Slate Robotics (/)

(/) Back to blogs (/blog)

How to implement ros_control on a custom robot

Taking advantage of one of the most foundational ROS packages

By Zach Allen, published on 30/03/2018

Introduction

ROS has a ton of built-in packages to make it so that you don't have to do math. When I was first studying robotics, I was blown away by how much math was required to just control a two-wheel differential drive robot. Kinematics, PID control, and everything in-between meant that I spent a lot of time solving problems that were already solved. This is the beauty of ROS.

ros_control is another one of those packages that allows you to focus your time on problems specific to your robot--the problems that haven't solved. Well, sort of. Unfortunately, I found that the tutorials on ros.org (http://wiki.ros.org/ros_control/Tutorials) for ros_control were not terribly decipherable. Only after I had spent a week going through source code could I really look back on these and understand what they meant.

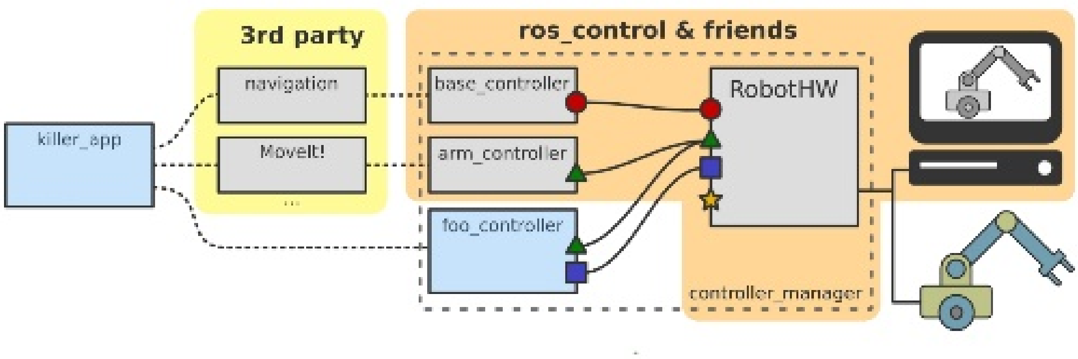
This is a blog post to help those of you who might be struggling like I was. The example code used throughout the tutorial is boilerplate code that won't build. It's to give you a better idea for how to setup your own robot-specific package. I'll also assume a basic understanding of ROS; cre your own packages, rostopics, rosservice, roscpp, etc

Overview

If you're building a custom robot, using ros_control gives you three basic advantages:

- Standardization of APIs for controllers and hardware interfaces, making it easier to integrate with other packages, such as MoveIt
- Built-in control loop feedback mechanisms like PID controllers
- Easily enforce joint limits at a low level in the stack

The most helpful image I've found of understanding how it works is this:



Adolfo Rodriguez Tsouroukdissian, "ros_control: An overview", ROSCon 2014

Other packages send certain high-level desired goals to the controllers, and ros_control utilizes its controllers to work with the hardware to move joints based on those specifications.

Let's say you want to move a joint from 0 radians to 1.57 radians. The robot's state before you do anything is 0 radians. You (or a high level package) sends 1.57 to that joint's pre-specified Joint Position Controller as the desired goal. The Joint Position Controller works with an even lower level package to actually send current to the actuators. The Joint Position Controller has its own PID that continuously reads the joint's state and adjusts the current sent to the motors based on the error between the current state and the goal state.

ros_control handles two things from that process:

- receiving the goals (effort, position, velocity, trajectory, etc.)
- running the PID controllers

ros_control doesn't know or handle:

- implementing hardware control (sending current to motors)
- reading hardware state

You have to control the hardware (actuators, servos, motors, etc.) using your own code by listening to what ros_control says it should do. You also must p te robot and its joints if you want to control position, velo

