> jbohren.com < home articles tutorials projects contact

Building Modular ROS Packages

posted 2014.02.14

ros catkin cmake modularity libraries tutorial c++

Introduction

The key power of the Catkin build tool is how it makes it easier to build modular software without having to keep track of the specific build products of each package. Modularity, in this case, comes in the form of building specific functionality into *libraries* which can be used by other packages. This tutorial is meant for someone with minimal to moderate CMake experience and minimal experience with Catkin.

This tutorial begins by separating the executable code from the ROS C++ Hello World Tutorial into a library and building it with CMake and Catkin. If you are unfamilar with Catkin or CMake, this tutorial will make more sense after you have worked through the Gentle Introduction to Catkin.

The next step involves creating a second package which depends on the first package and uses the functionality defined in our library. This inter-dependency then demonstrates how to use the `catkin_package()` CMake function to declare exported targets for a package.

> ***NOTE:*** *This tutorial was written for the ROS Hydro Distribution. Assuming the commands are still accurate, if you wish to follow this tutorial with a different distribution of ROS, any time* `hydro` *is mentioned, simply replace it with the shortname for that distribution.*

Pre-Requisites

- A computer running a recent Ubuntu Linix [1] LTS (long-term support) installation
- Minimal experience with the Linux and the command-line interface
- Minimal experience with compiling C++ code

Tools Used

- Ubuntu Linux [1]
- The bash shell [2]
- C++ [3]
- CMake [4]
- Catkin [5]
- Any plain-text editor (I like vim [6]).

ROS Packages Used

- roscpp
- rosconsole
- catkin

Number of Windows Needed

- Browser for these instructions
- Window for your text editor
- Terminal to navigate the filesystem and execute build commands

Contents

- Introduction
    - Pre-Requisites
    - Tools Used
    - ROS Packages Used
    - Number of Windows Needed
- Install ROS (If not installed)
    - Add the ROS Binary Package Repository
    - Install the Base ROS Packages
- Create a Standard Catkin Workspace
- Create a Catkin Package
- Separating Functionality into a Library
    - Create the Library Code
    - Create the Node
    - Building the Node (and getting a compiler error)
    - Building the Node (and getting a linker error)
    - Building the Node (and succeeding)
- Using Libraries from Other Packages
    - Create the Second Package and Node
    - Building the Node (and getting a compiler error again)
- Exporting Package Flags to Other Packages
- Conclusion

Install ROS (If not installed)

For Ubuntu Linux, you can follow the following instructions, for other Linux platforms, see the main ROS installation instructions. As of the writing of this tutorial, ROS packages are only built with the Debian package management system [7] . This makes it easy to install on *debian-based* Linux distributions like Ubuntu.

Add the ROS Binary Package Repository

First, add the binary package repository hosted on ros.org to your sysmtem. This will allow you to locate pre-compiled ROS packages, and only needs to be done once, but is idempotent:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -cs) main" >
```

Next, get the ros.org PGP public key. This also only needs to be done once and is also idempotent.This will let you verify that your ROS packages are actually coming from ros.org and not some malicious middle-man. This is done automatically whenever you install a package from ros.org.

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

Install the Base ROS Packages

First, update the binary package index. This should be done whenever you want to make sure your system knows about the latest versions of binary packages available:

```
sudo apt-get update
```

Finally, install the base ROS packages from the ROS “Hydromedusa” distribution:

```
sudo apt-get install ros-hydro-ros-base
```

There are lots of other ROS packages available to install, but for this tutorial you only need a few of the “core” packages. To see the list of currently available binary packags, their versions, and build status, you can see the ROS debian package build status page.

Create a Standard Catkin Workspace

First, make sure your environment is set up properly. To do this, inspect the contents of the `CMAKE_PREFIX_PATH` environment variable to see which Catkin workspaces are already loaded in your environment:

```
echo $CMAKE_PREFIX_PATH
```

If this is set to `/opt/ros/hydro` then you shouldn’t have any issues. If it is set to something else (or empty) open a new shell without sourcing the setup file of another workspace, and then source the system environment setup file:

```
source /opt/ros/hydro/setup.bash
```

Create the workspace directories in an empty directory of your choosing somewhere in your filesystem, and run `catkin_make` to generate setup files for your workspace which extend those in `/opt/ros/hydro` :

```
mkdir build devel src
catkin_make
```

You should now have a standard catkin workspace with the following structure. All paths in this tutorial will be given relative to the `.` shown below:

```
.
├── build
│   └── ...
├── devel
│   └── ...
└── src
    └── CMakeLists.txt -> /opt/ros/hydro/share/catkin/cmake/toplevel.cmake
```

Finally, source one of the setup files in your newly-populated `devel` directory to load this workspace into your environment.

```
source devel/setup.bash
```

Create a Catkin Package

Create a new directory for your package:

```
mkdir src/modular_lib_pkg
```

Add bare-bones Catkin `CMakeLists.txt` and `package.xml` files to make your directory a valid package:

src/modular_lib_pkg/CMakeLists.txt

```
# Declare the version of the CMake API for forward-compatibility
cmake_minimum_required(VERSION 2.8)

# Declare the name of the CMake Project
project(modular_lib_pkg)

# Find Catkin
find_package(catkin REQUIRED)
# Declare this project as a catkin package
catkin_package()
```

src/modular_lib_pkg/package.xml

```
<package>
  <!-- Package Metadata -->
  <name>modular_lib_pkg</name>
  <maintainer email="you@example.com">Your Name</maintainer>
  <description>
    A ROS tutorial on modularity.
  </description>
  <version>0.0.0</version>
  <license>BSD</license>
  <!-- Required by Catkin -->
  <buildtool_depend>catkin</buildtool_depend>
</package>
```

Separating Functionality into a Library

The first step in making code available for use in other ROS packages is to encapsulate its functionality into a library.

On most operating systems, including Linux 8 , there are two types of libraries: static libraries and dynamic libraries. Both of these types of libraries contain compiled binary code which can be executed directly by a computer.

Static libraries (`.a` for "archive" on Linux) are linked into an executable when it is built and it becomes part of that executable. When the executable is loaded, the binary code that was copied from the static library is also loaded. Dynamic libraries (`.so` for "shared object" on Linux), however, are not copied into the executable, and instead are loaded at runtime.

This means not only are dynamically-linked executables smaller, but also the libraries that they depend on chan change internally without necessitating recompilation of the executable.

In the ROS community, dynamic libraries are most commonly used, and this is what will be built by default when using Catkin.

Create the Library Code

The first step is to create the library. Our library will encapsulate the hello-world functionality used in the ROS C++ hello-world tutorial 9 so that you can call a single function called `say_hello()` to broadcast "Hello, world!" over the `/rosout` topic.

There's nothing fundamentally different between putting C++ code in a library as opposed to an executable. What *is* required, however, is to split the code *definition* from the *declaration*. This involves creating two files: a header file and a source file.

The header file should contain only what is needed by the *compiler* of anyone who uses the library. As such, it only needs to contain function and class delcarations, and does not need to contain function definitions.

The header with the declaration of our `say_hello()` function is as follows:

modular_lib_pkg/include/modular_lib_pkg/hello_world.h

```
// Inclusion guard to prevent this header from being included multiple times
#ifndef __MODULAR_LIB_PKG_HELLO_WORLD_H
#define __MODULAR_LIB_PKG_HELLO_WORLD_H

//! Broadcast a hello-world message over ROS_INFO
void say_hello();

#endif
```

Next is the source or implementation file. This file should contain what is needed by the *linker* to connect function calls to binary code. As such, it needs to contain all of the definitions of the functions declared in the corresponding header.

The source file with the definition of `say_hello()` is as follows:

modular_lib_pkg/src/hello_world.cpp

