CMake during the *configuration* step for all packages to be built.
- **Additional Make Args** – Arguments to be passed to Make during the *build* step for all packages to be built.
- **Additional catkin Make Args** – Similar to **Additional Make Args** but only applies to Catkin packages.
- **Internal Make Job Server** – Whether or not the internal job server should be used to coordinate parallel build jobs.
- **Cache Job Environments** – Whether or not environment variables should be cached between build jobs.

6.1.4 Package Filter Section

- **Package Whitelist** – Packages that will be built with a bare call to `catkin build`.
- **Package Blacklist** – Packages that will not be built unless explicitly named.

6.1.5 Notes Section

The summary will sometimes contain notes about the workspace or the action that you’re performing, or simply tell you that the workspace configuration appears valid.

6.1.6 Warnings Section

If something is wrong with your configuration such as a missing source space, an additional section will appear at the bottom of the summary with details on what is wrong and how you can fix it.

6.2 Workspace Anatomy

A standard catkin workspace, as defined by REP-0128, is a directory with a prescribed set of “spaces”, each of which is contained within a directory under the workspace root. The spaces that comprise the workspace are described in the following sections. In addition to the directories specified by REP-0128, catkin_tools also adds a visible `logs` directory and a hidden `.catkin_tools` directory. The `.catkin_tools` directory stores persistent build configuration and profiles.

<table>
<thead>
<tr>
<th>Space</th>
<th>Default Path</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Space</td>
<td>./src</td>
<td>Source code for all the packages.</td>
</tr>
<tr>
<td>Log Space</td>
<td>./logs</td>
<td>Logs from building and cleaning packages.</td>
</tr>
<tr>
<td>Build Space</td>
<td>./build</td>
<td>Intermediate build products for each package.</td>
</tr>
<tr>
<td>Devel Space</td>
<td>./devel</td>
<td>FHS tree or trees containing all final build products.</td>
</tr>
<tr>
<td>Install Space</td>
<td>./install</td>
<td>FHS tree or trees containing products of <em>install</em> targets.</td>
</tr>
</tbody>
</table>

6.2.1 source space

The **source space** contains the source code for all of the packages to be built in the workspace, as such, it is the only directory required to build a workspace. The **source space** is also the only directory in the catkin workspace which is not modified by any catkin command verb. No build products are written to the source space, they are all built “out-of-source” in the **build space**, described in the next section. You can consider the **source space** to be read-only.
6.2.2 log space

The `catkin` command generates a log space, called `logs` by default, which contains build logs for each package. Logs for each package are written in subdirectories with the same name as the package.

The latest log for each verb and stage in a given package’s log directory is also written with the format:

```
{VERB}.{STAGE}.log
```

Each previous logfile has the following format, where `{INDEX}` begins at 000 and increases with each execution of that verb and stage:

```
{VERB}.{STAGE}.{INDEX}.log
```

6.2.3 build space

Intermediate build products are written in the **build space**. The **build space** contains an isolated build directory for each package, as well as the log files which capture the output from each build stage. It is from these directories where commands like `cmake` and `make` are run.

6.2.4 devel space

Build products like executables, libraries, pkg-config files, and CMake config files, are generated in the **devel space**. The **devel space** is organized as an FHS tree.

Some build tools simply treat the **devel space** as an install prefix, but other buildtools like `catkin`, itself, can build targets directly into the **devel space** in order to skip the additional install step. For such packages, executing programs from the **devel space** sometimes requires that the source space is still available.

At the root of the **devel space** is a set of environment setup files which can be “sourced” in order to properly execute the space’s products.

6.2.5 install space

Finally, if the workspace is configured to install packages, the each will be installed into the **install space**. The **install space** has an FHS layout like the **devel space**, except it is entirely self-contained.

6.2.6 Additional Files Generated by `catkin_tools`

**Configuration Directory**

In addition to the standard workspace structure, `catkin_tools` also adds a marker directory called `.catkin_tools` at the root of the workspace. This directory both acts as a marker for the root of the workspace and contains persistent configuration information.

This directory contains subdirectories representing different configuration profiles, and inside of each profile directory are YAML files which contain verb-specific metadata. It additionally contains a file which lists the name of the active configuration profile if it is different from `default`. 

---

Chapter 6. Workspace Mechanics
Environment Setup Files

The FHS trees of the **devel space** and **install space** also contain several environment “setup” scripts. These setup scripts are intended to make it easier to use the resulting FHS tree for building other source code or for running programs built by the packages in the workspace.

The setup script can be used like this in **bash**:

```bash
$ source /path/to/workspace/devel/setup.bash
```

Or like this in **zsh**:

```zsh
% source /path/to/workspace/devel/setup.zsh
```

Sourcing these setup scripts adds this workspace and any “underlaid” workspaces to your environment, prefixing several environment variables with the appropriate local workspace folders.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMAKE_PREFIX_PATH</td>
<td>Used by CMake to find development packages, and used by Catkin for workspace chaining.</td>
</tr>
<tr>
<td>CPATH(^4)</td>
<td>Used by GCC to search for development headers.</td>
</tr>
<tr>
<td>LD_LIBRARY_PATH(^1)</td>
<td>Search path for dynamically loadable libraries.</td>
</tr>
<tr>
<td>DYLD_LIBRARY_PATH(^2)</td>
<td>Search path for dynamically loadable libraries.</td>
</tr>
<tr>
<td>PATH</td>
<td>Search path for executables.</td>
</tr>
<tr>
<td>PKG_CONFIG_PATH</td>
<td>Search path for pkg-config files.</td>
</tr>
<tr>
<td>PYTHONPATH</td>
<td>Search path for Python modules.</td>
</tr>
</tbody>
</table>

The setup scripts will also execute any Catkin “env-hooks” exported by packages in the workspace. For example, this is how *roslib* sets the **ROS_PACKAGE_PATH** environment variable.

\(^4\) Only in versions of *catkin <= 0.7.0* (ROS Kinetic), see the changelog

\(^1\) GNU/Linux Only

\(^2\) Mac OS X Only
Note: Like the **devel space**, the **install space** includes `setup.*` and related files at the top of the file hierarchy. This is not suitable for some packaging systems, so this can be disabled by passing the `-DCATKIN_BUILD_BINARY_PACKAGE="1"` option to `cmake` using the `--cmake-args` option for this verb. Though this will suppress the installation of the setup files, you will lose the functionality provided by them, namely extending the environment and executing environment hooks.

### 6.3 Source Packages and Dependencies

A package is any folder which contains a `package.xml` as defined by the ROS community in ROS Enhancement Proposals REP-0127 and REP-0140.

The `catkin build` command builds packages in the topological order determined by the dependencies listed in the package's `package.xml` file. For more information on which dependencies contribute to the build order, see the `build` verb documentation.

Additionally, the `build_type` tag is used to determine which build stages to use on the package. Supported build types are listed in [Build Types](#). Packages without a `build_type` tag are assumed to be catkin packages.

For example, plain CMake packages can be built by adding a `package.xml` file to the root of their source tree with the `build_type` flag set to `cmake` and appropriate `build_depend` and `run_depend` tags set, as described in REP-0136. This can been done to build packages like opencv, pcl, and flann.

### 6.4 Workspace Chaining / Extending

An important property listed in the configuration which deserves attention is the summary value of the `Extending` property. This affects which other collections of libraries and packages which will be visible to your workspace. This is process called “workspace chaining.”

Above, it’s mentioned that the Catkin setup files export numerous environment variables, including `CMAKE_PREFIX_PATH`. Since CMake 2.6.0, the `CMAKE_PREFIX_PATH` is used when searching for include files, binaries, or libraries using the `FIND_PACKAGE()`, `FIND_PATH()`, `FIND_PROGRAM()`, or `FIND_LIBRARY()` CMake commands.

As such, this is also the primary way that Catkin “chains” workspaces together. When you build a Catkin workspace for the first time, it will automatically use `CMAKE_PREFIX_PATH` to find dependencies. After that compilation, the value will be cached internally by each project as well as the Catkin setup files and they will ignore any changes to your `CMAKE_PREFIX_PATH` environment variable until they are cleaned.

Note: Workspace chaining is the act of putting the products of one workspace A in the search scope of another workspace B. When describing the relationship between two such chained workspaces, A and B, it is said that workspace B extends workspace A and workspace A is extended by workspace B. This concept is also sometimes referred to as “overlaying” or “inheriting” a workspace.

Similarly, when you source a Catkin workspace’s setup file from a workspace’s **devel space** or **install space**, it prepends the path containing that setup file to the `CMAKE_PREFIX_PATH` environment variable. The next time you initialize a workspace, it will extend the workspace that you previously sourced.

This makes it easy and automatic to chain workspaces. Previous tools like `catkin_make` and `catkin_make_isolated` had no easy mechanism for either making it obvious which workspace was being extended, nor did they provide features to explicitly extend a given workspace. This means that for users were unaware of Catkin’s use of `CMAKE_PREFIX_PATH`. 

30 Chapter 6. Workspace Mechanics
Since it’s not expected that 100% of users will read this section of the documentation, the catkin program adds both configuration consistency checking for the value of CMAKE_PREFIX_PATH and makes it obvious on each invocation which workspace is being extended. Furthermore, the catkin command adds an explicit extension interface to override the value of $CMAKE_PREFIX_PATH with the catkin config --extend command.

---

**Note:** While workspaces can be chained together to add search paths, invoking a build in one workspace will not cause products in any other workspace to be built.

---

The information about which workspace to extend can come from a few different sources, and can be classified in one of three ways:

### 6.4.1 No Chaining

This is what is shown in the above example configuration and it implies that there are no other Catkin workspaces which this workspace extends. The user has neither explicitly specified a workspace to extend, and the CMAKE_PREFIX_PATH environment variable is empty:

```
Extending: None
```

### 6.4.2 Implicit Chaining via CMAKE_PREFIX_PATH Environment or Cache Variable

In this case, the catkin command is *implicitly* assuming that you want to build this workspace against resources which have been built into the directories listed in your CMAKE_PREFIX_PATH environment variable. As such, you can control this value simply by changing this environment variable.

For example, ROS users who load their system’s installed ROS environment by calling something similar to `source /opt/ros/indigo/setup.bash` will normally see an Extending value such as:

```
Extending: [env] /opt/ros/indigo
```

If you don’t want to extend the given workspace, unsetting the CMAKE_PREFIX_PATH environment variable will change it back to none. Once you have built your workspace once, this CMAKE_PREFIX_PATH will be cached by the underlying CMake buildsystem. As such, the Extending status will subsequently describe this as the “cached” extension path:

```
Extending: [cached] /opt/ros/indigo
```

Once the extension mode is cached like this, you must use catkin clean to before changing it to something else.