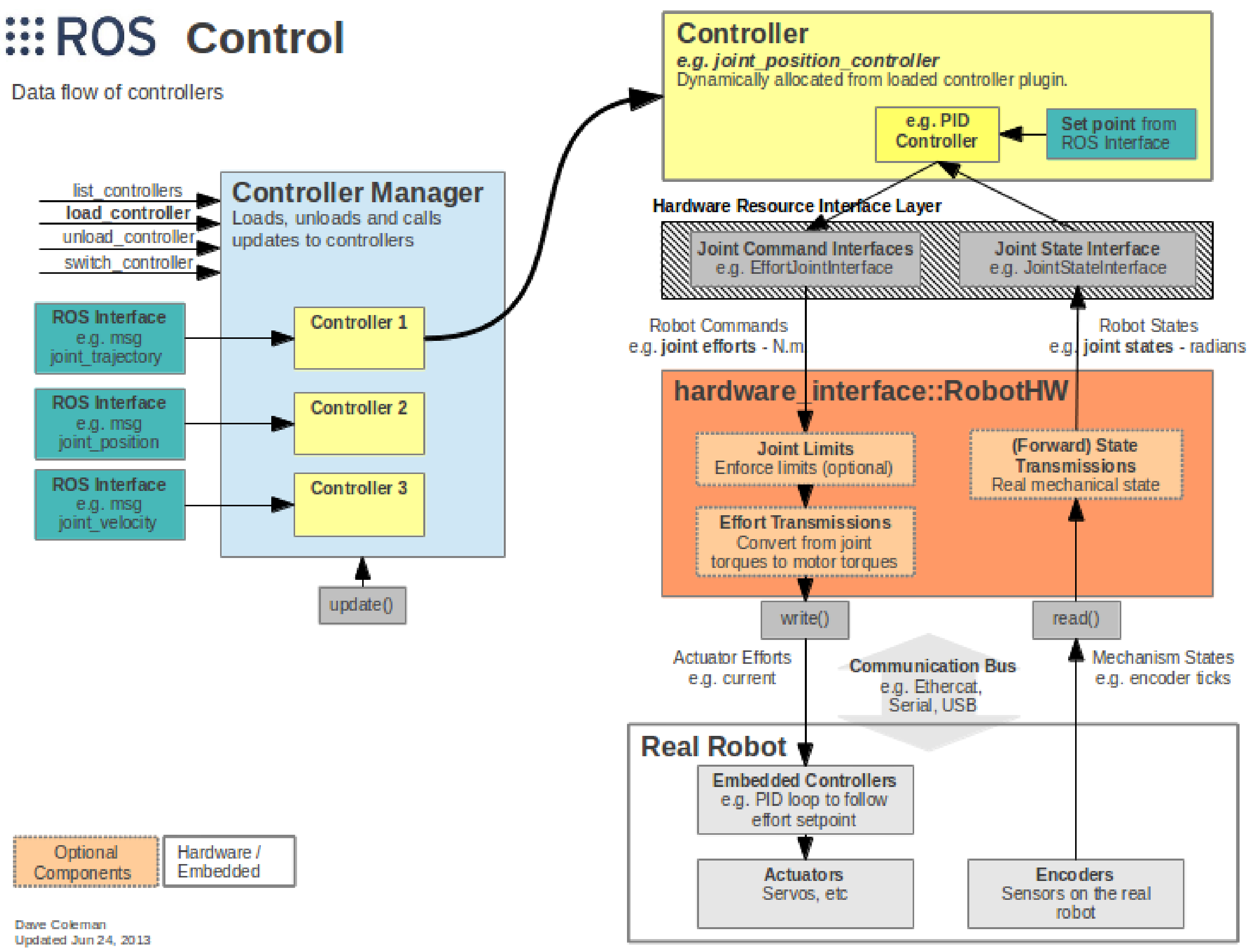
Installation

On Ubuntu, you can install `ros_control` from debian packages. Depending on your ROS distribution, you will need to change the below command. See the ros_control wiki (http://wiki.ros.org/ros_control) for a list of supported distributions.

```
sudo apt-get install ros-indigo-ros-control ros-indigo-ros-controllers
```

Setting up your package

One of the most confusing things for me was that ros_control isn't really it's own node. Well, the controller manager is. But it's really just a C++ library. The thing you actually build to get it to work is the middleware between ros_control and your hardware. This is called hardware_interface.



So, in order to get ros_control on our robot, we'll need to create our own package called something like ROBOT_hardware_intereface. At Slate Robotics, we built tr1_hardware_interface (https://github.com/SlateRobotics/tr1_hardware_interface) to control the TR1 (https://slaterobots.com/tr1) with ros_control, which is the basis for most of this tutorial. I recommend you check out that repository if you're confused at any point.

