

**Contents**

1. Overview
2. Overall Structure and Ordering
3. CMake Version
4. Package name
5. Finding Dependent CMake Packages
  1. What Does find\_package() Do?
  2. Why Are Catkin Packages Specified as Components?
  3. Boost
6. catkin\_package()
7. Specifying Build Targets
  1. Target Naming
  2. Custom output directory
  3. Include Paths and Library Paths
  4. Executable Targets
  5. Library Targets
  6. target\_link\_libraries
8. Messages, Services, and Action Targets
  1. Important Prerequisites/Constraints
  2. Example
9. Enabling Python module support
10. Unit Tests
11. Optional Step: Specifying Installable Targets
  1. Installing Python Executable Scripts
  2. Installing header files
  3. Installing roslaunch Files or Other Resources

# 1. Overview

The file **CMakeLists.txt** is the input to the CMake build system for building software packages. Any CMake-compliant package contains one or more CMakeLists.txt file that describe how to build the code and where to install it to. The CMakeLists.txt file used for a catkin project is a standard vanilla CMakeLists.txt file with a few additional constraints.

# 2. Overall Structure and Ordering

Your CMakeLists.txt file **MUST** follow this format otherwise your packages will not build correctly. The order in the configuration **DOES** count.

1. **Required CMake Version** (cmake\_minimum\_required)
2. **Package Name** (project())
3. **Find other CMake/Catkin packages needed for build** (find\_package())
4. **Enable Python module support** (catkin\_python\_setup())
5. **Message/Service/Action Generators**  
(add\_message\_files(), add\_service\_files(), add\_action\_files())
6. **Invoke message/service/action generation** (generate\_messages())

7. **Specify package build info export** (`catkin_package()`)
8. **Libraries/Executables to build**  
(`add_library()/add_executable()/target_link_libraries()`)
9. **Tests to build** (`catkin_add_gtest()`)
10. **Install rules** (`install()`)

## 3. CMake Version

Every catkin CMakeLists.txt file must start with the required version of CMake needed. Catkin requires version 2.8.3 or higher.

```
cmake_minimum_required(VERSION 2.8.3)
```

## 4. Package name

The next item is the name of the package which is specified by the CMake `project` function. Let us say we are making a package called *robot\_brain*.

```
project(robot_brain)
```

Note in CMake you can reference the project name anywhere later in the CMake script by using the variable `${PROJECT_NAME}` wherever needed.

## 5. Finding Dependent CMake Packages

We need to then specify which other CMake packages that need to be found to build our project using the CMake `find_package` function. There is always at least one dependency on catkin:

```
find_package(catkin REQUIRED)
```

If your project depends on other wet packages, they are automatically turned into components (in terms of CMake) of catkin. Instead of using `find_package` on those packages, if you specify them as components, it will make life easier. For example, if you use the package *nodelet*.

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

NB: You should only find\_package components for which you want build flags. You should not add runtime dependencies.

You could also do:

```
find_package(catkin REQUIRED)  
find_package(nodelet REQUIRED)
```

However, you will see that this is an inconvenient way of doing things.

## 5.1 What Does find\_package() Do?

If a package is found by CMake through `find_package`, it results in the creation of several CMake environment variables that give information about the found package. These environment variables can be utilized later in the CMake script. The environment variables describe where the packages exported header files are, where source files are, what libraries the package depends on, and the paths of those libraries. The names always follow the convention of `<PACKAGE NAME>_<PROPERTY>`:

- `<NAME>_FOUND` - Set to true if the library is found, otherwise false
- `<NAME>_INCLUDE_DIRS` or `<NAME>_INCLUDES` - The include paths exported by the package
- `<NAME>_LIBRARIES` or `<NAME>_LIBS` - The libraries exported by the package
- `<NAME>_DEFINITIONS` - ?

## 5.2 Why Are Catkin Packages Specified as Components?

Catkin packages are not really components of catkin. Rather the components feature of CMake was utilized in the design of catkin to save you significant typing time.

For catkin packages, if you `find_package` them as components of catkin, this is advantageous as a single set of environment variables is created with the `catkin_` prefix. For example, let us say you were using the package *nodelet* in your code. The recommended way of finding the package is:

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

This means that the include paths, libraries, etc exported by *nodelet* are also appended to the `catkin_` variables. For example, `catkin_INCLUDE_DIRS` contains the include paths not only for catkin but also for *nodelet* as well! This will come in handy later.

We could alternatively `find_package` *nodelet* on its own:

```
find_package(nodelet)
```

This means the *nodelet* paths, libraries and so on would not be added to `catkin_` variables.