### 6.4.3 Explicit Chaining via catkin config --extend

This behaves like the above implicit chaining except it means that this workspace is *explicitly* extending another workspace and the workspaces which the other workspace extends, recursively. This can be set with the catkin config --extend command. It will override the value of CMAKE_PREFIX_PATH and persist between builds.

```
Extending: [explicit] /tmp/path/to/other_ws
```
The current release of catkin_tools supports building two types of packages:

- **Catkin** – CMake packages that use the Catkin CMake macros
- **CMake** – “Plain” CMake packages

There is currently limited support for adding other build types. For information on extending catkin_tools to be able to build other types of packages, see *Adding New Build Types*. Below are details on the stages involved in building a given package for each of the currently-supported build types.

### 7.1 Catkin

Catkin packages are CMake packages which utilize the Catkin CMake macros for finding packages and defining configuration files.

#### 7.1.1 Configuration Arguments

- `--cmake-args`
- `--make-args`
- `--catkin-make-args`
## 7.1.2 Build Stages

<table>
<thead>
<tr>
<th>First</th>
<th>Subsequent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td></td>
<td>Create package build space if it doesn’t exist.</td>
</tr>
<tr>
<td>cmake</td>
<td>check</td>
<td>Run CMake configure step <strong>once</strong> for the first build and the `cmake_check_build_system` target for subsequent builds unless the <strong>--force-cmake</strong> argument is given.</td>
</tr>
<tr>
<td>preclean</td>
<td>optional</td>
<td>Run the <strong>clean</strong> target before building.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is only done with the <strong>--pre-clean</strong> option.</td>
</tr>
<tr>
<td>make</td>
<td></td>
<td>Build the default target with GNU make.</td>
</tr>
<tr>
<td>install</td>
<td>optional</td>
<td>Run the <strong>install</strong> target after building.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is only done with the <strong>--install</strong> option.</td>
</tr>
<tr>
<td>setupgen</td>
<td></td>
<td>Generate a <strong>setup.sh</strong> file to “source” the result space.</td>
</tr>
<tr>
<td>envgen</td>
<td></td>
<td>Generate an <strong>env.sh</strong> file for loading the result space’s environment.</td>
</tr>
</tbody>
</table>
7.2 CMake

7.2.1 Configuration Arguments

- `--cmake-args`
- `--make-args`
## 7.2.2 Build Stages

<table>
<thead>
<tr>
<th>First</th>
<th>Subsequent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td></td>
<td>Create package build space if it doesn’t exist.</td>
</tr>
<tr>
<td>cmake</td>
<td>check</td>
<td>Run CMake configure step <strong>once</strong> for the first build and the <code>cmake_check_build_system</code> target for subsequent builds unless the <code>--force-cmake</code> argument is given.</td>
</tr>
<tr>
<td>preclean</td>
<td>optional</td>
<td>Run the <code>clean</code> target before building. This is only done with the <code>--pre-clean</code> option.</td>
</tr>
<tr>
<td>make</td>
<td></td>
<td>Build the default target with GNU make.</td>
</tr>
<tr>
<td>install</td>
<td></td>
<td>Run the <code>install</code> target after building, and install products to the <strong>devel space</strong>. If the <code>--install</code> option is given, products are installed to the <strong>install space</strong> instead.</td>
</tr>
<tr>
<td>setupgen</td>
<td></td>
<td>Generate a <code>setup.sh</code> file if necessary.</td>
</tr>
</tbody>
</table>
8.1 Configuration Summary Warnings

The catkin tool is capable of detecting some issues or inconsistencies with the build configuration automatically. In these cases, it will often describe the problem as well as how to resolve it. The catkin tool will detect the following issues automatically.

8.1.1 Missing Workspace Components

- Uninitialized workspace (missing .catkin_tools directory)
- Missing source space as specified by the configuration

8.1.2 Inconsistent Environment

- The CMAKE_PREFIX_PATH environment variable is different than the cached CMAKE_PREFIX_PATH
- The explicitly extended workspace path yields a different CMAKE_PREFIX_PATH than the cached CMAKE_PREFIX_PATH
- The build space or devel space was built with a different tool such as catkin_make or catkin_make_isolated
- The build space or devel space was built in a different isolation mode

8.2 Dependency Resolution

8.2.1 Packages Are Being Built Out of Order

- The package.xml dependency tags are most likely incorrect. Note that dependencies are only used to order the packages, and there is no warning if a package can’t be found.
8.2.2 Incorrect Resolution of Workspace Overlays

It’s possible for a CMake package to include header directories as `SYSTEM` includes pointing to the workspace root include directory (like `/path/to/ws/devel/include`). If this happens, CMake will ignore any “normal” includes to that path, and prefer the `SYSTEM` include. This means that `/path/to/ws/devel/include` will be searched *after* any other normal includes. If another package specifies `/opt/ros/indigo/include` as a normal include, it will take precedence.

- Minimal example here: https://github.com/jbohren/isystem
- Overview of GCC’s system include precedence here: https://gcc.gnu.org/onlinedocs/cpp/System-Headers.html

As a workaround, you can force CMake to ignore all specified root include directories, and rely on `CPATH` for header resolution in these paths:

```
catkin config -a --cmake-args -DCMAKE_CXX_IMPLICIT_INCLUDE_DIRECTORIES="/opt/ros/indigo/include"
```

This is actually a bug in CMake and has been reported here: https://cmake.org/Bug/view.php?id=15970

8.3 Migration Problems

For troubleshooting problems when migrating from `catkin_make` or `catkin_make_isolated`, see *Migration Troubleshooting*. 
The `build` verb is used to build one or more packages in a catkin workspace. Like most verbs, `build` is context-aware and can be executed from within any directory contained by an initialized workspace. If a workspace is not yet initialized, `build` can initialize it with the default configuration, but only if it is called from the workspace root. Specific workspaces can also be built from arbitrary working directories with the `--workspace` option.

**Note:** To set up a workspace and clone the repositories used in the following examples, you can use `rosinstall_generator` and `wstool`. The following clones all of the ROS packages necessary for building the introductory ROS tutorials:

```bash
export ROS_DISTRO=indigo  # Set ROS distribution
mkdir -p /tmp/ros_tutorials_ws/src  # Create workspace
cd /tmp/ros_tutorials_ws/src  # Navigate to source space
rosinstall_generator --deps ros_tutorials > .rosinstall  # Get list of packages
wstool update  # Checkout all packages
cd /tmp/ros_tutorials_ws  # Navigate to ros workspace
   --root  # Initialize workspace

9.1 Basic Usage

9.1.1 Previewing The Build

Before actually building anything in the workspace, it is useful to preview which packages will be built and in what order. This can be done with the `--dry-run` option:

```bash
cd /tmp/ros_tutorials_ws  # Navigate to workspace
catkin build --dry-run  # Show the package build order
```

In addition to the listing the package names and in which order they would be built, it also displays the build type of each package.
9.1.2 Building a Workspace

When no packages are given as arguments, `catkin build` builds the entire workspace. It automatically creates directories for a **build space** and a **devel space**:

```
cd /tmp/ros_tutorials_ws  # Navigate to workspace
catkin build             # Build all the packages in the workspace
ls build                 # Show the resulting build space
ls devel                 # Show the resulting devel space
```

After the build finishes, the **build space** contains directories containing the intermediate build products for each package, and the **devel space** contains an FHS layout into which all the final build products are written.

**Note:** The products of `catkin build` differ significantly from the behavior of `catkin_make`, for example, which would have all of the build files and intermediate build products in a combined **build space** or `catkin_make_isolated` which would have an isolated FHS directory for each package in the **devel space**.

9.1.3 Status Line

When running `catkin build` with default options, it displays a “live” status line similar to the following:

```
[build - 20.2] [18/34 complete] [4/4 jobs] [1 queued] [xmlrpcpp:make (66%) - 4.9] ...
```

The status line stays at the bottom of the screen and displays the continuously-updated progress of the entire build as well as the active build jobs which are still running. It is composed of the following information:

- **[build - <T>]** – The first block on the left indicates the total elapsed build time `<T>` in seconds thus far.
- **[<M>/<N> complete]** – The second block from the left indicates the build progress in terms of the number of completed packages, `<M>` out of the total number of packages to be built `<N>`.
- **[<M>/<N> jobs]** – The third block from the left indicates the number of active total low-level jobs `<M>` out of the total number of low-level workers `<N>`.
- **[<N> queued]** – The fourth block from the left indicates the number of jobs `<N>` whose dependencies have already been satisfied and are ready to be built.
- **[<N> failed]** – The fifth block from the left indicates the number of jobs `<N>` which have failed. This block only appears once one or more jobs has failed.
- **[<package>:<stage> (<P>%) - <T>]** – The remaining blocks show details on the active jobs. These include the percent complete, `<P>`, of the stage, if available, as well as the time elapsed building the package, `<T>`.

When necessary, the status line can be disabled by passing the **--no-status** option to `catkin build`. This is sometimes required when running `catkin build` from within a program that doesn’t support the ASCII escape sequences required to reset and re-write the status line.

9.1.4 Console Messages

Normally, unless an error occurs, the output from each package’s build process is collected but not printed to the console. All that is printed is a pair of messages designating the start and end of a package’s build. This is formatted like the following for the `genmsg` package:
... Starting >>> {JOB} ...
Finished <<< {JOB} [ {TIME} seconds ] ...

Error messages are printed whenever a build job writes to stderr. In such cases, the build verb will automatically print the captured stderr buffer under a Warnings header once the job has completed, similarly to below:

```
____________________________________________________________________________
Warnings << {JOB}:{STAGE} {LOGFILE PATH}
  {WARNINGS}
  {REPRODUCTION COMMAND}
............................................................................
Finished << {JOB} [ {TIME} seconds ]
```

Note that the first line displays the path to the interleaved log file, which persists until the build space is cleaned. Additionally, if a package fails, the output to stderr is printed under the Errors header.

```
____________________________________________________________________________
Errors << {JOB}:{STAGE} {LOGFILE PATH}
  {ERRORS}
  {REPRODUCTION COMMAND}
............................................................................
Failed << {JOB}:{STAGE} [ Exited with code {EXIT CODE} ]
Failed << {JOB} [ {TIME} seconds ]
```

All of the messages from the underlying jobs can be shown when using the -v or --verbose option. This will print the normal messages when a build job starts and finishes as well as the interleaved output to stdout and stderr from each build command in a block.

All output can be printed interleaved with the --interleave-output option. In this case, each line is prefixed with the job and stage from which it came.

9.1.5 Build Summary

At the end of each build, a brief build summary is printed to guarantee that anomalies aren’t missed. This summary displays the total run-time, the number of successful jobs, the number of jobs which produced warnings, and the number of jobs which weren’t attempted due to failed dependencies.