Package directory

The directory for our ROBOT_hardware_interface package is simple and follows the standard guidelines. We'll need a `config/` directory where you have `.yaml` files to define controllers, hardware (the joints available to the controllers), and joint limits. We'll make some launch files in `launch/` to make it easy to fire-up certain controllers. There are only two cpp files we'll need to create in our `src/` directory, and two header files in our `include/ROBOT_hardware_interface/` directory. Here's what that looks like:

```
launch/
  ROBOT_position_controllers.launch
config/
  controllers.yaml
  hardware.yaml
  joint_limits.yaml
include/
  ROBOT_hardware_interface/
    ROBOT_hardware.h
    ROBOT_hardware_interface.h
src/
  ROBOT_hardware_interface.cpp
  ROBOT_hardware_interface_node.cpp
CMakeLists.txt
package.xml
```

launch/ROBOT_position_controllers.launch

We'll start by looking at the launch file and work our way through its dependencies.

```
<launch>
    <rosparam file="$(find ROBOT_hardware_interface)/config/hardware.yaml" command="loa
d"/>
    <rosparam file="$(find ROBOT_hardware_interface)/config/controllers.yaml" command
="load"/>
    <rosparam file="$(find ROBOT_hardware_interface)/config/joint_limits.yaml" command
="load"/>
    <node name="ROBOT_hardware_interface" pkg="ROBOT_hardware_interface" type="ROBOT_ha
rdware_interface_node" output="screen"/>
    <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="fa
lse" output="screen" ns="/"
        args="
            /ROBOT/controller/state
            /ROBOT/controller/position/YOUR_ROBOT_JOINT
            /ROBOT/controller/position/YOUR_OTHER_ROBOT_JOINT
        "/>
</launch>
```

All we're doing here is:

- loading rosparams hardware, controllers, and joint_limits
- firing up our ROBOT_hardware_interface node
- firing up controller_spawner node, which is part of ros_control stack

Take note of "args" in the controller_spawner node. We're specifying which controllers from controllers.yaml that we want to use. Speaking of whic ruct

config/controllers.yaml

`controllers.yaml` is where we define the overall available controllers to the controller_spawner node in our `.launch` file. Take notice of the fact that the args listed above pulls from the controllers defined here. You can define many more controllers in this file, but the `.launch` file ultimately gets to pick which are spawned.

```
ROBOT:
  controller:
    state:
      type: joint_state_controller/JointStateController
      publish_rate: 50
    position:
      YOUR_ROBOT_JOINT:
        type: effort_controllers/JointPositionController
        joint: YOUR_ROBOT_JOINT
        pid: {p: 10.0, i: 0.0, d: 1.0}
      YOUR_OTHER_ROBOT_JOINT:
        type: effort_controllers/JointPositionController
        joint: YOUR_ROBOT_JOINT
        pid: {p: 5.0, i: 2.0, d: 1.0}
```

The `type` element can be anything from the ros_controllers (https://github.com/ros-controls/ros_controllers) repository. Something confusing about this is that a controller is seemingly both an effort and position controller at the same time. For instance, there's both an `effort_controllers/JointPositionController` and `position_controllers/JointPositionController` option.

The difference is what commands get passed to your hardware. `effort_controllers/` means that the controller is using an effort command (the amount of current to the motors, in most cases) to control position, and `position_controllers/` means that the controller is using position itself to control position, which might make sense for controlling servos, for instance.

Further, each controller will require its own parameters. You can view the comments of the header files in the ros_controllers (https://github.com/ros-controls/ros_controllers) repository to see what's required. `JointPositionController`, for example, requires you define a `joint` and `pid` parameter. As you can see, the joint is something in rosparam, which came from our `config/hardware.yaml` file.

config/hardware.yaml

This is probably the simplest of the configuration files.

```
ROBOT:
  hardware_interface:
     loop_hz: 10 # hz
     joints:
       - YOUR_ROBOT_JOINT
       - YOUR_OTHER_ROBOT_JOINT
```

We have defined `loop_hz`, which is a parameter we've set here for convenience. We'll use it in our `src/ROBOT_hardware_interface.cpp` file. And of course, the joints are defined here as well.