This results in `nodelet_INCLUDE_DIRS`, `nodelet_LIBRARIES`, and so on. The same variables are also created using

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

## 5.3 Boost

If using C++ and Boost, you need to invoke `find_package()` on Boost and specify which aspects of Boost you are using as components. For example, if you wanted to use Boost threads, you would say:

```
find_package(Boost REQUIRED COMPONENTS thread)
```

## 6. catkin\_package()

**catkin\_package()** is a catkin-provided CMake macro. This is required to specify catkin-specific information to the build system which in turn is used to generate pkg-config and CMake files.

This function **must be called before** declaring any targets with `add_library()` or `add_executable()`. The function has 5 **optional** arguments:

- `INCLUDE_DIRS` - The exported include paths (i.e. `cflags`) for the package
- `LIBRARIES` - The exported libraries from the project
- `CATKIN_DEPENDS` - Other catkin projects that this project depends on
- `DEPENDS` - Non-catkin CMake projects that this project depends on. For a better understanding, see [this explanation](http://answers.ros.org/question/58498/what-is-the-purpose-of-catkin_depends/) ([http://answers.ros.org/question/58498/what-is-the-purpose-of-catkin\\_depends/](http://answers.ros.org/question/58498/what-is-the-purpose-of-catkin_depends/)).
- `CFG_EXTRAS` - Additional configuration options

Full macro documentation can be found [here](http://ros.org/doc/groovy/api/catkin/html/dev_guide/generated_cmake_api.html#catkin-package)

([http://ros.org/doc/groovy/api/catkin/html/dev\\_guide/generated\\_cmake\\_api.html#catkin-package](http://ros.org/doc/groovy/api/catkin/html/dev_guide/generated_cmake_api.html#catkin-package)).

As an example:

```
catkin_package(  
  INCLUDE_DIRS include  
  LIBRARIES ${PROJECT_NAME}  
  CATKIN_DEPENDS roscpp nodelet  
  DEPENDS eigen opencv)
```

This indicates that the folder "include" within the package folder is where exported headers go. The CMake environment variable `${PROJECT_NAME}` evaluates to whatever you passed to the `project()` function earlier, in this case it will be "robot\_brain". "roscpp" + "nodelet" are packages that need to be present to build/run this package, and "eigen" + "opencv" are system dependencies that need to be present to build/run this package.

## 7. Specifying Build Targets

Build targets can take many forms, but usually they represent one of two possibilities:

- Executable Target - programs we can run
- Library Target - libraries that can be used by executable targets at build and/or runtime

### 7.1 Target Naming

It is very important to note that **the names of build targets in catkin must be unique regardless of the folders they are built/installed to**. This is a requirement of CMake. However, unique names of targets are only necessary internally to CMake. One can have a target renamed to something else using the `set_target_properties()` function:

Example:

```
set_target_properties(rviz_image_view
                      PROPERTIES OUTPUT_NAME image_view
                      PREFIX "")
```

This will change the name of the target *rviz\_image\_view* to *image\_view* in the build and install outputs.

## 7.2 Custom output directory

While the default output directory for executables and libraries is usual set to a reasonable value it must be customized in certain cases. I.e. a library containing Python bindings must be placed in a different folder to be importable in Python:

Example:

```
set_target_properties(python_module_library
                      PROPERTIES LIBRARY_OUTPUT_DIRECTORY ${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE_PYTHON_DESTINATION})
```

## 7.3 Include Paths and Library Paths

Prior to specifying targets, you need to specify where resources can be found for said targets, specifically header files and libraries:

- Include Paths - Where can header files be found for the code (most common in C/C++) being built
- Library Paths - Where are libraries located that executable target build against?
- `include_directories(<dir1>, <dir2>, ..., <dirN>)`
- `link_directories(<dir1>, <dir2>, ..., <dirN>)`

### 7.3.1 include\_directories()

The argument to `include_directories` should be the `*_INCLUDE_DIRS` variables generated by your `find_package` calls and any additional directories that need to be included. If you are using catkin and Boost, your `include_directories()` call should look like:

```
include_directories(include ${Boost_INCLUDE_DIRS} ${catkin_INCLUDE_DIRS})
```

The first argument "include" indicates that the `include/` directory within the package is also part of the path.

### 7.3.2 link\_directories()

The CMake `link_directories()` function can be used to add additional library paths, however, this is not recommended. All catkin and CMake packages automatically have their link information added when they are `find_package`d. Simply link against the libraries in `target_link_libraries()`

Example:

```
link_directories(~/my_libs)
```

Please see [this cmake thread \(http://www.cmake.org/pipermail/cmake/2011-May/044295.html\)](http://www.cmake.org/pipermail/cmake/2011-May/044295.html) to see a detailed example of using `target_link_libraries()` over `link_directories()`.

## 7.4 Executable Targets

To specify an executable target that must be built, we must use the `add_executable()` CMake function.

```
add_executable(myProgram src/main.cpp src/some_file.cpp src/another_file.cpp)
```

This will build a target executable called *myProgram* which is built from 3 source files: `src/main.cpp`, `src/some_file.cpp` and `src/another_file.cpp`.