```
[build] Runtime: 1.9 seconds total.
[build] Summary: 4 of 7 jobs completed.
[build] Warnings: None.
[build] Abandoned: 1 jobs were abandoned.
```

A more detailed summary can also be printed with the --summarize command, which lists the result for each package in the workspace.

9.2 Building Subsets of Packages

Consider a Catkin workspace with a source space populated with the following Catkin packages which have yet to be built:
### 9.2.1 Building Specific Packages

Specific packages can also be built by specifying them as positional arguments after the `build` verb:

```bash
cd /tmp/ros_tutorials_ws  # Navigate to workspace
catkin build roslib      # Build roslib and its dependencies
```

As shown above, only 4 packages (roslib and its dependencies), of the total 36 packages would be built.

### 9.2.2 Context-Aware Building

In addition to building all packages or specified packages with various dependency requirements, `catkin build` can also determine the package containing the current working directory. This is equivalent to specifying the name of the package on the command line, and is done by passing the `--this` option to `catkin build` like the following:

```bash
cd /tmp/ros_tutorials_ws  # Navigate to workspace
cd src/ros/roslib         # Navigate to roslib source directory
lss                      # Show source directory contents
catkin build --this      # Build roslib and its dependencies
```

### 9.2.3 Skipping Packages

Suppose you built every package up to roslib, but that package had a build error. After fixing the error, you could run the same build command again, but the `build` verb provides an option to save time in this situation. If re-started from the beginning, none of the products of the dependencies of roslib would be re-built, but it would still take some time for the underlying build system to verify that for each package.

Those checks could be skipped, however, by jumping directly to a given package. You could use the `--start-with` option to continue the build where you left off after fixing the problem.

```bash
cd /tmp/ros_tutorials_ws  # Navigate to workspace
catkin build --start-with roslib # Build roslib and its dependents
```

**Note:** `catkin build` will assume that all dependencies leading up to the package specified with the `--start-with` option have already been successfully built.

### 9.2.4 Building Single Packages

If you’re only interested in building a single package in a workspace, you can also use the `--no-deps` option along with a package name. This will skip all of the package’s dependencies, build the given package, and then exit.
9.3 Advanced Options

9.3.1 Temporarily Changing Build Flags

While the build configuration flags are set and stored in the build context, it’s possible to temporarily override or augment them when using the `build` verb.

```bash
$ catkin build --cmake-args -DCMAKE_C_FLAGS="-Wall -W -Wno-unused-parameter"
```

9.3.2 Building With Warnings

It can sometimes be useful to compile with additional warnings enabled across your whole catkin workspace. To achieve this, use a command similar to this:

```bash
$ catkin build -v --cmake-args -DCMAKE_C_FLAGS="-Wall -W -Wno-unused-parameter"
```

This command passes the `-DCMAKE_C_FLAGS=`... argument to all invocations of `cmake`.

9.3.3 Configuring Build Jobs

By default `catkin build` on a computer with `N` cores will build up to `N` packages in parallel and will distribute `N` make jobs among them using an internal job server. If your platform doesn’t support job server scheduling, `catkin build` will pass `-jN -lN` to `make` for each package.

You can control the maximum number of packages allowed to build in parallel by using the `-p` or `--parallel-packages` option and you can change the number of `make` jobs available with the `-j` or `--jobs` option.

By default, these jobs options aren’t passed to the underlying `make` command. To disable the job server, you can use the `--no-jobserver` option, and you can pass flags directly to `make` with the `--make-args` option.

**Note:** Jobs flags (`-jN` and/or `-lN`) can be passed directly to `make` by giving them to `catkin build`, but other `make` arguments need to be passed to the `--make-args` option.

9.3.4 Configuring Memory Use

In addition to CPU and load limits, `catkin build` can also limit the number of running jobs based on the available memory, using the hidden `--mem-limit` flag. This flag requires installing the Python `psutil` module and is useful on systems without swap partitions or other situations where memory use needs to be limited.

Memory is specified either by percent or by the number of bytes.

For example, to specify that `catkin build` should not start additional parallel jobs when 50% of the available memory is used, you could run:
$ catkin build --mem-limit 50%

Alternatively, if it should not start additional jobs when over 4GB of memory is used, you can specify:

$ catkin build --mem-limit 4G

### 9.4 Full Command-Line Interface

Usage: catkin build [-h] [--workspace WORKSPACE] [--profile PROFILE]

   [-dry-run] [-get-env PKNAME] [-this] [-no-deps]
   [-unbuilt] [-start-with PKNAME | --start-with-this]
   [--continue-on-failure] [--force-cmake] [--pre-clean]
   [--no-install-lock] [--save-config] [-j JOBS]
   [-p PACKAGE_JOBS] [-l LOAD_AVERAGE]
   [--jobserver | --no-jobserver]
   [--env-cache | --no-env-cache] [--cmake-args ARG [ARG ...]
   | --no-cmake-args] [--make-args ARG [ARG ...] |
   --no-make-args] [--catkin-make-args ARG [ARG ...] |
   --no-catkin-make-args] [--verbose] [--interleave-output]
   [--no-status] [--summarize] [--no-summarize]
   [--override-build-tool-check]
   [--limit-status-rate LIMIT_STATUS_RATE] [--no-notify]
   [PKNAME ...]

Build one or more packages in a catkin workspace. This invokes ‘CMake’, ‘make’, and optionally ‘make install’ for either all or the specified packages in a catkin workspace. Arguments passed to this verb can temporarily override persistent options stored in the catkin profile config. If you want to save these options, use the --save-config argument. To see the current config, use the ‘catkin config’ command.

Optional arguments:
- h, --help show this help message and exit
--workspace WORKSPACE, -w WORKSPACE
   The path to the catkin_tools workspace or a directory contained within it (default: ".")
--profile PROFILE
   The name of a config profile to use (default: active profile)
--dry-run, -n
   List the packages which will be built with the given arguments without building them.
--get-env PKNAME
   Print the environment in which PKNAME is built to stdout.

Packages:
   Control which packages get built.

PKNAME
   Workspace packages to build, package dependencies are built as well unless --no-deps is used. If no packages are given, then all the packages are built.
--this
   Build the package containing the current working directory.
--no-deps
   Only build specified packages, not their dependencies.
--unbuilt
   Build packages which have yet to be built.
--start-with PKNAME
   Build a given package and those which depend on it, skipping any before it.

(continues on next page)
**--start-with-this**
Similar to `--start-with`, starting with the package containing the current directory.

**--continue-on-failure, -c**
Try to continue building packages whose dependencies built successfully even if some other requested packages fail to build.

### Build:
Control the build behavior.

**--force-cmake**
Runs cmake explicitly for each catkin package.

**--pre-clean**
Runs 'make clean' before building each package.

**--no-install-lock**
Prevents serialization of the install steps, which is on by default to prevent file install collisions.

### Config:
Parameters for the underlying build system.

**--save-config**
Save any configuration options in this section for the next build invocation.

**-j JOBS, --jobs JOBS**
Maximum number of build jobs to be distributed across active packages. (default is cpu count)

**-p PACKAGE_JOBS, --parallel-packages PACKAGE_JOBS**
Maximum number of packages allowed to be built in parallel (default is cpu count)

**-l LOAD_AVERAGE, --load-average LOAD_AVERAGE**
Maximum load average before no new build jobs are scheduled.

**--jobserver**
Use the internal GNU Make job server which will limit the number of Make jobs across all active packages.

**--no-jobserver**
Disable the internal GNU Make job server, and use an external one (like distcc, for example).

**--env-cache**
Re-use cached environment variables when re-sourcing a resultspace that has been loaded at a different stage in the task.

**--no-env-cache**
Don't cache environment variables when re-sourcing the same resultspace.

**--cmake-args ARG [ARG ...]**
Arbitrary arguments which are passed to CMake. It collects all of following arguments until a "--" is read.

**--no-cmake-args**
Pass no additional arguments to CMake.

**--make-args ARG [ARG ...]**
Arbitrary arguments which are passed to make. It collects all of following arguments until a "--" is read.

**--no-make-args**
Pass no additional arguments to make (does not affect --catkin-make-args).

**--catkin-make-args ARG [ARG ...]**
Arbitrary arguments which are passed to make but only for catkin packages. It collects all of following arguments until a "--" is read.

**--no-catkin-make-args**
Pass no additional arguments to make for catkin packages (does not affect --make-args).
The behavior of the command-line interface.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--verbose, -v</td>
<td>Print output from commands in ordered blocks once the command finishes.</td>
</tr>
<tr>
<td>--interleave-output, -i</td>
<td>Prevents ordering of command output when multiple commands are running at the same time.</td>
</tr>
<tr>
<td>--no-status</td>
<td>Suppresses status line, useful in situations where carriage return is not properly supported.</td>
</tr>
<tr>
<td>--summarize, --summary, -s</td>
<td>Adds a build summary to the end of a build; defaults to on with --continue-on-failure, off otherwise</td>
</tr>
<tr>
<td>--no-summarize, --no-summary</td>
<td>Explicitly disable the end of build summary</td>
</tr>
<tr>
<td>--override-build-tool-check</td>
<td>Use to override failure due to using different build tools on the same workspace.</td>
</tr>
<tr>
<td>--limit-status-rate LIMIT_STATUS_RATE, --status-rate LIMIT_STATUS_RATE</td>
<td>Limit the update rate of the status bar to this frequency. Zero means unlimited. Must be positive, default is 10 Hz.</td>
</tr>
<tr>
<td>--no-notify</td>
<td>Suppresses system pop-up notification.</td>
</tr>
</tbody>
</table>
10.1 Space Cleaning

For any configuration, any of the active profile’s spaces can be cleaned entirely. This includes any of the top-level directories which are configured for a given profile. See the full command line interface for specifying specific spaces to clean.

To clean all of the spaces for a given profile, you can call the `catkin clean` verb without arguments:

```
catkin clean
```

When running this command, `catkin` will prompt you to confirm that you want to delete the entire directories:

```
$ catkin clean
[clean] Warning: This will completely remove the following directories. (Use `--yes` or `-y` to skip this check)
[clean] Log Space: /tmp/quickstart_ws/logs
[clean] Build Space: /tmp/quickstart_ws/build
[clean] Devel Space: /tmp/quickstart_ws/devel
[clean] Are you sure you want to completely remove the directories listed above? [yN]:
```

If you want to skip this check, you can use the `--yes` or `-y` options:

```
$ catkin clean -y
[clean] Removing develspace: /tmp/quickstart_ws/devel
```

(continues on next page)
[clean] Removing buildspace: /tmp/quickstart_ws/build
[clean] Removing log space: /tmp/quickstart_ws/logs

Note: The `clean` verb will also ask for additional confirmation if any of the directories to be removed are outside of your workspace root. To skip this additional check, you can use the `--force` option.

10.2 Partial Cleaning

If a workspace is built with a linked `devel space`, the `clean` verb can be used to clean the products from individual packages. This is possible since the `catkin` program will symbolically link the build products into the `devel space`, and stores a list of these links.

10.2.1 Cleaning a Single Package

Cleaning a single package (or several packages) is as simple as naming them:

```
catkin clean PKGNAME
```

This will remove products from this package from the `devel space`, and remove its `build space`.

10.2.2 Cleaning Products from Missing Packages

Sometimes, you may disable or remove source packages from your workspace’s `source space`. After packages have been removed from your `source space`, you can automatically clean the “orphaned” products with the following command:

```
catkin clean --orphans
```

10.2.3 Cleaning Dependent Packages

When cleaning one package, it’s sometimes useful to also clean all of the packages which depend on it. This can prevent leftover elements from affecting the dependents. To clean a package and only the packages which depend on it, you can run the following:

```
catkin clean --dependents PKGNAME
```

10.3 Cleaning Products from All Profiles

By default, the `clean` operating is applied only to the active or specified profile. To apply it to all profiles, use the `--all-profiles` option.
10.4 Cleaning Everything

If you want to clean everything except the source space (i.e. all files and folders generated by the `catkin` command, you can use `--deinit` to “deinitialize” the workspace. This will clean all products from all packages for all profiles, as well as the profile metadata, itself. After running this, a `catkin_tools` workspace will need to be reinitialized to be used.