```cpp
// Include the ROS C++ APIs
#include <ros/ros.h>

void say_hello() {
  ROS_INFO_STREAM("Hello, world!");
}
```

Now that we've written the code for the library, we can add a rule to the `CMakeLists.txt` file to actually build it. Note that just like in the ROS C++ hello-world tutorial 9 , we need to add a dependency on `roscpp` in order to use ROS. This is just like adding an executable with the `add_executable()` CMake command: instead, we use `add_library()` :

src/modular_lib_pkg/CMakeLists.txt

```cmake
# Declare the version of the CMake API for forward-compatibility
cmake_minimum_required(VERSION 2.8)

# Declare the name of the CMake Project
project(modular_lib_pkg)

# Find and get all the information about the roscpp package
find_package(roscpp REQUIRED)

# Find Catkin
find_package(catkin REQUIRED)
# Declare this project as a catkin package
catkin_package()

# Add the headers from roscpp
include_directories(${roscpp_INCLUDE_DIRS})

# Define a library target called hello_world
add_library(hello_world src/hello_world.cpp)
target_link_libraries(hello_world ${roscpp_LIBRARIES})
```

Also, now that we're using the `roscpp` package, we need to list it as a build- and run-dependency of our package:

src/modular_lib_pkg/package.xml

```xml
<package>
  <!-- Package Metadata -->
  <name>modular_lib_pkg</name>
  <maintainer email="you@example.com">Your Name</maintainer>
  <description>
    A ROS tutorial on modularity.
  </description>
  <version>0.0.0</version>
  <license>BSD</license>

  <!-- Required by Catkin -->
  <buildtool_depend>catkin</buildtool_depend>

  <!-- Package Dependencies -->
  <build_depend>roscpp</build_depend>
  <run_depend>roscpp</run_depend>
</package>
```

At this point you should be able to compile the library by running `catkin_make` from the root of your workspace and see the following output:

```
Scanning dependencies of target hello_world
[100%] Building CXX object modular_lib_pkg/CMakeFiles/hello_world.dir/src/hello_wor
Linking CXX shared library /tmp/devel/lib/libhello_world.so
[100%] Built target hello_world
```

Notice that it built the `hello_world` target into a file called `libhello_world.so` . This is the standard naming convention for dynamic libraries on Linux. Also, it built the library into the `lib` subdirectory of the develspace, so when you source one of the setup files in the `devel` directory, it will make this library available for dynamic linking at runtime.

Create the Node

Now that we have our `hello_world` library, we can write a simple program to call the `say_hello()` function in that library. This program is nearly identical to the one used in the ROS C++ hello-world Tutorial 9 , except we replace the call to `ROS_INFO` with a call to `say_hello()` and we include the header file in the previous section.

modular_lib_pkg/src/hello_world_node.cpp

```
// Include the ROS C++ APIs
#include <ros/ros.h>

// Include the declaration of our library function
#include <modular_lib_pkg/hello_world.h>

// Standard C++ entry point
int main(int argc, char** argv) {
  // Initialize ROS
  ros::init(argc, argv, "hello_world_node");
  ros::NodeHandle nh;

  // Call our library function
  say_hello();

  // Wait for SIGINT/Ctrl-C
  ros::spin();
  return 0;
}
```

To build this node, just add an appropriate `add_executable()` call to the bottom of the package's `CMakeLists.txt` :

```
add_executable(hello_world_node src/hello_world_node.cpp)
target_link_libraries(hello_world_node ${roscpp_LIBRARIES})
```

Building the Node (and getting a compiler error)

At this point, you can try to build `hello_world_node` with `catkin_make`, but you will see the following error:

```
[100%] Building CXX object modular_lib_pkg/CMakeFiles/hello_world_node.dir/src/hell
/tmp/src/modular_lib_pkg/src/hello_world_node.cpp:5:42: fatal error: modular_lib_pk
compilation terminated.
```

The compiler is complaining about `modular_lib_pkg/hello_world.h` not existing, but we *know* it exists! The problem isn't that the file doesn't exist, but rather that we haven't told the compiler where to look for it.

In the same way that we added the header search paths for `roscpp`, we also need to add our own local include directory where we put our own headers. To do so, just add the relative path to `src/modular_lib_pkg/include` to the existing `include_directories()` command in `CMakeLists.txt`:

```
include_directories(include ${roscpp_INCLUDE_DIRS})
```

Building the Node (and getting a linker error)

At this point, you can try to build `hello_world_node` with `catkin_make` again, but you will see another error:

```
[100%] Building CXX object modular_lib_pkg/CMakeFiles/hello_world_node.dir/src/hell
Linking CXX executable /tmp/foo/devel/lib/modular_lib_pkg/hello_world_node
CMakeFiles/hello_world_node.dir/src/hello_world_node.cpp.o:hello_world_node.cpp:fun
collect2: ld returned 1 exit status
```

This time, `hello_world_node.cpp` is *compiled* successfully, but the *linker* reports an error that the `say_hello()` function is undefined. The declaration was found in the `hello_world.h` header file, otherwise it wouldn't have compiled, still the definition from `hello_world.cpp` was missing.

In order to resolve this, in addition to linking against `${roscpp_LIBRARIES}`, we also link `hello_world_node` against the `hello_world` target so that its symbols are defined for the linker. This is done by adding `hello_world` to the existing `target_link_libraries()` command like the following:

```
target_link_libraries(hello_world_node ${roscpp_LIBRARIES} hello_world)
```

The following `CMakeLists.txt` file contains both this and the previous modifications:

src/modular_lib_pkg/CMakeLists.txt

```
# Declare the version of the CMake API for forward-compatibility
cmake_minimum_required(VERSION 2.8)
```

```
# Declare the name of the CMake Project
project(modular_lib_pkg)

# Find and get all the information about the roscpp package
find_package(roscpp REQUIRED)

# Find Catkin
find_package(catkin REQUIRED)
# Declare this project as a catkin package
catkin_package()

# Add the local headers and the headers from roscpp
include_directories(include ${roscpp_INCLUDE_DIRS})

# Define a library target called hello_world
add_library(hello_world src/hello_world.cpp)
target_link_libraries(hello_world ${roscpp_LIBRARIES})

# Define an executable target called hello_world_node
add_executable(hello_world_node src/hello_world_node.cpp)
target_link_libraries(hello_world_node ${roscpp_LIBRARIES} hello_world)
```

Building the Node (and succeeding)

Now you should be able to compile `hello_world_node` succesfully and then (assuming you sourced one of your workspace's setup files) you can run it with `rosrun` :

```
rosrun modular_lib_pkg hello_world_node
```

This node does the same thing as before, except now, the core functionality is implemented in a separate library, which could more easily be used by other packages.

Using Libraries from Other Packages

Now that we've created a single package with its functionality built into a library, we can create another package which also uses that functionality. In this case, we'll create *another* `hello_world_node` in another package which also links against `libhello_world.so` from `modular_lib_pkg` .

Create the Second Package and Node

First, create a package for the new node called `modular_node_pkg` :

```
mkdir src/modular_node_pkg
```

Next, add the source code for our node. This code is exactly the same as the `hello_world_node.cpp` in the `modular_lib_pkg` :

modular_node_pkg/hello_world_node.cpp

```
// Include the ROS C++ APIs
#include <ros/ros.h>

// Include the declaration of our library function
#include <modular_lib_pkg/hello_world.h>

// Standard C++ entry point
int main(int argc, char** argv) {
  // Initialize ROS
  ros::init(argc, argv, "hello_world_node");
  ros::NodeHandle nh;

  // Call our library function
  say_hello();

  // Wait for SIGINT/Ctrl-C
  ros::spin();
  return 0;
}
```

Then add the following `CMakeLists.txt` and `package.xml` files to the new package. Note that now that we're using the `modular_lib_pkg` just like we're using the `roscpp` package, we need to find its headers and libraries just like we do with `roscpp` :

src/modular_node_pkg/CMakeLists.txt

```
# Declare the version of the CMake API for forward-compatibility
cmake_minimum_required(VERSION 2.8)