## 7.5 Library Targets

The `add_library()` CMake function is used to specify libraries to build. By default catkin builds shared libraries.

```
add_library(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
```

## 7.6 target\_link\_libraries

Use the `target_link_libraries()` function to specify which libraries an executable target links against. This is done typically after an `add_executable()` call. Add `${catkin_LIBRARIES}` if [ros](http://answers.ros.org/question/63656/how-to-solve-undefined-reference-to-rosinit-on-groovy/?answer=63674#post-id-63674) is not found (<http://answers.ros.org/question/63656/how-to-solve-undefined-reference-to-rosinit-on-groovy/?answer=63674#post-id-63674>).

Syntax:

```
target_link_libraries(<executableTargetName>, <lib1>, <lib2>, ... <libN>)
```

Example:

```
add_executable(foo src/foo.cpp)
add_library(moo src/moo.cpp)
target_link_libraries(foo moo) -- This links foo against libmoo.so
```

Note that there is no need to use `link_directories()` in most use cases as that information is automatically pulled in via `find_package()`.

# 8. Messages, Services, and Action Targets

Messages (.msg), services (.srv), and actions (.action) files in ROS require a special preprocessor build step before being built and used by ROS packages. The point of these macros is to generate programming language-specific files so that one can utilize messages, services, and actions in their programming language of choice. The build system will generate bindings using all available generators (e.g. `gencpp`, `genpy`, `genlisp`, etc).

There are three macros provided to handle messages, services, and actions respectively:

- `add_message_files`
- `add_service_files`
- `add_action_files`

These macros must then be followed by a call to the macro that invokes generation:

```
generate_messages()
```

## 8.1 Important Prerequisites/Constraints

- **These macros must come BEFORE the `catkin_package()` macro in order for generation to work correctly.**

```
find_package(catkin REQUIRED COMPONENTS ...)
add_message_files(...)
add_service_files(...)
add_action_files(...)
generate_messages(...)
catkin_package(...)
...
```

- **Your `catkin_package()` macro must have a `CATKIN_DEPENDS message_runtime` dependency on `message_runtime`.**

```
catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)
```

- **You must use `find_package()` for the package `message_generation`, either alone or as a component of `catkin`:**

```
find_package(catkin REQUIRED COMPONENTS message_generation)
```

- **Your `package.xml` (`/catkin/package.xml`) file must contain a build dependency on `message_generation` (`/message_generation`) and a runtime dependency on `message_runtime` (`/message_runtime`). This is not necessary if the dependencies are pulled in transitively from other packages.**
- **If you have a target which (even transitively) depends on some other target that needs messages/services/actions to be built, you need to add an explicit dependency on target `catkin_EXPORTED_TARGETS`, so that they are built in the correct order. This case applies almost always, unless your package really doesn't use any part of ROS. Unfortunately, this dependency cannot be automatically propagated. (`some_target` is the name of the target set by `add_executable()`):**

```
add_dependencies(some_target ${catkin_EXPORTED_TARGETS})
```

- **If you have a package which builds messages and/or services as well as executables that use these, you need to create an explicit dependency on the automatically-generated message target so that they are built in the correct order. (some\_target is the name of the target set by add\_executable()):**

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS})
```

- **If you your package satisfies both of the above conditions, you need to add both dependencies, i.e.:**

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
```

## 8.2 Example

If your package has two messages in a directory called "msg" named "MyMessage1.msg" and "MyMessage2.msg" and these messages depend on std\_msgs (/std\_msgs) and sensor\_msgs (/sensor\_msgs), a service in a directory called "srv" named "MyService.srv", defines executable message\_program that uses these messages and service, and executable does\_not\_use\_local\_messages\_program, which uses some parts of ROS, but not the messages/service defined in this package, then you will need the following in your CMakeLists.txt:

Toggle line numbers



```

1  # Get the information about this package's buildtime dependencies
2  find_package(catkin REQUIRED
3      COMPONENTS message_generation std_msgs sensor_msgs)
4
5  # Declare the message files to be built
6  add_message_files(FILES
7      MyMessage1.msg
8      MyMessage2.msg
9  )
10
11 # Declare the service files to be built
12 add_service_files(FILES
13     MyService.srv
14 )
15
16 # Actually generate the language-specific message and service files
17 generate_messages(DEPENDENCIES std_msgs sensor_msgs)
18
19 # Declare that this catkin package's runtime dependencies
20 catkin_package(
21     CATKIN_DEPENDS message_runtime std_msgs sensor_msgs
22 )
23
24 # define executable using MyMessage1 etc.
25 add_executable(message_program src/main.cpp)
26 add_dependencies(message_program ${${PROJECT_NAME}_EXPORTED_TARGETS} ${c
atkin_EXPORTED_TARGETS})
27
28 # define executable not using any messages/services provided by this pac
kage
29 add_executable(does_not_use_local_messages_program src/main.cpp)
30 add_dependencies(does_not_use_local_messages_program ${catkin_EXPORTED_T
ARGETS})