```
catkin clean --deinit
```

10.5 Full Command-Line Interface

```
usage: catkin clean [-h] [--workspace WORKSPACE] [--profile PROFILE]
                  [--orphans] [--setup-files]
                  [PKGNAME ...]
```

Deletes various products of the build verb.

optional arguments:

- `--workspace WORKSPACE, -w WORKSPACE`
  The path to the catkin_tools workspace or a directory contained within it (default: ".")

- `--profile PROFILE`
  The name of a config profile to use (default: active profile)

- `--dry-run, -n`
  Show the effects of the clean action without modifying the workspace.

- `--verbose, -v`
  Verbose status output.

- `--yes, -y`
  Assume "yes" to all interactive checks.

- `--force, -f`
  Allow cleaning files outside of the workspace root.

- `--all-profiles`
  Apply the specified clean operation for all profiles in this workspace.

Full:

Remove everything except the source space.

- `--deinit`
  De-initialize the workspace, delete all build profiles and configuration. This will also clean subdirectories for all profiles in the workspace.

Spaces:

Clean workspace subdirectories for the selected profile.

- `--build, --build-space`
  Remove the entire build space.

- `--devel, --devel-space`
  Remove the entire devel space.

- `--install, --install-space`
  Remove the entire install space.

- `--logs, --log-space`
  Remove the entire log space.

Packages:

(continues on next page)
Clean products from specific packages in the workspace. Note that these options are only available in a `linked` devel space layout. These options will also automatically enable the `--force-cmake` option for the next build invocation.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKGNAME</td>
<td>Explicitly specify a list of specific packages to clean from the build, devel, and install space.</td>
</tr>
<tr>
<td>--this</td>
<td>Clean the package containing the current working directory from the build, devel, and install space.</td>
</tr>
<tr>
<td>--dependents, --deps</td>
<td>Clean the packages which depend on the packages to be cleaned.</td>
</tr>
<tr>
<td>--orphans</td>
<td>Remove products from packages are no longer in the source space. Note that this also removes packages which are blacklisted or which contain <code>CATKIN_IGNORE</code> marker files.</td>
</tr>
</tbody>
</table>

Advanced:

Clean other specific parts of the workspace.

--setup-files Clear the catkin-generated setup files from the devel and install spaces.
The `config` verb can be used to both view and manipulate a workspace’s configuration options. These options include all of the elements listed in the configuration summary.

By default, the `config` verb gets and sets options for a workspace’s *active* profile. If no profiles have been specified for a workspace, this is a default profile named `default`.

**Note:** Calling `catkin config` on an uninitialized workspace will not automatically initialize it unless it is used with the `--init` option.

### 11.1 Viewing the Configuration Summary

Once a workspace has been initialized, the configuration summary can be displayed by calling `catkin config` without arguments from anywhere under the root of the workspace. Doing so will not modify your workspace. The `catkin` command is context-sensitive, so it will determine which workspace contains the current working directory.

### 11.2 Appending or Removing List-Type Arguments

Several configuration options are actually *lists* of values. Normally for these options, the given values will replace the current values in the configuration.

If you would only like to modify, but not replace the value of a list-type option, you can use the `-a/--append-args` and `-r/--remove-args` options to append or remove elements from these lists, respectively.

List-type options include:

- `--cmake-args`
- `--make-args`
- `--catkin-make-args`
• --whitelist
• --blacklist

11.3 Installing Packages

Without any additional arguments, packages are not “installed” using the standard CMake install() targets. Addition of the --install option will configure a workspace so that it creates an install space and write the products of all install targets to that FHS tree. The contents of the install space, which, by default, is located in a directory named install will look like the following:

```
$ ls ./install
_lib_util.py bin env.sh etc include
lib setup.bash setup.sh setup.zsh share
```

11.4 Explicitly Specifying Workspace Chaining

Normally, a catkin workspace automatically “extends” the other workspaces that have previously been sourced in your environment. Each time you source a catkin setup file from a result-space (devel-space or install-space), it sets the $CMAKE_PREFIX_PATH in your environment, and this is used to build the next workspace. This is also sometimes referred to as “workspace chaining” and sometimes the extended workspace is referred to as a “parent” workspace.

With catkin config, you can explicitly set the workspace you want to extend, using the --extend argument. This is equivalent to sourcing a setup file, building, and then reverting to the environment before sourcing the setup file. For example, regardless of your current environment variable settings (like $CMAKE_PREFIX_PATH), using --extend can build your workspace against the /opt/ros/indigo install space.

Note that in case the desired parent workspace is different from one already being used, using the --extend argument also necessitates cleaning your workspace with catkin clean.

If you start with an empty $CMAKE_PREFIX_PATH, the configuration summary will show that you’re not extending any other workspace, as shown below:

```
$ echo $CMAKE_PREFIX_PATH
$ mkdir -p /tmp/path/to/my_catkin_ws/src
$ cd /tmp/path/to/my_catkin_ws
$ catkin init
```

```
Profile: default
Extending: None
Workspace: /tmp/path/to/my_catkin_ws
-----------------------------------------------
Source Space: [exists] /tmp/path/to/my_catkin_ws/src
Log Space: [exists] /tmp/path/to/my_catkin_ws/logs
Build Space: [exists] /tmp/path/to/my_catkin_ws/build
Devel Space: [exists] /tmp/path/to/my_catkin_ws/devel
Install Space: [unused] /tmp/path/to/my_catkin_ws/install
DESTDIR: [unused] None
-----------------------------------------------
Devel Space Layout: linked
Install Space Layout: None
-----------------------------------------------
... (continues on next page)
```
Initialized new catkin workspace in `/tmp/path/to/my_catkin_ws`

WARNING: Your workspace is not extending any other result space, but it is set to use a 'linked' devel space layout. This requires the 'catkin' CMake package in your source space in order to be built.

At this point you have a workspace which doesn’t extend anything. With the default devel space layout, this won’t build without the catkin CMake package, since this package is used to generate setup files.

If you realize this after the fact, you still can explicitly tell catkin build to extend some result space. Suppose you wanted to extend a standard ROS system install like /opt/ros/indigo. This can be done with the --extend option like so:

```
$ catkin clean
$ catkin config --extend /opt/ros/indigo
```

Profile: default
Extending: [explicit] /opt/ros/indigo
Workspace: /tmp/path/to/my_catkin_ws

Source Space: [exists] /tmp/path/to/my_catkin_ws/src
Log Space: [missing] /tmp/path/to/my_catkin_ws/logs
Build Space: [missing] /tmp/path/to/my_catkin_ws/build
Devel Space: [missing] /tmp/path/to/my_catkin_ws/devel
Install Space: [unused] /tmp/path/to/my_catkin_ws/install
DESTDIR: [unused] None

Devel Space Layout: linked
Install Space Layout: None

Workspace configuration appears valid.

```
$ catkin build
...
```

```
$ source devel/setup.bash
$ echo $CMAKE_PREFIX_PATH
/tmp/path/to/my_catkin_ws:/opt/ros/indigo
```

## 11.5 Whitelisting and Blacklisting Packages

Packages can be added to a package whitelist or blacklist in order to change which packages get built. If the whitelist is non-empty, then a call to catkin build with no specific package names will only build the packages on the whitelist. This means that you can still build packages not on the whitelist, but only if they are named explicitly or are dependencies of other whitelisted packages.

To set the whitelist, you can call the following command:

```
$ catkin build
```
catkin config --whitelist foo bar

To clear the whitelist, you can use the --no-whitelist option:

```
catkin config --no-whitelist
```

If the blacklist is non-empty, it will filter the packages to be built in all cases except where a given package is named explicitly. This means that blacklisted packages will not be built even if another package in the workspace depends on them.

**Note:** Blacklisting a package does not remove it’s build directory or build products, it only prevents it from being rebuilt.

To set the blacklist, you can call the following command:

```
catkin config --blacklist baz
```

To clear the blacklist, you can use the --no-blacklist option:

```
catkin config --no-blacklist
```

Note that you can still build packages on the blacklist and whitelist by passing their names to `catkin build` explicitly.

### 11.6 Accelerated Building with Environment Caching

Each package is built in a special environment which is loaded from the current workspace and any workspaces that the current workspace is extending. If you are confident that your workspace’s environment is not changing during a build, you can tell `catkin build` to cache these environments with the --env-cache option. This has the effect of dramatically reducing build times for workspaces where many packages are already built.

### 11.7 Full Command-Line Interface

