catkin_tools Documentation

Release 0.0.0

William Woodall

December 21, 2015

Contents

CHAPTER 1

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 behaviour 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
$ 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 python-catkin-tools
```

1.2 Installing on other platforms with pip

Simply install it with `pip`:

```
$ sudo pip install -U catkin_tools
```

1.3 Installing from source

First clone the source for `catkin_tools`:

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

Then install with the `setup.py` file:

```
$ python setup.py install
```

1.3.1 Developing

Listed here are some useful tips for developing against `catkin_tools`.

1.4 Install `catkin_tools` for developing

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

```
$ python 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:

```
$ python setup.py develop -u
```

CHAPTER 2

Quickstart

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 thorogh usage details see the individual verb documentation.

2.1 TL;DR

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

```
$ mkdir -p /tmp/path/to/my_catkin_ws/src      # Make a new workspace and source space
$ cd /tmp/path/to/my_catkin_ws                # Navigate to the workspace root
$ catkin init                                 # Initialize it with a hidden marker file
$ cd /tmp/path/to/my_catkin_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 build                                # Build all packages in the workspace
$ source ../devel/setup.bash                  # Load the workspace's environment
$ catkin clean --all                          # Clean the build products
$ catkin build pkg_d                          # Build `pkg_d` and its deps
$ cd /tmp/path/to/my_catkin_ws/src/pkg_c      # Navigate to `pkg_c`'s source directory
$ catkin build --this                         # Build `pkg_c` and its deps
$ catkin build --this --no-deps               # Rebuild only `pkg_c`
```

2.2 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:

```
$ 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
Build Space:       [missing] /tmp/path/to/my_catkin_ws/build
Devel Space:       [missing] /tmp/path/to/my_catkin_ws/devel
Install Space:     [missing] /tmp/path/to/my_catkin_ws/install
DESTDIR:                     None
--------------------------------------------------------------
Isolate Develspaces:         False
Install Packages:            False
Isolate Installs:            False
--------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
--------------------------------------------------------------
Workspace configuration appears valid.
--------------------------------------------------------------
```

Now the directory `/tmp/path/to/my_catkin_ws` 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 be 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 Configuration Summary. 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.

2.3 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 three trivial packages: `pkg_a`, `pkg_b`, and `another_one`:

```
$ cd /tmp/path/to/my_catkin_ws/src
$ catkin_create_pkg pkg_a
Created file pkg_a/CMakeLists.txt
Created file pkg_a/package.xml
Successfully created files in /tmp/path/to/my_catkin_ws/src/pkg_a. Please adjust the values in packag
```

```
$ catkin_create_pkg pkg_b
Created file pkg_b/CMakeLists.txt
Created file pkg_b/package.xml
Successfully created files in /tmp/path/to/my_catkin_ws/src/pkg_b. Please adjust the values in packag
$ catkin_create_pkg another_one
Created file another_one/CMakeLists.txt
Created file another_one/package.xml
Successfully created files in /tmp/path/to/my_catkin_ws/src/another_one. Please adjust the values in
```

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

```
$ cd /tmp/path/to/my_catkin_ws
$ tree -aL 2
.
-- .catkin_tools
|   -- README
-- src
    -- another_one
    -- pkg_a
    -- pkg_b
```

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 not only will temporary or final build products never be placed in a package's source directory (or anywhere in the **source space** for that matter), but also all build directories are aggregated in the **build space** and all final build products (executables, libraries, etc.) will be put in the **devel space**.

2.4 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
--------------------------------------------------------------
Profile:                     default
Extending:                   None
Workspace:                   /tmp/path/to/my_catkin_ws
Source Space:       [exists] /tmp/path/to/my_catkin_ws/src
Build Space:       [missing] /tmp/path/to/my_catkin_ws/build
Devel Space:       [missing] /tmp/path/to/my_catkin_ws/devel
Install Space:     [missing] /tmp/path/to/my_catkin_ws/install
DESTDIR:                     None
--------------------------------------------------------------
Isolate Develspaces:         False
Install Packages:            False
Isolate Installs:            False
--------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
--------------------------------------------------------------
Workspace configuration appears valid.
--------------------------------------------------------------
```

```
Found '3' packages in 0.0 seconds.
Starting ==> another_one
Starting ==> pkg_a
Starting ==> pkg_b
Finished <== pkg_b       [ 2.0 seconds ]
Finished <== another_one [ 2.0 seconds ]
Finished <== pkg_a       [ 3.4 seconds ]
[build] Finished.
[build] Runtime: 3.4 seconds
```

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 (to one level deep):

```
$ cd /tmp/path/to/my_catkin_ws
$ tree -aL 1
.
-- build
-- .catkin_tools
-- devel
-- src
```

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.

2.5 Loading the Workspace Environment

In order to properly "use" the products of the workspace, it's 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:

```
$ source /tmp/path/to/my_catkin_ws/devel/setup.bash
```

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.

2.6 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 --all` 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 any following command:

```
$ catkin clean --build --devel
Removing buildspace: /tmp/path/to/my_catkin_ws/build
Removing develspace: /tmp/path/to/my_catkin_ws/devel
```

For more information on less aggressive cleaning options see the clean verb documentation.

CHAPTER 3

Workspace Mechanics

This chapter defines how Catkin workspaces are organized and used, as well as some standardized nomenclature for describing elements of a Catkin workspace. 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. As such, these projects need to be organized in a "workspace" which contains both the source and build products for a collection of "packages."

3.1 Anatomy of a Catkin Workspace

A standard catkin workspace, as defined by REP-0128, is a folder with a prescribed set "spaces", each of which is normally a folder within the workspace:

- **source space** – default: `./src`
- **build space** – default: `./build`
- **devel space** – default: `./devel`
- **install space** – default: `./install`

Though there are standard conventions for the layout and names of the workspace's various spaces, they can be renamed for a given build using the `catkin config` verb.

3.1.1 source space

The **source space** is where the code for your packages resides and normally is in the folder `/path/to/workspace/src`. The build command considers **source space** to be read-only, in that during a build no files or folders should be created or modified in that folder. Therefore catkin workspaces are said to be built "out of source", which simply means that the folder in which you build your code is not under or part of the folder with contains the source code.

3.1.2 build space

Temporary build files are put into the **build space**, which by default is in the `/path/to/workspace/build` folder. The **build space** is the working directory in which commands like `cmake` and `make` are run.

3.1.3 devel space

Generated files, like executables, libraries, pkg-config files, CMake config files, or message code, are placed in the **devel space**. By convention the **devel space** is located as a peer to the **source space** and **build space** in the `/path/to/workspace/devel` folder. The layout of the **devel space** is intended to mimic the root of an FHS filesystem, with folders like `include`, `lib`, `bin`, and `share`. Running code is possible from **devel space** because references to the **source space** are created.

3.1.4 install space

Finally, if the packages in the workspace are setup for installing, the `--install` option can be invoked to install the packages to the `CMAKE_INSTALL_PREFIX`, which in REP-0128 terms is the **install space**. The **install space**, like the **devel space**, has an FHS layout along with some generated setup files. The **install space** is set to `/path/to/workspace/install` by changing the `CMAKE_INSTALL_PREFIX` by default. This is done to prevent users from accidentally trying to install to the normal `CMAKE_INSTALL_PREFIX` path, `/usr/local`. Unlike the **devel space**, the **install space** is completely standalone and does not require the **source space** or **build space** to function, and is suitable for packaging.

3.2 Additional Workspace Directories with `catkin_tools`

3.2.1 Hidden Marker / Config 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 than `default`.

3.2.2 Build Log Directory

Another addition is the `build_logs` directory which is generated in the **build space** and contains individual build logs for each package.

3.3 Environment Setup Files

In addition to the FHS folders, some setup scripts are generated in the **devel space**, e.g. `setup.bash` or `setup.zsh`. These setup scripts are intended to make it easier to use the resulting **devel space** for building on top of the packages that were just built or for running programs built by those packages. The setup script can be used like this in `bash`:

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

Or like this in `zsh`:

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

Sourcing these setup scripts adds this workspace and any "underlaid" workspaces to your environment, prefixing the `CMAKE_PREFIX_PATH`, `PKG_CONFIG_PATH`, `PATH`, `LD_LIBRARY_PATH`, `CPATH`, and `PYTHONPATH` with local workspace folders.

The setup scripts will also execute any shell hooks exported by packages in the workspace, which is how `roslib`, for example, sets the `ROS_PACKAGE_PATH` environment variable.

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 loose the functionality provided by them, namely extending the environment and executing environment hooks.

3.4 Workspace Packages and Dependencies

A workspace's packages consist of any packages found in the **source space**. A package is any folder which contains a `package.xml` as defined in REP-0127.

The `catkin build` command determines the order in which packages are built based on the `depend`, `build_depend`, `run_depend`, and `build_type` tags in a package's `package.xml` file.

- The `*_depend` tags are used to determine the topological build order of the packages.
- The `build_type` tag is used to determine which build work flow to use on the package.

Packages without an explicitly defined `build_type` tag are assumed to be catkin packages, but 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 has been done in the past for building packages like `opencv`, `pcl`, and `flann`.

3.5 Understanding the Build Process

3.5.1 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 amongst 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.

3.5.2 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 buildtools 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.

3.5.3 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
- 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

3.6 Workspace Chaining and the Importance of CMAKE_PREFIX_PATH

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.

On one hand, this makes it easy and automatic to chain workspaces. At the same time, however, 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 unaware of Catkin's use of `CMAKE_PREFIX_PATH`

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.

CHAPTER 4

Configuration Summary

4.1 Contents of the Config Summary

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

```
$ 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
Build Space:       [missing] /tmp/path/to/my_catkin_ws/build
Devel Space:       [missing] /tmp/path/to/my_catkin_ws/devel
Install Space:     [missing] /tmp/path/to/my_catkin_ws/install
DESTDIR:                     None
--------------------------------------------------------------
Isolate Develspaces:         False
Install Packages:            False
Isolate Installs:            False
--------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
--------------------------------------------------------------
Whitelisted Packages:        None
Blacklisted Packages:        None
--------------------------------------------------------------
Workspace configuration appears valid.
--------------------------------------------------------------
```

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:

4.1.1 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*

```
Extending:                     None
```

  - *Implicit Chaining* – Derived from the `CMAKE_PREFIX_PATH` environment or cache variable.

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

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

  - *Explicit Chaining* – Specified by `catkin config --extend`

```
Extending:         [explicit] /opt/ros/hydro
```

- **[* Space]** – Lists the paths to each of the catkin "spaces" and whether or not they exist
- **DESTDIR** – An optional prefix to the **install space** as defined by GNU Standards
- **Isolate Develspaces** – Builds products (like libraries and binaries) into individual FHS subdirectories in the **devel space**, instead of a single FHS directory
- **Install Packages** – Enable creating and installation into the **install space**
- **Isolate Installs** – Installs products into individual FHS subdirectories in the **install space**
- **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.
- **Package Whitelist** – These are the packages that will be built with a bare call to `catkin build`
- **Package Blacklist** – These are the packages that will *not* be built unless explicitly named

4.1.2 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.

4.1.3 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.

4.2 Workspace Chaining Mode

An important property listed in the configuration 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." For more details on this see the details about workspace chaining and `CMAKE_PREFIX_PATH` in Workspace Mechanics.

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

4.2.1 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
```

4.2.2 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/hydro/setup.bash` will normally see an `Extending` value such as:

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

If you don't want to extend the given workspace, unsetting the `CMAKE_PREFIX_PATH` environment variable will change it back to none. You can also alternatively

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/hydro
```

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

4.2.3 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
```

CHAPTER 5

Cheat Sheet

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.