```

If, additionally, you want to build actionlib (/actionlib) actions, and have an action specification file called "MyAction.action" in the "action" directory, you must add `actionlib_msgs` to the list of components which are `find_package`d with catkin and add the following call before the call to `generate_messages(...)`:


```

add_action_files(FILES
    MyAction.action
)

```

Furthermore the package must have a build dependency on `actionlib_msgs`.

## 9. Enabling Python module support

If your ROS package provides some Python modules, you should create a  `setup.py` ([http://docs.ros.org/api/catkin/html/user\\_guide/setup\\_dot\\_py.html](http://docs.ros.org/api/catkin/html/user_guide/setup_dot_py.html)) file and call

```
catkin_python_setup()
```


before the call to `generate_messages()` and `catkin_package()`.

## 10. Unit Tests

There is a catkin-specific macro for handling gtest-based unit tests called `catkin_add_gtest()`.

```
if(CATKIN_ENABLE_TESTING)
  catkin_add_gtest(myUnitTest test/utest.cpp)
endif()
```

## 11. Optional Step: Specifying Installable Targets

After build time, targets are placed into the devel space of the catkin workspace. However, often we want to install targets to the system (information about installation paths can be found in  REP 122 (<http://ros.org/reps/rep-0122.html>)) so that they can be used by others or to a local folder to test a system-level installation. In other words, if you want to be able to do a "make install" of your code, you need to specify where targets should end up.

This is done using the CMake `install()` function which takes as arguments:

- TARGETS - which targets to install
- ARCHIVE DESTINATION - Static libraries and DLL (Windows) .lib stubs
- LIBRARY DESTINATION - Non-DLL shared libraries and modules
- RUNTIME DESTINATION - Executable targets and DLL (Windows) style shared libraries

Take as an example for a shared library:

```
install(TARGETS ${PROJECT_NAME}
  ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
  LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
  RUNTIME DESTINATION ${CATKIN_GLOBAL_BIN_DESTINATION}
)
```

Here is another example for an executable:

```
install(TARGETS ${PROJECT_NAME}_node
  RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Besides these standard destination some files must be installed to special folders. I.e. a library containing Python bindings must be installed to a different folder to be importable in Python:

```
install(TARGETS python_module_library
  ARCHIVE DESTINATION ${CATKIN_PACKAGE_PYTHON_DESTINATION}
  LIBRARY DESTINATION ${CATKIN_PACKAGE_PYTHON_DESTINATION}
)
```

## 11.1 Installing Python Executable Scripts

For Python code, the install rule looks different as there is no use of the `add_library()` and `add_executable()` functions so as for CMake to determine which files are targets and what type of targets they are. Instead, use the following in your CMakeLists.txt file:

```
catkin_install_python(PROGRAMS scripts/myscript
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

Detailed information about installing python scripts and modules, as well as best practices for folder layout can be found in the [catkin manual](http://wiki.ros.org/catkin)

([http://docs.ros.org/api/catkin/html/howto/format2/installing\\_python.html](http://docs.ros.org/api/catkin/html/howto/format2/installing_python.html)).

If you only install Python scripts and do not provide any modules, you need neither to create the above mentioned `setup.py` file, nor to call `catkin_python_setup()`.

## 11.2 Installing header files

Header files must also be installed to the "include" folder, This is often done by installing the files of an entire folder (optionally filtered by filename patterns and excluding SVN subfolders). This can be done with an install rule that looks as follows:

```
install(DIRECTORY include/${PROJECT_NAME}/
  DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
  PATTERN ".svn" EXCLUDE
)
```

or if the subfolder under include does not match the package name:

```
install(DIRECTORY include/
  DESTINATION ${CATKIN_GLOBAL_INCLUDE_DESTINATION}
  PATTERN ".svn" EXCLUDE
)
```

## 11.3 Installing roslaunch Files or Other Resources

Other resources like launchfiles can be installed to `${CATKIN_PACKAGE_SHARE_DESTINATION}`:

```
install(DIRECTORY launch/  
  DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}/launch  
  PATTERN ".svn" EXCLUDE)
```

Except where otherwise noted, the ROS wiki is  
licensed under the

Wiki: catkin/CMakeLists.txt (last edited 2019-07-25 23:30:47 by seanyen (/seanyen))

Creative Commons Attribution 3.0

(<http://creativecommons.org/licenses/by/3.0/>)

Brought to you by:  Open Source Robotics Foundation

(<http://www.osrfoundation.org>)