```
usage: catkin config [-h] [--workspace WORKSPACE] [--profile PROFILE]
                   [--append-args | --remove-args] [--init]
                   [--extend EXTEND_PATH | --no-extend] [--mkdirs]
                   [--authors NAME [EMAIL ...] | --maintainers NAME
                   [EMAIL ...] | --licenses LICENSE [LICENSE ...]]
                   [--whitelist PKG [PKG ...] | --no-whitelist]
                   [--blacklist PKG [PKG ...] | --no-blacklist]
                   [--build-space BUILD_SPACE | --default-build-space]
                   [--devel-space DEVEL_SPACE | --default-devel-space]
                   [--install-space INSTALL_SPACE | --default-install-space]
                   [--log-space LOG_SPACE | --default-log-space]
                   [--source-space SOURCE_SPACE | --default-source-space]
                   [-x SPACE_SUFFIX]
                   [--link-devel | --merge-devel | --isolate-devel]
                   [--install | --no-install]
                   [--isolate-install | --merge-install] [-j JOBS]
                   [-p PACKAGE_JOBS] [-l LOAD_AVERAGE]
```

(continues on next page)
optional arguments:
  -h, --help     show this help message and exit
  --workspace WORKSPACE, -w WORKSPACE
                  The path to the catkin_tools workspace or a directory
                  contained within it (default: ".")
  --profile PROFILE
                  The name of a config profile to use (default: active
                  profile)

Behavior:
  Options affecting argument handling.
  --append-args, -a For list-type arguments, append elements.
  --remove-args, -r For list-type arguments, remove elements.

Workspace Context:
  Options affecting the context of the workspace.
  --init          Initialize a workspace if it does not yet exist.
  --extend EXTEND_PATH, -e EXTEND_PATH
                  Explicitly extend the result-space of another catkin
                  workspace, overriding the value of $CMAKE_PREFIX_PATH.
  --no-extend     Un-set the explicit extension of another workspace as
                  set by --extend.
  --mkdirs        Create directories required by the configuration (e.g.
                  source space) if they do not already exist.

Package Create Defaults:
  Information of default authors/maintainers of created packages
  --authors NAME [EMAIL ...]
                  Set the default authors of created packages
  --maintainers NAME [EMAIL ...]
                  Set the default maintainers of created packages
  --licenses LICENSE [LICENSE ...]
                  Set the default licenses of created packages

Package Build Defaults:
  Packages to include or exclude from default build behavior.
  --whitelist PKG [PKG ...]
                  Set the packages on the whitelist. If the whitelist is
                  non-empty, only the packages on the whitelist are
                  built with a bare call to `catkin build`.
  --no-whitelist  Clear all packages from the whitelist.
  --blacklist PKG [PKG ...]
Set the packages on the blacklist. Packages on the blacklist are not built with a bare call to `catkin build`.

--no-blacklist Clear all packages from the blacklist.

Spaces:
Location of parts of the catkin workspace.

--build-space BUILD_SPACE, -b BUILD_SPACE
The path to the build space.

--default-build-space
Use the default path to the build space ("build")

--devel-space DEVEL_SPACE, -d DEVEL_SPACE
The path to the devel space.

--default-devel-space
Use the default path to the devel space ("devel")

--install-space INSTALL_SPACE, -i INSTALL_SPACE
The path to the install space.

--default-install-space
Use the default path to the install space ("install")

--log-space LOG_SPACE, -L LOG_SPACE
The path to the log space.

--default-log-space
Use the default path to the log space ("logs")

--source-space SOURCE_SPACE, -s SOURCE_SPACE
The path to the source space.

--default-source-space
Use the default path to the source space ("src")

-x SPACE_SUFFIX, --space-suffix SPACE_SUFFIX
Suffix for build, devel, and install space if they are not otherwise explicitly set.

Devel Space:
Options for configuring the structure of the devel space.

--link-devel Build products from each catkin package into isolated spaces, then symbolically link them into a merged devel space.

--merge-devel Build products from each catkin package into a single merged devel spaces.

--isolate-devel Build products from each catkin package into isolated devel spaces.

Install Space:
Options for configuring the structure of the install space.

--install Causes each package to be installed to the install space.

--no-install Disables installing each package into the install space.

--isolate-install Install each catkin package into a separate install space.

--merge-install Install each catkin package into a single merged install space.

Build Options:
Options for configuring the way packages are built.
-j JOBS, --jobs JOBS  Maximum number of build jobs to be distributed across active packages. (default is cpu count)

-p PACKAGE_JOBS, --parallel-packages PACKAGE_JOBS  
Maximum number of packages allowed to be built in parallel (default is cpu count)

-1 LOAD_AVERAGE, --load-average LOAD_AVERAGE  
Maximum load average before no new build jobs are scheduled

--jobserver  Use the internal GNU Make job server which will limit the number of Make jobs across all active packages.

--no-jobserver  Disable the internal GNU Make job server, and use an external one (like distcc, for example).

--env-cache  Re-use cached environment variables when re-sourcing a resultspace that has been loaded at a different stage in the task.

--no-env-cache  Don't cache environment variables when re-sourcing the same resultspace.

--cmake-args ARG [ARG ...]  
Arbitrary arguments which are passed to CMake. It collects all of following arguments until a "--" is read.

--no-cmake-args  Pass no additional arguments to CMake.

--make-args ARG [ARG ...]  
Arbitrary arguments which are passed to make. It collects all of following arguments until a "--" is read.

--no-make-args  Pass no additional arguments to make (does not affect --catkin-make-args).

--catkin-make-args ARG [ARG ...]  
Arbitrary arguments which are passed to make but only for catkin packages. It collects all of following arguments until a "--" is read.

--no-catkin-make-args  Pass no additional arguments to make for catkin packages (does not affect --make-args).
This verb enables you to quickly create workspace elements like boilerplate Catkin packages.

### 12.1 Full Command-Line Interface

**usage:** catkin create [-h] [--workspace WORKSPACE] [--profile PROFILE] (pkg) ...

Creates catkin workspace resources like packages.

**positional arguments:**
- **{pkg}** sub-command help
  - pkg Create a new catkin package.

**optional arguments:**
- **-h, --help** show this help message and exit
- **--workspace WORKSPACE, -w WORKSPACE** The path to the catkin_tools workspace or a directory contained within it (default: ".")
- **--profile PROFILE** The name of a config profile to use (default: active profile)

### 12.1.1 catkin create pkg

**usage:** catkin create pkg [-h] [-p PATH] --rosdistro ROSDISTRO
- **-v MAJOR.MINOR.PATCH** [-l LICENSE] [-m NAME EMAIL]
  - [-d DESCRIPTION]
  - [--catkin-deps [DEP ...]] [--system-deps [DEP ...]]
  - [--boost-components [COMP ...]]
  - PKG_NAME [PKG_NAME ...]

(continues on next page)
Create a new Catkin package. Note that while the default options used by this command are sufficient for prototyping and local usage, it is important that any publicly-available packages have a valid license and a valid maintainer e-mail address.

**positional arguments:**

- **PKG_NAME**
  The name of one or more packages to create. This name should be completely lower-case with individual words separated by underscores.

**optional arguments:**

- **-h, --help**
  show this help message and exit

- **-p PATH, --path PATH**
  The path into which the package should be generated.

- **--rosdistro ROSDISTRO**
  The ROS distro (default: environment variable ROS_DISTRO if defined)

**Package Metadata:**

- **-v MAJOR.MINOR.PATCH, --version MAJOR.MINOR.PATCH**
  Initial package version. (default 0.0.0)

- **-l LICENSE, --license LICENSE**
  The software license under which the code is distributed, such as BSD, MIT, GPLv3, or others. (default: "TODO")

- **-m NAME EMAIL, --maintainer NAME EMAIL**
  A maintainer who is responsible for the package. (default: [username, username@todo.todo]) (multiple allowed)

- **-a NAME EMAIL, --author NAME EMAIL**
  An author who contributed to the package. (default: no additional authors) (multiple allowed)

- **-d DESCRIPTION, --description DESCRIPTION**
  Description of the package. (default: empty)

**Package Dependencies:**

- **--catkin-deps [DEP ...], -c [DEP ...]**
  The names of one or more Catkin dependencies. These are Catkin-based packages which are either built as source or installed by your system's package manager.

- **--system-deps [DEP ...], -s [DEP ...]**
  The names of one or more system dependencies. These are other packages installed by your operating system's package manager.

**C++ Options:**

- **--boost-components [COMP ...]**
  One or more boost components used by the package.
The `env` verb can be used to both print the current environment variables and run a command in a modified environment. This verb is supplied as a cross-platform alternative to the UNIX `env` command or the `cmake -E environment` command. It is primarily used in the build stage command reproduction.

### 13.1 Full Command-Line Interface

```plaintext
usage: catkin env [-h] [-i] [-s] [NAME=VALUE ...] [COMMAND] [ARG ...]

Run an arbitrary command in a modified environment.

positional arguments:
  NAME=VALUE     Explicitly set environment variables for the subcommand. These override variables given to stdin.

optional arguments:
  -h, --help     show this help message and exit
  -i, --ignore-environment
                 Start with an empty environment.
  -s, --stdin    Read environment variable definitions from stdin. Variables should be given in NAME=VALUE format, separated by null-bytes.

command:
  COMMAND        Command to run. If omitted, the environment is printed to stdout.
  ARG            Arguments to the command.
```
catkin init – Initialize a Workspace

The `init` verb is the simplest way to “initialize” a catkin workspace so that it can be automatically detected automatically by other verbs which need to know the location of the workspace root.

This verb does not store any configuration information, but simply creates the hidden `.catkin_tools` directory in the specified workspace. If you want to initialize a workspace simultaneously with an initial config, see the `--init` option for the `config` verb.

Catkin workspaces can be initialized anywhere. The only constraint is that catkin workspaces cannot contain other catkin workspaces. If you call `catkin init` and it reports an error saying that the given directory is already contained in a workspace, you can call `catkin config` to determine the root of that workspace.

### 14.1 Full Command-Line Interface

```
usage: catkin init [-h] [--workspace WORKSPACE] [--reset]

Initializes a given folder as a catkin workspace.

optional arguments:
  -h, --help            show this help message and exit
  --workspace WORKSPACE, -w WORKSPACE
                        The path to the catkin_tools workspace or a directory
                        contained within it (default: ".")
  --reset               Reset (delete) all of the metadata for the given
                        workspace.
```

63
The list verb for the catkin command is used to find and list information about catkin packages. By default, it will list the packages in the workspace containing the current working directory. It can also be used to list the packages in any other arbitrary directory.

15.1 Checking for Catkin Package Warnings

In addition to the names of the packages in your workspace, running catkin list will output any warnings about catkin packages in your workspace. To suppress these warnings, you can use the --quiet option.

15.2 Using Unformatted Output in Shell Scripts

catkin list --unformatted is useful for automating shell scripts in UNIX pipe-based programs.

15.3 Full Command-Line Interface

```
usage: catkin list [-h] [--workspace WORKSPACE] [--profile PROFILE]
                  [--deps | --rdeps] [--depends-on [PKG ...]]
                  [--rdepends-on [PKG ...]] [--this]
                  [--directory [DIRECTORY ...]] [--quiet] [--unformatted]