# Declare the name of the CMake Project
project(modular_node_pkg)

# Find and get all the information about the roscpp package
find_package(roscpp REQUIRED)

# Find and get all the information about the modular_lib_pkg package
find_package(modular_lib_pkg REQUIRED)

# Find Catkin
find_package(catkin REQUIRED)
# Declare this project as a catkin package
catkin_package()

# Add the headers from roscpp
include_directories(${roscpp_INCLUDE_DIRS} ${modular_lib_pkg_INCLUDE_DIRS})

# Define an executable  target called hello_world_node
add_executable(hello_world_node2 hello_world_node.cpp)
target_link_libraries(hello_world_node2 ${roscpp_LIBRARIES} ${modular_lib_pkg_LIBRAI
```

> ***NOTE:*** *Goofy or not, the way that Catkin works, it combines all of your packages into a single CMake project. This means that each package* ***must have unique target names****. Otherwise the world will implode and unhappiness will descend upon the land. If you don't want to have this constraint, you can*

use `catkin_make_isolated` which will build each package in isolation, but will be slower.

`src/modular_lib_pkg/package.xml`

```
<package>
  <!-- Package Metadata -->
  <name>modular_node_pkg</name>
  <maintainer email="you@example.com">Your Name</maintainer>
  <description>
    A ROS tutorial on modularity.
  </description>
  <version>0.0.0</version>
  <license>BSD</license>

  <!-- Required by Catkin -->
  <buildtool_depend>catkin</buildtool_depend>

  <!-- Package Dependencies -->
  <build_depend>roscpp</build_depend>
  <build_depend>modular_lib_pkg</build_depend>

  <run_depend>roscpp</run_depend>
  <run_depend>modular_lib_pkg</run_depend>
</package>
```

After creating these files, your workspace should look like the following:

```
.
├── build
│   └── ...
├── devel
│   └── ...
└── src
    ├── CMakeLists.txt -> /opt/ros/hydro/share/catkin/cmake/toplevel.cmake
    ├── modular_lib_pkg
    │   ├── CMakeLists.txt
    │   ├── include
    │   │   └── modular_lib_pkg
    │   │       └── hello_world.h
    │   ├── package.xml
    │   └── src
    │       ├── hello_world.cpp
    │       └── hello_world_node.cpp
    └── modular_node_pkg
        ├── CMakeLists.txt
        ├── hello_world_node.cpp
        └── package.xml
```

Building the Node (and getting a compiler error again)

If you try to build yor workspace by running `catkin_make` at this point, you will get the same compiler error as before, but this time with the new node!

```
[100%] Building CXX object modular_node_pkg/CMakeFiles/hello_world_node2.dir/hello_
/tmp/foo/src/modular_node_pkg/hello_world_node.cpp:5:41: fatal error: modular_lib_p
compilation terminated.
```

Despite the fact that you included `${modular_lib_pkg_INCLUDE_DIRS}` in the `include_directories()` CMake function, it still couldn't find the header. This is because this sort of information needs to be *exported* by the other package.

With the current workspace, not only will `${modular_lib_pkg_INCLUDE_DIRS}` be empty, but also `${modular_lib_pkg_LIBRARIES}` will also be empty.

Exporting Package Flags to Other Packages

In the previous secion, our second package, `modular_node_pkg`, was unable to get the compilation or linker flags from the first package, `modular_lib_pkg`. This is because the flags weren't exported by `modular_lib_pkg`. With Catkin, exporting such information is done with the `catkin_package()` command in the `CMakeLists.txt` file, and in the case of `modular_lib_pkg`, we didn't pass it any arguments:

```
catkin_package()
```

This function can be left empty if we don't need to export anything, but if we do, there are several optional arguments 10 and the following are most commonly used:

- **`INCLUDE_DIRS`** One or more header directories that should be made available to other packages. These directories are relative to the path of the given `CMakeLists.txt` file.
- **`LIBRARIES`** One or more libraries that should be made available to other packages. These are the *target names* of the libraries.
- **`CATKIN_DEPENDS`** One or more names of Catkin packages whose build flags should be passed transitively to any package which depends on this one. This will cause dependent packages to automatically call `find_package()` on each of these names.
- **`DEPENDS`** One or more names of packages whose build flags should be passed transitively to any package which depends on this one. If a name like `foo` is given here, then Catkin will add whatever the contents of the `${foo_INCLUDE_DIRS}` and `${foo_LIBRARIES}` variables will be exported as part of this package's include directories and libraries, respectively.

In our case, we want to export both a local include directory and a library, so we modify the `catkin_package()` call in the `modular_lib_pkg` `CMakeLists.txt` to export the flags for our include directory and library.

Additionally, we should declare that anyone depending on this package should also use build flags from the `roscpp` package. This is important either if we link our library against libraries from the `roscpp` package or if any of our exported header files `#include` headers from `roscpp`.

```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES hello_world
  CATKIN_DEPENDS roscpp
  )
```

> ***NOTE:*** *In this specific case, leaving out the `CATKIN_DEPENDS` on `roscpp` won't cause any problems, but this is only because it is unlikely that someone would try to build a ROS C++ node without depending on `roscpp` directly. A motivating example will be shown in the next section.*

The complete `CMakeLists.txt` for `modular_lib_pkg` is as follows:

src/modular_lib_pkg/CMakeLists.txt

```
# Declare the version of the CMake API for forward-compatibility
cmake_minimum_required(VERSION 2.8)

# Declare the name of the CMake Project
project(modular_lib_pkg)

# Find and get all the information about the roscpp package
find_package(roscpp REQUIRED)

# Find Catkin
find_package(catkin REQUIRED)
# Declare this project as a catkin package and export the necessary build flags
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES hello_world
  CATKIN_DEPENDS roscpp
  )

# Add the local headers and the headers from roscpp
include_directories(include ${roscpp_INCLUDE_DIRS})

# Define a library target called hello_world
add_library(hello_world src/hello_world.cpp)
target_link_libraries(hello_world ${roscpp_LIBRARIES})

# Define an executable target called hello_world_node
add_executable(hello_world_node src/hello_world_node.cpp)
target_link_libraries(hello_world_node ${roscpp_LIBRARIES} hello_world)
```

You can now build the workspace again with `catkin_make`, but this time it should succeed:

```
[ 33%] Building CXX object modular_lib_pkg/CMakeFiles/hello_world.dir/src/hello_wor
Linking CXX shared library /tmp/devel/lib/libhello_world.so
[ 33%] Built target hello_world
[ 66%] Building CXX object modular_lib_pkg/CMakeFiles/hello_world_node.dir/src/hell
Linking CXX executable /tmp/devel/lib/modular_lib_pkg/hello_world_node
[ 66%] Built target hello_world_node
[100%] Building CXX object modular_node_pkg/CMakeFiles/hello_world_node2.dir/hello_
Linking CXX executable /tmp/devel/lib/modular_node_pkg/hello_world_node2
[100%] Built target hello_world_node2
```

And finally, (assuming you still have your workspace environment set up), you can run `hello_world_node2` :

```
rosrun modular_node_pkg hello_world_node2
```

Conclusion

This tutorial has demonstrated some of the basic features of Catkin which enable packags to share code in a modular way. In future code that you write, you now know how to design your packages in such a way that makes it easy for your code to be re-used, simply by partitioning your code between libraries and executables and by declaring the necessary build flags so that others only need to know your API and the name of your package to depend on it!

references

1. The Ubuntu Linux Distribution ↩ ↩2
2. The Bourne Again Shell ↩
3. The C++ Programming Language ↩
4. The CMake Cross-Platform Buildsystem ↩
5. The Catkin Build Tool ↩
6. The VIM Text Editor ↩
7. The Debian Package Management System ↩
8. Static, Shared Dynamic and Loadable Linux Libraries ↩
9. ROS C++ Hello World (The Simplest ROS Tutorial) ↩ ↩2 ↩3
10. `catkin_package()` API Documentation ↩