5.1 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 explicity extends another workspace:

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

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`

5.2 Configuring Workspaces

View the current configuration:

- `catkin config`

Setting and un-setting 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`

5.3 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 aditional 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 /path/to/folder)`

... or in the current folder:

- `catkin build $(catkin list -u .)`

5.4 Cleaning Build Products

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

- `catkin clean -a`

... or just the build space:

- `catkin clean --build`

... or just delete the *CMakeCache.txt* files for each package:

- `catkin clean --cmake-cache`

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

- `catkin clean --orphans`

5.5 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`

5.6 Profile Cookbook

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

```
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:

```
catkin config --profile my_pkg -x _my_pkg_test
catkin build --profile my_pkg my_pkg
catkin clean --profile my_pkg --all
```

5.7 Manipulating Workspace Chaining

Change from implicit to explicit chaining:

```
catkin clean -a
catkin config --extend /opt/ros/hydro
```

Change from explicit to implicit chaining:

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

5.8 Building With Other Jobservers

Build with `distcc`:

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

CHAPTER 6

Troubleshooting

6.1 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.

6.1.1 Missing Workspace Components

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

6.1.2 Inconsistent Environment

- The `CMAKE_PREFIX_PATH` environment variable is different than the cahced `CMAKE_PREFIX_PATH`
- The explicitly extended workspace path yeilds 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

6.2 Dependency Resolution

6.2.1 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.
- Run `catkin list --deps /path/to/ws/src` to list the dependencies of each package and look for errors.

CHAPTER 7

`catkin build` – Build Packages

The `build` verb for the `catkin` command is used to build one or more packages in a catkin workspace. Like most `catkin` verbs, the `catkin build` verb is context-aware. This means that it can be executed from within any directory contained by an *initialized* workspace.

If a workspace is not yet initialized, `catkin build` can initialize it, but only if it is called from the workspace root and the default workspace structure is desired.

Workspaces can also be built from arbitrary working directories if the user specifies the path to the workspace with the `--workspace` option.

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

```
$ mkdir -p /tmp/path/to/my_catkin_ws/src
$ cd /tmp/path/to/my_catkin_ws
$ catkin init
$ cd /tmp/path/to/my_catkin_ws/src
$ rosinstall_generator --deps ros_tutorials > .rosinstall
$ wstool update
```

7.1 Basic Usage

Consider a Catkin workspace with a **source space** populated with the following Catkin packages which have yet to be built:

```
$ pwd
/tmp/path/to/my_catkin_ws

$ ls ./*
./src:
catkin              console_bridge     genlisp            genpy
message_runtime     ros_comm           roscpp_core        std_msgs
common_msgs         gencpp             genmsg             message_generation
ros                 ros_tutorials      rospack
```

7.1.1 Previewing The Build

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

```
$ catkin build --dry-run
--------------------------------------------------------------
Profile:                     default
Extending:                   None
Workspace:                   /tmp/path/to/my_catkin_ws
Source Space:       [exists] /tmp/path/to/my_catkin_ws/src
Build Space:       [missing] /tmp/path/to/my_catkin_ws/build
Devel Space:       [missing] /tmp/path/to/my_catkin_ws/devel
Install Space:     [missing] /tmp/path/to/my_catkin_ws/install
DESTDIR:                     None
--------------------------------------------------------------
Isolate Develspaces:         False
Install Packages:            False
Isolate Installs:            False
--------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
--------------------------------------------------------------
Whitelisted Packages:        None
Blacklisted Packages:        None
--------------------------------------------------------------
Workspace configuration appears valid.
--------------------------------------------------------------
Found '36' packages in 0.0 seconds.
Packages to be built:
- catkin               (catkin)
- genmsg               (catkin)
- gencpp               (catkin)
- genlisp              (catkin)
- genpy                (catkin)
- console_bridge       (cmake)
- cpp_common           (catkin)
- message_generation   (catkin)
- message_runtime      (catkin)
- ros_tutorials        (metapackage)
- rosbuild             (catkin)
- rosclean             (catkin)
- roscpp_traits        (catkin)
- rosgraph             (catkin)
- roslang              (catkin)
- roslaunch            (catkin)
- rosmaster            (catkin)
- rospack              (catkin)
- roslib               (catkin)
- rosparam             (catkin)
- rospy                (catkin)
- rostime              (catkin)
- roscpp_serialization (catkin)
- rosunit              (catkin)
- rosconsole           (catkin)
- rostest              (catkin)
- std_msgs             (catkin)
- geometry_msgs        (catkin)
- rosgraph_msgs        (catkin)
- std_srvs             (catkin)
- xmlrpcpp             (catkin)
- roscpp               (catkin)
```

```
- roscpp_tutorials     (catkin)
- rosout               (catkin)
- rospy_tutorials      (catkin)
- turtlesim            (catkin)
Total packages: 36
```

In addition to the listing the package names and in which order they would be built, it also displays the buildtool type of each package. Among those listed above are:

- **catkin** – A CMake package which uses Catkin
- **cmake** – A "vanilla" CMake package
- **metapackage** – A package which contains no build products, but just groups other packages together for distribution

7.1.2 Building Specific Packages

In addition to the usage above, the `--dry-run` option will show what the behavior of `catkin build` will be with various other options. For example, the following will happen when you specify a single package to build:

```
$ catkin build roscpp_tutorials --dry-run
....
Found '36' packages in 0.1 seconds.
Packages to be built:
- catkin               (catkin)
- genmsg               (catkin)
- gencpp               (catkin)
- genlisp              (catkin)
- genpy                (catkin)
- console_bridge       (cmake)
- cpp_common           (catkin)
- message_generation   (catkin)
- message_runtime      (catkin)
- rosbuild             (catkin)
- roscpp_traits        (catkin)
- roslang              (catkin)
- rospack              (catkin)
- roslib               (catkin)
- rostime              (catkin)
- roscpp_serialization (catkin)
- rosunit              (catkin)
- rosconsole           (catkin)
- std_msgs             (catkin)
- rosgraph_msgs        (catkin)
- xmlrpcpp             (catkin)
- roscpp               (catkin)
- roscpp_tutorials     (catkin)
Total packages: 23
```

As shown above, only 23 packages (`roscpp_tutorials` and its dependencies), of the total 36 packages would be built.

7.1.3 Skipping Packages

Suppose you built every package up to `roscpp_tutorials`, 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 `roscpp_tutorials` would be re-built, but it would still take some time for the underlying byuildsystem 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. (The following example uses the `--dry-run` option again to preview the behavior):

```
$ catkin build roscpp_tutorials --start-with roscpp_tutorials --dry-run
....
Found '36' packages in 0.0 seconds.
Packages to be built:
(skip) catkin                (catkin)
(skip) genmsg                (catkin)
(skip) gencpp                (catkin)
(skip) genlisp               (catkin)
(skip) genpy                 (catkin)
(skip) console_bridge        (cmake)
(skip) cpp_common            (catkin)
(skip) message_generation    (catkin)
(skip) message_runtime       (catkin)
(skip) rosbuild              (catkin)
(skip) roscpp_traits         (catkin)
(skip) roslang               (catkin)
(skip) rospack               (catkin)
(skip) roslib                (catkin)
(skip) rostime               (catkin)
(skip) roscpp_serialization  (catkin)
(skip) rosunit               (catkin)
(skip) rosconsole            (catkin)
(skip) std_msgs              (catkin)
(skip) rosgraph_msgs         (catkin)
(skip) xmlrpcpp              (catkin)
(skip) roscpp                (catkin)
------ roscpp_tutorials      (catkin)
Total packages: 23
```

However, you should be careful when using the `--start-with` option, as `catkin build` will assume that all dependencies leading up to that package have already been successfully built.

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.

```
$ catkin build roscpp_tutorials --no-deps roscpp_tutorials --dry-run
....
Found '36' packages in 0.0 seconds.
Packages to be built:
- roscpp_tutorials     (catkin)
Total packages: 1
```

7.1.4 Build Products

At this point the workspace has not been modified, but once we tell the `build` verb to actually build the workspace then directories for a **build space** and a **devel space** will be created:

```
$ catkin build
Creating buildspace directory, '/tmp/path/to/my_catkin_ws/build'
------------------------------------------------------------
Profile:                     default
```

```
Extending:                  None
Workspace:                  /tmp/path/to/my_catkin_ws
Source Space:       [exists] /tmp/path/to/my_catkin_ws/src
Build Space:       [missing] /tmp/path/to/my_catkin_ws/build
Devel Space:       [missing] /tmp/path/to/my_catkin_ws/devel
Install Space:     [missing] /tmp/path/to/my_catkin_ws/install
DESTDIR:                    None
--------------------------------------------------------------
Isolate Develspaces:        False
Install Packages:           False
Isolate Installs:           False
--------------------------------------------------------------
Additional CMake Args:      None
Additional Make Args:       None
Additional catkin Make Args: None
--------------------------------------------------------------
Whitelisted Packages:       None
Blacklisted Packages:       None
--------------------------------------------------------------
Workspace configuration appears valid.
--------------------------------------------------------------
Found '36' packages in 0.0 seconds.
Starting ==> catkin
Starting ==> console_bridge
Finished <== catkin [ 2.4 seconds ]

....

[build] Finished.
[build] Runtime: 3 minutes and 54.6 seconds
```

Since no packages were given as arguments, `catkin build` built all of the packages in the workspace.

As shown above, after the build finishes, we now have a **build space** with a folder containing the intermediate build products for each package and a **devel space** with an FHS layout into which all the final build products have been written.

```
$ ls ./*
./build:
catkin               genlisp             message_runtime      roscpp
rosgraph_msgs        rosout              rostest              turtlesim
build_logs           genmsg              ros_tutorials
roscpp_serialization roslang             rospack              rostime
xmlrpcpp             console_bridge      genpy                rosbuild
roscpp_traits        roslaunch           rosparam             rosunit
cpp_common           geometry_msgs       rosclean
roscpp_tutorials     roslib              rospy                std_msgs
gencpp               message_generation  rosconsole           rosgraph
rosmaster            rospy_tutorials     std_srvs

./devel:
_setup_util.py bin            env.sh         etc            include
lib            setup.bash     setup.sh       setup.zsh      share

./src:
catkin             console_bridge     genlisp            genpy
message_runtime    ros_comm           roscpp_core        std_msgs
common_msgs        gencpp             genmsg             message_generation
ros                ros_tutorials      rospack
```

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**.

7.2 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:

```
$ cd /tmp/path/to/my_catkin_ws/src/roscpp_tutorials
$ catkin build --this --dry-run
....
Found '36' packages in 0.0 seconds.
Packages to be built:
- roscpp_tutorials     (catkin)
Total packages: 1
```

7.3 Controlling the Number of 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 jobserver. If your platform doesn't support jobserver 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 jobserver, 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.

7.4 Controlling Command-Line Output

7.4.1 Status Line

While running `catkin build` with default options, you would have seen the "live" status lines similar to the following:

```
[build - 5.9] [genmsg - 1.3] [message_runtime - 0.7] ...        [4/4 Active | 3/36 Completed]
```

This 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:

- **Total Build Time** – The first block on the left, indicates the total elapsed build time in seconds thus far. Above, `[build - 5.9]` means that the build has been running for a total of `5.9` seconds.

- **Active Job Status** – The next blocks show the currently active jobs with as name of the package being built and the elapsed time for that job, in seconds. The above block like `[genmsg - 1.3]` means that the `genmsg` package is currently being built, and it has been building for `1.3` seconds.
- **Active and Completed Counts** – The final block, justified to the right, is the number of packages being actively built out of the total allowed parallel jobs (specified with the `-p` options) as well as the number of completed packages out of the total. Above, the block `[4/4 Active | 3/36 Completed]` means that there are four out of four jobs active and three of the total 36 packages to be built have been completed.

This status line can be disabled by passing the `--no-status` option to `catkin build`.

7.4.2 Package Build Messages

Normally, unless an error occurs, the output from each package's build proces 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 ==> genmsg
Finished <== genmsg [ 2.4 seconds ]
```

However, if you would like to see more of the messages from the underlying buildsystem, you can invoke the `-v` or `--verbose` option. This will print the normal message when a package build starts and finished as well as the output of each build command in a block, once it finishes:

```
Starting ==> catkin

[catkin]: ==> '/path/to/my_catkin_ws/build/catkin/build_env.sh /usr/local/bin/cmake /path/to/my_catki
-- The C compiler identification is Clang 5.0.0
-- The CXX compiler identification is Clang 5.0.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Using CATKIN_DEVEL_PREFIX: /path/to/my_catkin_ws/devel/catkin
-- Using CMAKE_PREFIX_PATH: /path/to/my_catkin_ws/install
-- This workspace overlays: /path/to/my_catkin_ws/install
-- Found PythonInterp: /usr/bin/python (found version "2.7.5")
-- Using PYTHON_EXECUTABLE: /usr/bin/python
-- Python version: 2.7
-- Using default Python package layout
-- Found PY_em: /Library/Python/2.7/site-packages/em.pyc
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /path/to/my_catkin_ws/build/catkin/test_results
-- Found gtest: gtests will be built
-- catkin 0.5.86
-- Configuring done
-- Generating done
-- Build files have been written to: /path/to/my_catkin_ws/build/catkin
[catkin]: <== '/path/to/my_catkin_ws/build/catkin/build_env.sh /usr/local/bin/cmake /path/to/my_catki

[catkin]: ==> '/path/to/my_catkin_ws/build/catkin/build_env.sh /usr/bin/make -j4 -l4' in '/path/to/my
[catkin]: <== '/path/to/my_catkin_ws/build/catkin/build_env.sh /usr/bin/make -j4 -l4' finished with r
```

```
[catkin]: ==> '/path/to/my_catkin_ws/build/catkin/build_env.sh /usr/bin/make install' in '/path/to/my
Install the project...
-- Install configuration: ""
... truncated for brevity
[catkin]: <== '/path/to/my_catkin_ws/build/catkin/build_env.sh /usr/bin/make install' finished with r

Finished <== catkin [ 3.4 seconds ]
```

Note: The printing of these command outputs maybe be interleaved with commands from other package builds if more than one package is being built at the same time.

If you want to see the output from commands streaming to the screen, then you can use the `-i` or `--interleave` option. This option will cause the output from commands to be pushed to the screen immediately, instead of buffering until the command finishes. This ends up being pretty confusing, so when interleaved output is used `catkin build` prefixes each line with `[<package name>]:` like this:

```
[roscpp_traits]: ==> '/Users/william/my_catkin_ws/build/roscpp_traits/build_env.sh /usr/bin/make cmak
[ros_tutorials]: -- The CXX compiler identification is Clang 5.0.0
[ros_tutorials]: -- Check for working C compiler: /usr/bin/cc
[roscpp_traits]: ==> '/Users/william/my_catkin_ws/build/roscpp_traits/build_env.sh /usr/bin/make -j4
[rosbuild]: ==> '/Users/william/my_catkin_ws/build/rosbuild/build_env.sh /usr/bin/make -j4 -l4' in '/
[rosclean]: -- The C compiler identification is Clang 5.0.0
[ros_tutorials]: -- Check for working C compiler: /usr/bin/cc -- works
[ros_tutorials]: -- Detecting C compiler ABI info
[rosclean]: -- The CXX compiler identification is Clang 5.0.0
[rosclean]: -- Check for working C compiler: /usr/bin/cc
```

Note: When you use `-p 1` and `-v` at the same time, `-i` is implicitly added.

7.5 Running Tests Built in a Workspace

Running tests for a given package typically is done by invoking a special `make` target like `test` or `run_tests`. catkin packages all define the `run_tests` target which aggregates all types of tests and runs them together. So in order to get tests to build and run for your packages you need to pass them this additional `run_tests` or `test` target as a command line option to `make`.

To run catkin tests for all catkin packages in the workspace, use the following:

```
$ catkin run_tests
```

Or the longer version:

```
$ catkin build [...] --catkin-make-args run_tests
```

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

```
$ catkin run_tests --no-deps --this
```

For non-catkin packages which define a `test` target, you can do this:

```
$ catkin build [...] --make-args test
```

If you want to run tests for just one package, then you should build that package and this narrow down the build to just that package with the additional make argument:

```
$ # First build the package
$ catkin build package
...
$ # Then run its tests
$ catkin build package --no-deps --catkin-make-args run_tests
$ # Or for non-catkin packages
$ catkin build package --no-deps --make-args test
```

For catkin packages and the `run_tests` target, failing tests will not result in an non-zero exit code. So if you want to check for failing tests, use the `catkin_test_results` command like this:

```
$ catkin_test_results build/<package name>
```

The result code will be non-zero unless all tests passed.