Lists catkin packages in the workspace or other arbitrary folders.
```

optional arguments:
- h, --help show this help message and exit
--workspace WORKSPACE, -w WORKSPACE
  The path to the catkin_tools workspace or a directory contained within it (default: ".")
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--profile PROFILE</code></td>
<td>The name of a config profile to use (default: active profile)</td>
</tr>
<tr>
<td>Information:</td>
<td>Control which information is shown.</td>
</tr>
<tr>
<td><code>--deps, --dependencies</code></td>
<td>Show direct dependencies of each package.</td>
</tr>
<tr>
<td><code>--rdeps, --recursive-dependencies</code></td>
<td>Show recursive dependencies of each package.</td>
</tr>
<tr>
<td>Packages:</td>
<td>Control which packages are listed.</td>
</tr>
<tr>
<td><code>--depends-on [PKG ...]</code></td>
<td>Only show packages that directly depend on specific package(s).</td>
</tr>
<tr>
<td><code>--rdepends-on [PKG ...], --recursive-depends-on [PKG ...]</code></td>
<td>Only show packages that recursively depend on specific package(s).</td>
</tr>
<tr>
<td><code>--this</code></td>
<td>Show the package which contains the current working directory.</td>
</tr>
<tr>
<td><code>--directory [DIRECTORY ...], -d [DIRECTORY ...]</code></td>
<td>Pass list of directories process all packages in directory</td>
</tr>
<tr>
<td>Interface:</td>
<td>The behavior of the command-line interface.</td>
</tr>
<tr>
<td><code>--quiet</code></td>
<td>Don't print out detected package warnings.</td>
</tr>
<tr>
<td><code>--unformattted, -u</code></td>
<td>Print list without punctuation and additional details.</td>
</tr>
</tbody>
</table>
The `locate` verb can be used to locate important locations in the workspace such as the active `source`, `build`, `devel`, and `install` spaces, and package directories in the workspace.

## 16.1 Full Command-Line Interface

```
usage: catkin locate [-h] [--workspace WORKSPACE] [--profile PROFILE] [-e]
                   [-r] [-q] [-b | -d | -i | -L | -s] [--this]
                   [--shell-verbs] [--examples] [PACKAGE]
```

Get the paths to various locations in a workspace.

### optional arguments:

- `-h`, `--help`  show this help message and exit
- `--workspace WORKSPACE`, `--w WORKSPACE`  The path to the catkin_tools workspace or a directory contained within it (default: ".")
- `--profile PROFILE`  The name of a config profile to use (default: active profile)

### Behavior:

- `-e`, `--existing-only`  Only print paths to existing directories.
- `-r`, `--relative`  Print relative paths instead of the absolute paths.
- `-q`, `--quiet`  Suppress warning output.

### Sub-Space Options:

Get the absolute path to one of the following locations in the given workspace with the given profile.

- `-b`, `--build`, `--build-space`  Get the path to the build space.
- `-d`, `--devel`, `--devel-space`
Get the path to the devel space.

   -i, --install, --install-space

   Get the path to the install space.

   -L, --logs, --log-space

   Get the path to the log space.

   -s, --src, --source-space

   Get the path to the source space.

Package Directories:
Get the absolute path to package directories in the given workspace and sub-space. By default this will output paths in the workspace's source space. If the -b (--build) flag is given, it will output the path to the package's build directory. If the -d or -i (--devel or --install) flags are given, it will output the path to the package's share directory in that space. If no package is provided, the base space paths are printed, e.g. `catkin locate -s` might return `/path/to/ws/src` and `catkin locate -s foo` might return `/path/to/ws/src/foo`.

<table>
<thead>
<tr>
<th>PACKAGE</th>
<th>The name of a package to locate.</th>
</tr>
</thead>
<tbody>
<tr>
<td>--this</td>
<td>Locate package containing current working directory.</td>
</tr>
</tbody>
</table>

Special Directories:
Get the absolute path to a special catkin location

   --shell-verbs    Get the path to the shell verbs script.

   --examples   Get the path to the examples directory.
Many verbs contain a --profile option, which selects which configuration profile to use, without which it will use the “active” profile. The profile verb enables you to manager the available profiles as well as set the “active” profile when using other verbs.

Even without using the profile verb, any use of the catkin command which changes the workspace is implicitly using a configuration profile called default.

The profile verb has several sub-commands for profile management. These include the following:

- list – List the available profiles
- set – Set the active profile by name.
- add – Add a new profile by name.
- rename – Rename a given profile.
- remove – Remove a profile by name.

17.1 Creating Profiles Automatically

After initializing a workspace, you can start querying information about profiles. Until you execute a verb which actually writes a profile configuration, however, there will be no profiles listed:

```
$ mkdir -p /tmp/path/to/my_catkin_ws/src
$ cd /tmp/path/to/my_catkin_ws
$ catkin init
$ catkin profile list
[profile] This workspace has no metadata profiles. Any configuration settings will automatically be applied to a new profile called `default`.
```

To see these effects, you can run catkin config to write a default configuration to the workspace:
The `profile` verb now shows that the profile named "default" is available and is active. Calling `catkin config` with the `--profile` argument will automatically create a profile based on the given configuration options:

```
$ catkin config --profile alternate -x _alt
```

Note that while the profile named `alternate` has been configured, it is still not `active`, so any calls to catkin-verbs without an explicit `--profile alternate` option will still use the profile named `default`.
17.2 Explicitly Creating Profiles

Profiles can also be added explicitly with the `add` command. This profile can be initialized with configuration information from either the default settings or another profile.

```
$ catkin profile list
[profile] Available profiles:
  - alternate
  - default (active)
$ catkin profile add alternate_2 --copy alternate
[profile] Created a new profile named alternate_2 based on profile alternate
[profile] Available profiles:
  - alternate
  - alternate_2
  - default (active)
```

17.3 Setting the Active Profile

The active profile can be easily set with the `set` sub-command. Suppose a workspace has the following profiles:

```
$ catkin profile list
[profile] Available profiles:
  - alternate
  - alternate_2
  - default (active)
$ catkin profile set alternate_2
[profile] Activated catkin metadata profile: alternate_2
[profile] Available profiles:
  - alternate
  - alternate_2 (active)
  - default
```

17.4 Renaming and Removing Profiles

The `profile` verb can also be used for renaming and removing profiles:

```
$ catkin profile list
[profile] Available profiles:
  - alternate
  - alternate_2 (active)
  - default
$ catkin profile rename alternate_2 alternate2
[profile] Renamed profile alternate_2 to alternate2
[profile] Available profiles:
  - alternate
  - alternate2 (active)
  - default
$ catkin profile remove alternate
[profile] Removed profile: alternate
[profile] Available profiles:
  - alternate2 (active)
  - default
```
17.5 Full Command-Line Interface

usage: catkin profile [-h] [--workspace WORKSPACE] {list,set,add,rename,remove} ...

Manage config profiles for a catkin workspace.

positional arguments:
    {list,set,add,rename,remove} sub-command help
    list    List the available profiles.
    set     Set the active profile by name.
    add     Add a new profile by name.
    rename  Rename a given profile.
    remove  Remove a profile by name.

optional arguments:
    -h, --help     show this help message and exit
    --workspace WORKSPACE, -w WORKSPACE
                  The path to the catkin workspace. Default: current
                  working directory

17.5.1 catkin profile list

usage: catkin profile list [-h] [--unformatted] [--active]

optional arguments:
    -h, --help     show this help message and exit
    --unformatted, -u Print profile list without punctuation and additional
details.
    --active      Print only active profile.

17.5.2 catkin profile set

usage: catkin profile set [-h] name

positional arguments:
    name    The profile to activate.

optional arguments:
    -h, --help     show this help message and exit

17.5.3 catkin profile add

usage: catkin profile add [-h] [-f] [--copy BASE_PROFILE | --copy-active | --extend PARENT_PROFILE] name

positional arguments:
    name    The new profile name.
optional arguments:
- `-h, --help` show this help message and exit
- `-f, --force` Overwrite an existing profile.
- `--copy BASE_PROFILE` Copy the settings from an existing profile. (default: None)
- `--copy-active` Copy the settings from the active profile.
- `--extend PARENT_PROFILE` Extend another profile

17.5.4 catkin profile rename

```
usage: catkin profile rename [-h] [-f] current_name new_name
```

positional arguments:
- `current_name` The current name of the profile to be renamed.
- `new_name` The new name for the profile.

optional arguments:
- `-h, --help` show this help message and exit
- `-f, --force` Overwrite an existing profile.

17.5.5 catkin profile remove

```
usage: catkin profile remove [-h] [name ...]
```

positional arguments:
- `name` One or more profile names to remove.

optional arguments:
- `-h, --help` show this help message and exit
CHAPTER 18

**catkin test – Test Packages**

The `test` verb is used to test one or more packages in a catkin workspace. Like most verbs, `test` is context-aware and can be executed from within any directory contained by an initialized workspace. Specific workspaces can also be built from arbitrary working directories with the `--workspace` option.

### 18.1 Basic Usage

Before running tests for packages in the workspace, they have to be built with `catkin build`. Then, to run the tests, use the following:

```
$ catkin test
```

Under the hood, this invokes the `make` targets `run_tests` or `test`, depending on the package. catkin packages all define the `run_tests` target which aggregates all types of tests and runs them together. For cmake packages that do not use catkin, the `test` target is invoked. This target is usually populated by cmake when the `enable_testing()` command is used in the `CMakeLists.txt`. If it does not exist, a warning is printed.

To run a catkin test for a specific catkin package, from a directory within that package:

```
$ catkin test --this
```

### 18.2 Advanced Options

To manually specify a different `make` target, use `--test-target`:

```
$ catkin test --test-target gtest
```

It is also possible to use `--catkin-test-target` to change the target only for catkin packages.

Normally, the tests are run in parallel, similar to the build jobs of `catkin build`. To avoid building packages in parallel or to reduce the amount of parallel jobs, use `-p`:
Sometimes, it can be helpful to see the output of tests while they are still running. This can be achieved using \texttt{--interleave-output}.

### 18.3 Full Command-Line Interface

```
usage: catkin test \[-h\] \[--workspace WORKSPACE\] \[--profile PROFILE\] \[--this\]
           \[--continue-on-failure\] \[-p PACKAGE_JOBS\] \[-t TARGET\]
           \[--catkin-test-target TARGET\] \[--make-args ARG ARG ...\]
           \[--verbose\] \[--interleave-output\] \[--summarize\]
           \[--no-status\] \[--limit-status-rate LIMIT_STATUS_RATE\]
           \[--no-notify\]
           [PKGNAME ...]
```

Test one or more packages in a catkin workspace. This invokes `make run_tests` or `make test` for either all or the specified packages in a catkin workspace.

**optional arguments:**

\texttt{-h, --help} show this help message and exit

\texttt{--workspace WORKSPACE, \-w WORKSPACE} The path to the catkin_tools workspace or a directory contained within it (default: ".")

\texttt{--profile PROFILE} The name of a config profile to use (default: active profile)

**Packages:**

Control which packages get tested.

\texttt{PKGNAME} Workspace packages to test. If no packages are given, then all the packages are tested.

\texttt{--this} Test the package containing the current working directory.

\texttt{--continue-on-failure, \-c} Continue testing packages even if the tests for other requested packages fail.

**Config:**

Parameters for the underlying build system.

\texttt{\-p PACKAGE_JOBS, --parallel-packages PACKAGE_JOBS} Maximum number of packages allowed to be built in parallel (default is cpu count)

\texttt{\-t TARGET, --test-target TARGET} Make target to run for tests (default is "run_tests" for catkin and "test" for cmake)

\texttt{--catkin-test-target TARGET} Make target to run for tests for catkin packages, overwrites \texttt{--test-target} (default is "run_tests")

\texttt{--make-args ARG ARG ...} Arbitrary arguments which are passed to make. It collects all of following arguments until a "--" is read.

**Interface:**

(continues on next page)
The behavior of the command-line interface.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--verbose, -v</td>
<td>Print output from commands in ordered blocks once the command finishes.</td>
</tr>
<tr>
<td>--interleave-output, -i</td>
<td>Prevents ordering of command output when multiple commands are running at the same time.</td>
</tr>
<tr>
<td>--summarize, --summary, -s</td>
<td>Adds a summary to the end of the log</td>
</tr>
<tr>
<td>--no-status</td>
<td>Suppresses status line, useful in situations where carriage return is not properly supported.</td>
</tr>
<tr>
<td>--limit-status-rate LIMIT_STATUS_RATE, --status-rate LIMIT_STATUS_RATE</td>
<td>Limit the update rate of the status bar to this frequency. Zero means unlimited. Must be positive, default is 10 Hz.</td>
</tr>
<tr>
<td>--no-notify</td>
<td>Suppresses system pop-up notification.</td>
</tr>
</tbody>
</table>
You can use the `locate` verb to locate the shell file for your installation. When you source the resulting file, you can use bash/zsh shell functions which provide added utility.

```
. `catkin locate --shell-verbs`
```

Provided verbs are:

- `catkin cd` – Change to package directory in source space.
- `catkin source` – Source the devel space or install space of the containing workspace.

## 19.1 Full Command-Line Interface

Change to package directory in source space with `cd` verb.

```
usage: catkin cd [ARGS...]
ARGS are any valid catkin locate arguments
```

The `source` verb sources the devel space or install space of the containing workspace.

```
usage: catkin source [-w /path/to/ws]
Sources setup.sh in the workspace.
optional arguments:
  -w [/path/to/ws] Source setup.sh from given workspace.
```
Verb Aliasing

The `catkin` command allows you to define your own verb “aliases” which expand to more complex expressions including built-in verbs, command-line options, and other verb aliases. These are processed before any other command-line processing takes place, and can be useful for making certain use patterns more convenient.

### 20.1 The Built-In Aliases

You can list the available aliases using the `--list-aliases` option to the `catkin` command. Below are the built-in aliases as displayed by this command:

```
$ catkin --list-aliases
b: build
bt: b --this
ls: list
install: config --install
```

### 20.2 Defining Additional Aliases

Verb aliases are defined in the `verb_aliases` sub-directory of the `catkin config` folder, `~/.config/catkin/verb_aliases`. Any YAML files in that folder (files with a `.yaml` extension) will be processed as definition files.

These files are formatted as simple YAML dictionaries which map aliases to expanded expressions, which must be composed of other `catkin` verbs, options, or aliases:

```
<ALIAS>: <EXPRESSION>
```

For example, aliases which configure a workspace profile so that it ignores the value of the `CMAKE_PREFIX_PATH` environment variable, and instead extends one or another ROS install spaces could be defined as follows:
After defining these aliases, one could use them with optional additional options and build a given configuration profile.

```bash
$ catkin extend-overlay
$ catkin profile set overlay
$ catkin build some_package
```

**Note:** The `catkin` command will initialize the `verb_aliases` directory with a file named `00-default-aliases.yaml` containing the set of built-in aliases. These defaults can be overridden by adding additional definition files, but the default alias file should not be modified since any changes to it will be over-written by invocations of the `catkin` command.

## 20.3 Alias Precedence and Overriding Aliases

Verb alias files in the `verb_aliases` directory are processed in alphabetical order, so files which start with larger numbers will override files with smaller numbers. In this way you can override the built-in aliases using a file which starts with a number higher than 00-.

For example, the `bt: build --this` alias exists in the default alias file, `00-default-aliases.yaml`, but you can create a file to override it with an alternate definition defined in a file named `01-my-aliases.yaml`.

```yaml
# ~/.config/catkin/verb_aliases/01-my-aliases.yaml
# Override `bt` to build with no deps
bt: build --this --no-deps
```

You can also disable or unset an alias by setting its value to `null`. For example, the `ls: list alias` is defined in the default aliases, but you can override it with this entry in a custom file named something like `02-unset.yaml`:

```yaml
# ~/.config/catkin/verb_aliases/02-unset.yaml
# Disable `ls` alias
ls: null
```

## 20.4 Recursive Alias Expansion

Additionally, verb aliases can be recursive, for instance in the `bt` alias, the `b` alias expands to `build` so that `b --this` expands to `build --this`. The `catkin` command shows the expansion of aliases when they are invoked so that their behavior is more transparent:

```bash
$ catkin bt
=> Expanding alias 'bt' from 'catkin bt' to 'catkin b --this'
=> Expanding alias 'b' from 'catkin b --this' to 'catkin build --this'
...
CHAPTER 21

Linked Devel Space

In addition to the merged and isolated devel space layouts provided by catkin_make and catkin_make_isolated, respectively, catkin_tools provides a default linked layout which enables robust cleaning of individual packages from a workspace. It does this by building each package into its own hidden FHS tree, and then symbolically linking all products into the unified devel space which is specified in the workspace configuration.

When building with a linked layout, Catkin packages are built into FHS trees stored in the .private hidden directory at the root of the devel space. Within this directory is a directory for each package in the workspace.

21.1 Setup File Generation

In the merged layout, every package writes and then over-writes the colliding setup files in the root of the devel space. This leads to race conditions and other problems when trying to parallelize building. With the linked layout, however, only one package generates these files, and this is either a built-in “prebuild” package, or if it exists in the workspace, the catkin CMake package, itself.

21.2 .catkin File Generation

When using the linked layout, catkin_tools is also responsible for managing the .catkin file in the root of the devel space.
One of the core modules in catkin_tools is the job executor. The executor performs jobs required to complete a task in a way that maximizes (or achieves a specific) resource utilization subject to job dependency constraints. The executor is closely integrated with logging and job output capture. This page details the design and implementation of the executor.

22.1 Execution Model

The execution model is fairly simple. The executor executes a single task for a given command (i.e. `build`, `clean`, etc.). A task is a set of jobs which are related by an acyclic dependency graph. Each job is given a unique identifier and is composed of a set of dependencies and a sequence of executable stages, which are arbitrary functions or subprocess calls which utilize one or more workers to be executed. The allocation of workers is managed by the job server. Throughout execution, synchronization with the user-facing interface and output formatting are mediated by a simple event queue.

The executor is single-threaded and uses an asynchronous loop to execute jobs as futures. If a job contains blocking stages it can utilize a normal thread pool for execution, but is still only guaranteed one worker by the main loop of the executor. See the following section for more information on workers and the job server.

The input to the executor is a list of topologically-sorted jobs with no circular dependencies and some parameters which control the job server behavior. These behavior parameters are explained in detail in the following section.

Each job is in one of the following life-cycle states at any time:

- **PENDING** Not ready to be executed (dependencies not yet completed)
- **QUEUED** Ready to be executed once workers are available
- **ACTIVE** Being executed by one or more workers
- **FINISHED** Has been executed and either succeeded or failed (terminal)
- **ABANDONED** Was not built because a prerequisite was not met (terminal)

All jobs begin in the PENDING state, and any jobs with unsatisfiable dependencies are immediately set to ABANDONED, and any jobs without dependencies are immediately set to QUEUED. After the state initialization, the
Fig. 1: **Executor Job Life-cycle**

executor processes jobs in a main loop until they are in one of the two terminal states (**FINISHED** or **ABANDONED**). Each main loop iteration does the following:

- While job server tokens are available, create futures for **QUEUED** jobs and make them **ACTIVE**.
- Report status of all jobs to the event queue.
- Retrieve **ACTIVE** job futures which have completed and set them **FINISHED**.
- Check for any **PENDING** jobs which need to be **ABANDONED** due to failed jobs.
- Change all **PENDING** jobs whose dependencies are satisfied to **QUEUED**.

Once each job is in one of terminal states, the executor pushes a final status event and returns.

### 22.2 Job Server Resource Model

As mentioned in the previous section, each task includes a set of jobs which are activated by the **job server**. In order to start a queued job, at least one worker needs to be available. Once a job is started, it is assigned a single worker from the job server. These are considered **top-level** jobs since they are managed directly by the catkin executor. The number of top-level jobs can be configured for a given task.

Additionally to top-level parallelism, some job stages are capable of running in parallel, themselves. In such cases, the job server can interface directly with the underlying stage’s low-level job allocation. This enables multi-level parallelism without allocating more than a fixed number of jobs.

Fig. 2: **Executor Job Flow and Resource Utilization** – In this snapshot of the job pipeline, the executor is executing four of six possible top-level jobs, each with three stages, and using seven of eight total workers. Two jobs are executing sub-processes, which have side-channel communication with the job server.

One such parallel-capable stage is the GNU Make build stage. In this case, the job server implements a GNU Make job server interface, which involves reading and writing tokens from file handles passed as build flags to the Make command.

For top-level jobs, additional resources are monitored in addition to the number of workers. Both system load and memory utilization checks can be enabled to prevent overloading a system.

### 22.3 Executor Job Failure Behavior

The executor’s behavior when a job fails can be modified with the following two parameters:

- **continue_on_failure** Continue executing jobs even if one job fails. If this is set to `false` (the default), it will cause the executor to abandon all pending and queued jobs and stop after the first failure. Note that active jobs will still be allowed to complete before the executor returns.

- **continue_without_deps** Continue executing jobs even if one or more of their dependencies have failed. If this is set to `false` (the default), it will cause the executor to abandon only the jobs which depend on the failed job. If it is set to `true`, then it will build dependent jobs regardless.
22.4 Jobs and Job Stages

As mentioned above, a job is a set of dependencies and a sequence of job stages. Jobs and stages are constructed before a given task starts executing, and hold only specifications of what needs to be done to complete them. All stages are given a label for user introspection, a logger interface, and can either require or not require allocation of a worker from the job server.