7.6 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:

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

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

7.7 Debugging Build Errors

As mentioned above, by default the output from each build is optimistically hidden to give a clean overview of the workspace build, but when there is a problem with a build a few things happen.

First, the package with a failure prints the failing command's output to the screen between some enclosing lines:

```
[rospack]: ==> '/path/to/my_catkin_ws/build/rospack/build_env.sh /usr/bin/make -j4 -l4' in '/path/to/
[ 66%] Built target rospack
make[1]: *** [CMakeFiles/rosstackexe.dir/all] Interrupt: 2
make[1]: *** [CMakeFiles/rospackexe.dir/all] Interrupt: 2
make: *** [all] Interrupt: 2
[rospack]: <== '/path/to/my_catkin_ws/build/rospack/build_env.sh /usr/bin/make -j4 -l4' failed with
```

And the status line is updated to reflect that that package has run into an issue by placing a `!` in front of it:

```
[build - 1.7] [!cpp_common] [!rospack] [genlisp - 0.3]        [1/1 Active | 10/23 Completed]
```

Then the `catkin build` command waits for the rest of the currently still building packages to finish (without starting new package builds) and then summarizes the errors for you:

```
[build] There were '2' errors:

Failed to build package 'cpp_common' because the following command:

    # Command run in directory: /path/to/my_catkin_ws/build/cpp_common
    /path/to/my_catkin_ws/build/cpp_common/build_env.sh /usr/bin/make -j4 -l4

Exited with return code: -2

Failed to build package 'rospack' because the following command:
```

```
    # Command run in directory: /path/to/my_catkin_ws/build/rospack
    /path/to/my_catkin_ws/build/rospack/build_env.sh /usr/bin/make -j4 -l4

Exited with return code: -2
Build summary:
 Successful catkin
 Successful genmsg
 ...
 Failed     cpp_common
 Failed     rospack
 Not built  roscpp_serialization
 Not built  roscpp
 ...
```

Packages marked as *Not built* were requested, but not yet built because catkin stopped due to failed packages.

To try to build as many requested packages as possible (instead of stopping after the first package failed), you can pass the `--continue-on-failure` option. Then the `catkin build` command will then continue building packages whose dependencies built successfully

If you don't want to scroll back up to find the error amongst the other output, you can `cat` the whole build log out of the `build_logs` folder in the **build space**:

```
$ cat build/build_logs/rospack.log
[rospack]: ==> '/path/to/my_catkin_ws/build/rospack/build_env.sh /usr/bin/make cmake_check_build_syst
[rospack]: <== '/path/to/my_catkin_ws/build/rospack/build_env.sh /usr/bin/make cmake_check_build_syst
[rospack]: ==> '/path/to/my_catkin_ws/build/rospack/build_env.sh /usr/bin/make -j4 -l4' in '/path/to/
[ 66%] Built target rospack
make[1]: *** [CMakeFiles/rosstackexe.dir/all] Interrupt: 2
make[1]: *** [CMakeFiles/rospackexe.dir/all] Interrupt: 2
make: *** [all] Interrupt: 2
[rospack]: <== '/path/to/my_catkin_ws/build/rospack/build_env.sh /usr/bin/make -j4 -l4' failed with r
```

7.8 Full Command-Line Interface

```
usage: catkin build [-h] [--workspace WORKSPACE] [--profile PROFILE]
                    [--dry-run] [--this] [--no-deps]
                    [--start-with PKGNAME | --start-with-this | --continue-on-failure]
                    [--force-cmake] [--no-install-lock] [--save-config]
                    [--parallel-jobs PARALLEL_JOBS]
                    [--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] [--no-notify]
                    [PKGNAME [PKGNAME ...]]

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.

Packages:
  Control which packages get built.

  PKGNAME               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.
  --start-with PKGNAME  Build a given package and those which depend on it,
                        skipping any before it.
  --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.
  --no-install-lock     Prevents serialization of the install steps, which is
                        on by default to prevent file install collisions

Config:
  Parameters for the underlying buildsystem.

  --save-config         Save any configuration options in this section for the
                        next build invocation.
  --parallel-jobs PARALLEL_JOBS, --parallel PARALLEL_JOBS, -p PARALLEL_JOBS
                        Maximum number of packages which could be built in
                        parallel (default is cpu count)
  --cmake-args ARG [ARG ...]
                        Arbitrary arguments which are passes 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 passes 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 passes 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).
```

```
Interface:
  The behavior of the command-line interface.

  --verbose, -v          Print output from commands in ordered blocks once the
                         command finishes.
  --interleave-output, -i
                         Prevents ordering of command output when multiple
                         commands are running at the same time.
  --no-status            Suppresses status line, useful in situations where
                         carriage return is not properly supported.
  --no-notify            Suppresses system popup notification.
```

CHAPTER 8

`catkin clean` – Clean Build Products

The `clean` verb makes it easier and safer to clean various products of a catkin workspace. In addition to removing entire **build**, **devel**, and **install spaces**, it also gives you more fine-grained control over removing just parts of these directories.

The `clean` verb is context-aware, but in order to work, it must be given the path to an initialized catkin workspace, or called from a path contained in an initialized catkin workspace. This is because the paths to the relevant spaces are contained in a workspace's metadata directory.

8.1 Full Command-Line Interface

```
usage: catkin clean [-h] [--workspace WORKSPACE] [--profile PROFILE] [-a] [-b]
                    [-d] [-i] [-c] [-o]

Deletes various products of the build verb.

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)

Basic:
  Clean workspace subdirectories.

  -a, --all             Remove all of the *spaces associated with the given or
                        active profile. This will remove everything but the
                        source space and the hidden .catkin_tools directory.
  -b, --build           Remove the buildspace.
  -d, --devel           Remove the develspace.
  -i, --install         Remove the installspace.

Advanced:
  Clean only specific parts of the workspace.

  -c, --cmake-cache     Clear the CMakeCache for each package, but leave build
                        and devel spaces.
  -o, --orphans         Remove only build directories whose source packages
                        are no longer enabled or in the source space. This
                        might require --force-cmake on the next build.
```

CHAPTER 9

`catkin config` – Configure a Workspace

The `config` verb can be used to both view and mapiulate a workspace's configuration options. These options include all of the elements listed in thr 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 uninitialied workspace will not automatically initialize it unless it is used with the `--init` option.