Stage execution is performed asynchronously by Python’s asyncio module. This means that exceptions thrown in job stages are handled directly by the executor. It also means job stages can be interrupted easily through Python’s normal signal handling mechanism.

Stages can either be command stages (sub-process commands) or function stages (python functions). In either case, loggers used by stages support segmentation of stdout and stderr from job stages for both real-time introspection and logging.

22.4.1 Command Stages

In addition to the basic arguments mentioned above, command stages are parameterized by the standard sub-process command arguments including the following:

- The command, itself, and its arguments,
- The working directory for the command,
- Any additional environment variables,
- Whether to use a shell interpreter
- Whether to emulate a TTY
- Whether to partition stdout and stderr

When executed, command stages use asyncio’s asynchronous process executor with a custom I/O protocol.

22.4.2 Function Stages

In addition to the basic arguments mentioned above, function stages are parameterized by a function handle and a set of function-specific Python arguments and keyword arguments. When executed, they use the thread pool mentioned above.

Since the function stages aren’t sub-processes, I/O isn’t piped or redirected. Instead, a custom I/O logger is passed to the function for output. Functions used as function stages should use this logger to write to stdout and stderr instead of using normal system calls.

22.5 Introspection via Executor Events

Introspection into the different asynchronously-executed components of a task is performed by a simple event queue. Events are created by the executor, loggers, and stages, and they are consumed by an output controller. Events are defined by an event identifier and a data payload, which is an arbitrary dictionary.

There are numerous events which correspond to changes in job states, but events are also used for transporting captured I/O from job stages.

The modeled events include the following:

- JOB_STATUS A report of running job states,
Fig. 3: **Executor Event Pipeline** – Above, the executor writes events to the event queue, and the I/O loggers used by function and command stages write output events as well. All of these events are handled by the output controller, which writes to the real `stdout` and `stderr`.

- QUEUED_JOB A job has been queued to be executed,
- STARTED_JOB A job has started to be executed,
- FINISHED_JOB A job has finished executing (succeeded or failed),
- ABANDONED_JOB A job has been abandoned for some reason,
- STARTED_STAGE A job stage has started to be executed,
- FINISHED_STAGE A job stage has finished executing (succeeded or failed),
- STAGE_PROGRESS A job stage has executed partially,
- STDOUT A status message from a job,
- STDERR A warning or error message from a job,
- SUBPROCESS A sub process has been created,
- MESSAGE Arbitrary string message
CHAPTER 23

Adding New Build Types

The current release of catkin_tools supports building two types of packages:

- **Catkin** – CMake packages that use the Catkin CMake macros
- **CMake** – “Plain” CMake packages

In order to fully support additional build types, numerous additions need to be made to the command-line interfaces so that the necessary parameters can be passed to the `build` verb. For partial support, however, all that’s needed is to add a build type identifier and a function for generating build jobs.

The supported build types are easily extendable using the setuptools entry_points interface without modifying the catkin_tools project, itself. Regardless of what package the entry_point is defined in, it will be defined in the setup.py of that package, and will take this form:

```python
from setuptools import setup

setup(
    ...
    entry_points={
        ...
        'catkin_tools.jobs': [
            'mybuild = my_package.some.module:description',
        ],
    },
}
```

This entry in the setup.py places a file in the PYTHONPATH when either the install or the develop verb is given to setup.py. This file relates the key (in this case mybuild) to a module and attribute (in this case my_package.some.module and description).

Then the catkin command will use the pkg_resources modules to retrieve these mapping at run time. Any entry for the catkin_tools.jobs group must point to a description attribute of a module, where the description attribute is a dict. The description dict should take this form:
This dict defines all the information that the catkin command needs to create jobs for the mybuild build type. The build_type key takes a string which is the build type identifier. The description key takes a string which briefly describes the build type. The create_build_job key takes a callable (function) factory which is called in order to create a Job to build a package of type mybuild.

The signature of the factory callable should be similar to the following:

```python
def create_mybuild_build_job(context, package, package_path, dependencies, **kwargs):
    # Initialize empty list of build stages
    stages = []

    # Add stages required to build `mybuild`-type packages,
    # based on the configuration context.
    # ...

    # Create and return new build Job
    return Job(
        jid=package.name,
        deps=dependencies,
        stages=stages)
```

 CHAPTER 24

Extending the `catkin` command

The `catkin` command is designed to be easily extendable using the `setuptools` `entry_points` interface without modifying the `catkin_tools` project itself. Regardless of what package the `entry_point` is defined in, it will be defined in the `setup.py` of that package, and will take this form:

```python
from setuptools import setup

setup(
    ...
    entry_points={
        ...
        'catkin_tools.commands.catkin.verbs': [
            # Example from catkin_tools' setup.py:
            # 'list = catkin_tools.verbs.catkin_list:description',
            'my_verb = my_package.some.module:description',
        ],
    },
)
```

This entry in the `setup.py` places a file in the `PYTHONPATH` when either the `install` or the `develop` verb is given to `setup.py`. This file relates the key (in this case `my_verb`) to a module and attribute (in this case `my_package.some.module` and `description`). Then the `catkin` command will use the `pkg_resources` modules to retrieve these mappings at run time. Any entry for the `catkin_tools.commands.catkin.verbs` group must point to a `description` attribute of a module, where the `description` attribute is a `dict`. The `description` dict should take this form (the `description` from the `build` verb for example):

```python
description = dict(
    verb='build',
    description="Builds a catkin workspace",
    main=main,
    prepare_arguments=prepare_arguments,
    argument_preprocessor=argument_preprocessor,
)
```

This `dict` defines all the information that the `catkin` command needs to provide and execute your verb. The `verb`
key takes a string which is the verb name (as shown in help and used for invoking the verb). The description key takes a string which is the description which is shown in the catkin -h output. The main key takes a callable (function) which is called when the verb is invoked. The signature of the main callable should be like this:

```python
def main(opts):
    # ...
    return 0
```

Where the `opts` parameter is the `Namespace` object returns from `ArgumentParser.parse_args(...)` and should return an exit code which is passed to `sys.exit`.

The `prepare_arguments` key takes a function with this signature:

```python
def prepare_arguments(parser):
    add = parser.add_argument
    # What packages to build
    add('packages', nargs='*',
        help='Workspace packages to build, package dependencies are built as well unless --no-deps is used.
        'If no packages are given, then all the packages are built.')
    add('--no-deps', action='store_true', default=False,
        help='Only build specified packages, not their dependencies.')

    return parser
```

The above example is a snippet from the `build` verb's `prepare_arguments` function. The purpose of this function is to take a given `ArgumentParser` object, which was created by the `catkin` command, and add this verb's `argparse` arguments to it and then return it.

Finally, the `argument_preprocessor` command is an optional entry in the `description_dict` which has this signature:

```python
def argument_preprocessor(args):
    """Processes the arguments for the build verb, before being passed to argparse""
    # CMake/make pass-through flags collect dashed options. They require special
    # handling or argparse will complain about unrecognized options.
    if args is None else
    extract_make_args = extract_cmake_and_make_and_catkin_make_arguments
    args, cmake_args, make_args, catkin_make_args = extract_make_args(args)
    # Extract make jobs flags.
    jobs_flags = extract_jobs_flags(' '.join(args))
    if jobs_flags:
        jobs_flags = re.sub(jobs_flags, '', ' '.join(args)).split()
    extras = {
        'cmake_args': cmake_args,
        'make_args': make_args + (jobs_flags or []),
        'catkin_make_args': catkin_make_args,
    }
    return args, extras
```

The above example is the `argument_preprocessor` function for the `build` verb. The purpose of the `argument_preprocessor` callable is to allow the verb to preprocess its own arguments before they are passed to `argparse`. In the case of the `build` verb, it is extracting the CMake and Make arguments before having them passed to `argparse`. The input parameter to this function is the list of arguments which come after the verb, and this function is only called when this verb has been detected as the first positional argument to the `catkin` command. So, you do not need to worry about making sure the arguments you just got are yours. This function should return a tuple where the first item in the tuple is the potentially modified list of arguments, and the second item is a dictionary of
keys and values which should be added as attributes to the opts parameter which is later passed to the main callable. In this way you can take the arguments for your verb, parse them, remove some, add some or whatever, then you can additionally return extra information which needs to get passed around the argparse parse_args function. Most verbs should not need to do this, and in fact the built-in list verb’s description dict does not include one:

```python
description = dict(
    verb='list',
    description="Lists catkin packages in a given folder",
    main=main,
    prepare_arguments=prepare_arguments,
)
```

Hopefully, this information will help you get started when you want to extend the catkin command with custom verbs.

This Python package provides command line tools for working with the catkin meta-buildsystem and catkin workspaces. These tools are separate from the Catkin CMake macros used in Catkin source packages. For documentation on creating catkin packages, see: http://docs.ros.org/api/catkin/html/

**Note:** This package was announced in March 2015 and is still in beta. See the GitHub Milestones for the current release schedule and roadmap.

**Note:** Users of catkin_make and catkin_make_isolated should go to the Migration Guide for help transitioning to catkin_build.
The `catkin` Command-Line Interface (CLI) tool is the single point of entry for most of the functionality provided by this package. All invocations of the `catkin` CLI tool take this form:

```
$ catkin [global options] <verb> [verb arguments and options]
```

The different capabilities of the `catkin` CLI tool are organized into different sub-command “verbs.” This is similar to common command-line tools such as `git` or `apt-get`. Verbs include actions such as `build` which builds a `catkin` workspace or `list` which simply lists the `catkin` packages found in one or more folders.

Verbs can take arbitrary arguments and options, but they must all come after the verb. For more help on the usage of a particular verb, simply pass the `-h` or `--help` option after the verb.

### 25.1 Built-in `catkin` Verbs

Each of the following verbs is built-in to the `catkin` command and has its own detailed documentation:

- **build** – Build packages in a `catkin` workspace
- **config** – Configure a `catkin` workspace’s layout and settings
- **clean** – Clean products generated in a `catkin` workspace
- **create** – Create structures like `Catkin` packages
- **env** – Run commands with a modified environment
- **init** – Initialize a `catkin` workspace
- **list** – Find and list information about `catkin` packages in a workspace
- **locate** – Get important workspace directory paths
- **profile** – Manage different named configuration profiles
- **test** – Test one or more packages in a `catkin` workspace
25.2 Contributed Third Party Verbs

- lint – Check catkin packages for common errors

25.3 Shell Support for the catkin Command

If you are using bash or zsh, then you can source an extra setup file to gain access to some additional verbs. For more information see: Shell support in catkin command.

25.4 Extending the catkin command

If you would like to add a verb to the catkin command without modifying its source, please read Adding New Verbs.