9.1 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.

9.2 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`

9.3 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
_setup_util.py bin            env.sh         etc            include
lib            setup.bash     setup.sh       setup.zsh      share
```

9.4 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.

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

For example, regardless of your current environment variable settings (like `$CMAKE_PREFIX_PATH`), this will build your workspace against the `/opt/ros/hydro` install space.

First start with an empty `CMAKE_PREFIX_PATH` and initialize, build, and source a workspace:

```
$ 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
...
--------------------------------------------------------------
Workspace configuration appears valid.
--------------------------------------------------------------

$ cd /tmp/path/to/my_catkin_ws
$ catkin create pkg aaa
$ catkin create pkg bbb
$ catkin create pkg ccc
$ catkin build
...

$ source devel/setup.bash
$ echo $CMAKE_PREFIX_PATH
/tmp/path/to/my_catkin_ws/devel

$ catkin config
```

```
------------------------------------------------------------
Profile:                     default
Extending:                   None
Workspace:                   /tmp/path/to/my_catkin_ws
...
------------------------------------------------------------
Workspace configuration appears valid.
------------------------------------------------------------
```

At this point you have a workspace which doesn't extend anything. If you realize this after the fact, you can explicitly tell it to extend another workspace. Suppose you wanted to extend a standard ROS system install like `/opt/ros/hydro`. This can be done with the `--extend` option:

```
$ catkin config --extend /opt/ros/hydro
------------------------------------------------------------
Profile:                     default
Extending:        [explicit] /opt/ros/hydro
Workspace:                   /tmp/path/to/my_catkin_ws
Source Space:       [exists] /tmp/path/to/my_catkin_ws/src
Build Space:       [missing] /tmp/path/to/my_catkin_ws/build
Devel Space:       [missing] /tmp/path/to/my_catkin_ws/devel
Install Space:     [missing] /tmp/path/to/my_catkin_ws/install
DESTDIR:                     None
------------------------------------------------------------
Isolate Develspaces:         False
Install Packages:            False
Isolate Installs:            False
------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
------------------------------------------------------------
Whitelisted Packages:        None
Blacklisted Packages:        None
------------------------------------------------------------
Workspace configuration appears valid.
------------------------------------------------------------

$ catkin clean --setup-files
$ catkin build
...

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

9.5 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:

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 pevents it from being rebuilt.

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

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.

9.6 Full Command-Line Interface

```
usage: catkin config [-h] [--workspace WORKSPACE] [--profile PROFILE]
                     [--append-args | --remove-args] [--init]
                     [--extend EXTEND_PATH | --no-extend] [--mkdirs]
                     [--whitelist PKG [PKG ...] | --no-whitelist]
                     [--blacklist PKG [PKG ...] | --no-blacklist]
                     [-s SOURCE_SPACE | --default-source-space]
                     [-b BUILD_SPACE | --default-build-space]
                     [-d DEVEL_SPACE | --default-devel-space]
                     [-i INSTALL_SPACE | --default-install-space]
                     [-x SPACE_SUFFIX] [--isolate-devel | --merge-devel]
                     [--install | --no-install]
                     [--isolate-install | --merge-install]
                     [--parallel-jobs PARALLEL_JOBS]
                     [--cmake-args ARG [ARG ...] | --no-cmake-args]
                     [--make-args ARG [ARG ...] | --no-make-args]
                     [--catkin-make-args ARG [ARG ...] |
                     --no-catkin-make-args]

This verb is used to configure a catkin workspace's configuration and layout.
Calling `catkin config` with no arguments will display the current config and
affect no changes if a config already exists for the current workspace and
profile.

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 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.

  -s SOURCE_SPACE, --source-space SOURCE_SPACE
                        The path to the source space.
  --default-source-space
                        Use the default path to the source space ("src")
  -b BUILD_SPACE, --build-space BUILD_SPACE
                        The path to the build space.
  --default-build-space
                        Use the default path to the build space ("build")
  -d DEVEL_SPACE, --devel-space DEVEL_SPACE
                        Sets the target devel space
  --default-devel-space
                        Sets the default target devel space ("devel")
  -i INSTALL_SPACE, --install-space INSTALL_SPACE
                        Sets the target install space
  --default-install-space
                        Sets the default target install space ("install")
  -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.

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

  --parallel-jobs PARALLEL_JOBS, --parallel PARALLEL_JOBS, -p PARALLEL_JOBS
                        Maximum number of packages which could be built in
                        parallel (default is cpu count)
  --cmake-args ARG [ARG ...]
                        Arbitrary arguments which are passes to CMake. It must
                        be passed after other arguments since it collects all
                        following options.
  --no-cmake-args       Pass no additional arguments to CMake.
  --make-args ARG [ARG ...]
                        Arbitrary arguments which are passes to make.It must
                        be passed after other arguments since it collects all
                        following options.
  --no-make-args        Pass no additional arguments to make (does not affect
                        --catkin-make-args).
  --catkin-make-args ARG [ARG ...]
                        Arbitrary arguments which are passes to make but only
                        for catkin packages.It must be passed after other
                        arguments since it collects all following options.
  --no-catkin-make-args
                        Pass no additional arguments to make for catkin
                        packages (does not affect --make-args).
```

CHAPTER 10

`catkin create` – Create Packages

This verb enables you to quickly create workspace elements like boilerplate Catkin packages.

10.1 Full Command-Line Interface

```
usage: catkin create [-h] {pkg} ...

Creates a catkin workspace

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

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

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

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 publically-available packages have a valid license and a valid maintainer
e-mail address.

positional arguments:
  PKGNAME               The name of the package to create. This name should be
                        completely lower-case with individual words separated
                        by undercores.

optional arguments:
  -h, --help            show this help message and exit
  --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 [DEP ...]], -c [DEP [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 [DEP ...]], -s [DEP [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 [COMP ...]]
                        One or more boost components used by the package.
```

CHAPTER 11

`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 `caktin 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.

11.1 Full Command-Line Interface

```
usage: catkin init [-h] [--workspace WORKSPACE] [--profile PROFILE] [--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.
```

CHAPTER 12

`catkin list` – List Package Info

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 directoy. It can also be used to list the packages in any other arbitrary directory.

12.1 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.

12.2 Using Unformatted Output in Shell Scripts

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

12.3 Full Command-Line Interface

```
usage: catkin list [-h] [--deps] [--depends-on [DEPENDS_ON [DEPENDS_ON ...]]]
                   [folders [folders ...]]

Lists catkin packages in the workspace or other arbitray folders.

positional arguments:
  folders               Folders in which to find packages. (default: cwd)

optional arguments:
  -h, --help            show this help message and exit
  --deps, --dependencies
                        List dependencies of each package.
  --depends-on [DEPENDS_ON [DEPENDS_ON ...]]
                        List all packages that depend on supplied argument
                        package(s).
  --quiet               Don't print out detected package warnings.
  --unformatted, -u     Print list without punctuation and additional details.
```

CHAPTER 13

`catkin profile` – Manage Profiles

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 impliclty 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.

13.1 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 by applied to a new profile called `default`.
```

To see these effects, you can run `catkin config` to write a default configuration to the workspace:

```
$ 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
Build Space:       [missing] /tmp/path/to/my_catkin_ws/build
Devel Space:       [missing] /tmp/path/to/my_catkin_ws/devel
Install Space:     [missing] /tmp/path/to/my_catkin_ws/install
```

```
DESTDIR:                     None
--------------------------------------------------------------
Isolate Develspaces:         False
Install Packages:            False
Isolate Installs:            False
--------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
--------------------------------------------------------------
Workspace configuration appears valid.
--------------------------------------------------------------
$ catkin profile list
[profile] Available profiles:
- default (active)
```

The `profile` verb now shows that the profile named "default" is avialable 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
--------------------------------------------------------------------
Profile:                     alternate
Extending:                   None
Workspace:                   /tmp/path/to/my_catkin_ws
Source Space:       [exists] /tmp/path/to/my_catkin_ws/src
Build Space:       [missing] /tmp/path/to/my_catkin_ws/build_alt
Devel Space:       [missing] /tmp/path/to/my_catkin_ws/devel_alt
Install Space:     [missing] /tmp/path/to/my_catkin_ws/install_alt
DESTDIR:                     None
--------------------------------------------------------------------
Isolate Develspaces:         False
Install Packages:            False
Isolate Installs:            False
--------------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
--------------------------------------------------------------------
Workspace configuration appears valid.
--------------------------------------------------------------------
$ catkin profile list
[profile] Available profiles:
- alternate
- default (active)
```

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`.

13.2 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)
```

13.3 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
```

13.4 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 alterate
[profile] Removed profile: alternate
[profile] Available profiles:
- alternate2 (active)
- default
```

13.5 Full Command-Line Interface

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

Manage metadata 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
```

13.5.1 catkin profile list

```
usage: catkin profile list [-h]

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

13.5.2 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
```

13.5.3 catkin profile add

```
usage: catkin profile add [-h] [-f] [--copy BASE_PROFILE | --copy-active] 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.
```

13.5.4 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.
```

13.5.5 `catkin profile remove`

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

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

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

CHAPTER 14

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.

14.1 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
```

14.2 Defining Additional Aliases

Verb aliases are defined in the `verb_aliases` subdirectory 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:

```
# ~/.config/catkin/verb_aliases/10-ros-distro-aliases.yaml
extend-sys: config --profile sys --extend /opt/ros/hydro -x _sys
extend-overlay: config --profile overlay --extend ~/ros/hydro/install -x _overlay
```

After defining these aliases, one could use them with optional additional options and build a given configuration profile.

```
$ 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.

14.3 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`.

```
# ~/.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`:

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

14.4 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:

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

CHAPTER 15

Extending the `catkin` command

The `catkin` command is setup to be easily extended using the `setuptools` notion of `entry_points`, see: http://guide.python-distribute.org/creation.html#entry-points By using the `entry_points` extensions to the `catkin` command can be made within the `catkin_tools` package or an external package. 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:

```
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 mapping 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):

```
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:

```
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:

```
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
             '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:

```
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.
    args = sys.argv[1:] if args is None else args
    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:
        args = re.sub(jobs_flags, '', ' '.join(args)).split()
        jobs_flags = jobs_flags.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:

```
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.

Note: This is the documentation for the `catkin` command-line tool and **not** the Catkin package specification documentation. For documentation on writing catkin packages, see: http://docs.ros.org/api/catkin/html/

CHAPTER 16

The `catkin` command

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.

Additionally, global options can be provided before the verb, options like `-d` for debug level verbosity or `-h` for help on the `catkin` CLI tool itself. 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.

16.1 Built-in `catkin` command 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 structrures like Catkin packages
- init – Initialize a catkin workspace
- list – Find and list information about catkin packages in a workspace
- profile – Manage different named configuration profiles

16.2 Extending the `catkin` command

If you would like to add a verb to the `catkin` command without modifying its source, please read Extending the catkin command.