You can list as many joints as your robot has to offer h mes these joints as we loop through to control them in the y advantage when actuating the motors.

config/joint_limits.yaml

The required parameters and schema for the joint_limits can be found at ros_control/joint_limits_interface (https://github.com/ros-controls/ros_control/tree/indigo-devel/joint_limits_interface). I've inserted some sample data below, but you will need to change the values depending on the specification of your robot.

```
joint_limits:
  YOUR_ROBOT_JOINT:
    has_position_limits: true
    min_position: -1.0
    max_position: 1.5708
    has_velocity_limits: true
    max_velocity: 2.0
    has_acceleration_limits: true
    max_acceleration: 5.0
    has_jerk_limits: true
    max_jerk: 100.0
    has_effort_limits: true
    max_effort: 1.0
  YOUR_OTHER_ROBOT_JOINT:
    has_position_limits: true
    min_position: -1.5708
    max_position: 3.1415
    has_velocity_limits: true
    max_velocity: 2.0
    has_acceleration_limits: true
    max_acceleration: 5.0
    has_jerk_limits: true
    max_jerk: 100.0
    has_effort_limits: true
    max_effort: 1.0
```

include/ROBOT_hardware_interface/ROBOT_hardware.h

The `ROBOT_hardware` class will be a base class for our `ROBOT_hardware_interface` class that we will tackle in the next subsection. Here, we will store many of the interfaces to `ros_control` as well as variables for joint information that get initialized by the class deriving `ROBOT_hardware`. This is a fairly abstract class that can be shared by various robots or if you wish to make multiple `ROBOT_hardware_interface` classes. Also note that there is no `.cpp` file corresponding with this header file.

```
#ifndef ROS_CONTROL__ROBOT_HARDWARE_H
#define ROS_CONTROL__ROBOT_HARDWARE_H

#include <hardware_interface/joint_state_interface.h>
#include <hardware_interface/joint_command_interface.h>
#include <hardware_interface/robot_hw.h>
#include <joint_limits_interface/joint_limits.h>
#include <joint_limits_interface/joint_limits_interface.h>
#include <joint_limits_interface/joint_limits_rosparam.h>
#include <joint_limits_interface/joint_limits_urdf.h>
#include <controller_manager/controller_manager.h>
#include <boost/scoped_ptr.hpp>
#include <ros/ros.h>

namespace ROBOT_hardware_interface
{
    /// \brief Hardware interface for a robot
    class ROBOTHardware : public hardware_interface::RobotHW
    {
        protected:
            // Interfaces
            hardware_interface::JointStateInterface joint_state_interface_;
            hardware_interface::PositionJointInterface position_joint_interface_;
            hardware_interface::VelocityJointInterface velocity_joint_interface_;
            hardware_interface::EffortJointInterface effort_joint_interface_;

            joint_limits_interface::EffortJointSaturationInterface effort_joint_satur
```

include/ROBOT_hardware_interface/ROBOT_hardware_interface.h

`ROBOT_hardware_interface.h` defines our list of available variables and class members. In the next file, we'll define the init, update, read, and write methods, which will make up the bulk of the work for getting ros_control installed.

```
#ifndef ROS_CONTROL__ROBOT_HARDWARE_INTERFACE_H
#define ROS_CONTROL__ROBOT_HARDWARE_INTERFACE_H

#include <hardware_interface/joint_state_interface.h>
#include <hardware_interface/joint_command_interface.h>
#include <hardware_interface/robot_hw.h>
#include <joint_limits_interface/joint_limits_interface.h>
#include <joint_limits_interface/joint_limits.h>
#include <joint_limits_interface/joint_limits_urdf.h>
#include <joint_limits_interface/joint_limits_rosparam.h>
#include <controller_manager/controller_manager.h>
#include <boost/scoped_ptr.hpp>
#include <ros/ros.h>
#include <ROBOTcpp/ROBOT.h>
#include <ROBOT_hardware_interface/ROBOT_hardware.h>

using namespace hardware_interface;
using joint_limits_interface::JointLimits;
using joint_limits_interface::SoftJointLimits;
using joint_limits_interface::PositionJointSoftLimitsHandle;
using joint_limits_interface::PositionJointSoftLimitsInterface;


namespace ROBOT_hardware_interface
{
    static const double POSITION_STEP_FACTOR = 10;
    static const double VELOCITY_STEP_FACTOR = 10;
```

src/ROBOT_hardware_interface.cpp

Here, we define the classes that were stated in `ROBOT_hardware_interface.h` . Our goals are to:

1. init(): Register joint-specific handles to each type of controller interfaces (joint state, position, effort, etc.)
2. read(): Read joint positions from the robot's hardware and set the `joint_position_` array with that data.
3. write(): Actuate the robot's joints using the `joint_effort_command_` variable. Based on our position controllers defined in `config/controllers.yaml` , this will be set by `ros_control` by using the desired joint position (`joint_position_command_`), the error between `joint_position_` and `joint_position_command_` , and the PID parameter for that controller in `config/controllers.yaml` --all of which gets calculated when we call `update()` .
4. update(): simply read joint state, call `update()` on our controller manager, and write/actuate the joints based on what's calculated.

```
#include <sstream>
#include <ROBOT_hardware_interface/ROBOT_hardware_interface.h>
#include <joint_limits_interface/joint_limits_interface.h>
#include <joint_limits_interface/joint_limits.h>
#include <joint_limits_interface/joint_limits_urdf.h>
#include <joint_limits_interface/joint_limits_rosparam.h>
#include <ROBOTcpp/ROBOT.h>

using namespace hardware_interface;
using joint_limits_interface::JointLimits;
using joint_limits_interface::SoftJointLimits;
using joint_limits_interface::PositionJointSoftLimitsHandle;
using joint_limits_interface::PositionJointSoftLimitsInterface;

namespace ROBOT_hardware_interface
{
    ROBOTHardwareInterface::ROBOTHardwareInterface(ros::NodeHandle& nh) : nh_(nh) {
        init();
        controller_manager_.reset(new controller_manager::ControllerManager(this, nh
_));
        nh_.param("/ROBOT/hardware_interface/loop_hz", loop_hz_, 0.1);
        ros::Duration update_freq = ros::Duration(1.0/loop_hz_);
        non_realtime_loop_ = nh_.createTimer(update_freq, &TR1HardwareInterface::upda
te, this);
    }

    ROBOTHardwareInterface::~ROBOTHardwareInterface() {
```

src/ROBOT_hardware_interface_node.cpp

Finally, we'll wrap everything up in a nice little node that can be executed by our `.launch` file. The only thing of note here is that we're setting up the ROS node, a node handle, and passing that node handle to our ROBOT_hardware_interface class.

```
#include <ROBOT_hardware_interface/ROBOT_hardware_interface.h>
int main(int argc, char** argv)
{
    ros::init(argc, argv, "ROBOT_hardware_interface");
    ros::NodeHandle nh;
    ros::AsyncSpinner spinner(1);
    spinner.start();
    ROBOT_hardware_interface::ROBOTHardwareInterface ROBOT(nh);
    ros::spin();
    return 0;
}
```

A word about ROBOTcpp

You may have noticed in a few of the files references to ROBOTcpp or a ROBOT object, such as this line in `src/ROBOT_hardware_interface.cpp`:

```
ROBOT.getJoint(joint_names_[i]).actuate(joint_effort_command_[i]);
```


What we've done in our implementation of `ros_control` on the TR1 is create a separate C++ library for simple hardware functionality called tr1cpp (https://github.com/SlateRobotics/tr1cpp). This allows us to decouple the `ros_control` `TR1_hardware_interface` package from actual implementation of sending current to motors and reading bits of data from joint sensors.

Ultimately, something is sending a PWM signal to a motor driver that sends current to the motor. So, when you call `write()` and `read()` in your `src/ROBOT_hardware_interface.cpp` file, you need to connect the dots between the `ros_control` output/input and your actual hardware--something that synchronously executes the `joint_effort_command_` variable on the hardware.

`tr1cpp` has an `actuate()` method on the `joint` class, so we just pass that value to that method. Of course, you can set this up anyway you like. This is just what we found to be a solution.

Conclusion

This guide has shown you an example for how to setup `ros_control` on your custom robot. `ros_control` was one of those things that took a long time to figure out, but we've been so happy with how easily we can spin up high-level packages on the TR1 (https://slaterobots.com/tr1) now that we have it. Anybody interested in building complex robots really need to think about configuring `ros_control`. If that's you, we hope this has been a helpful resource on that journey.

Best wishes!

Join our newsletter!

Stay up to date with the latest TR1 and Slate Robotics developments.

(/)

Slate Robotics (/)

Human-sized robots for hackers

TR1

Overview (/tr1)
Specs (/tr1/specs)
Buy (/shop/tr1)

Other

T-Shirts (https://teespring.com/stores/slate-robotics)

COMPANY

About Us (/about)
Q&A (/questions)
Careers (/careers)
Blog (/blog)
Privacy Policy (/privacy-policy)
Terms & Conditions (/terms-and-conditions)
Sales Policies (/sales-policies)

SOCIAL

Github (https://www.github.com/SlateRobotics/)
Facebook (https://www.facebook.com/SlateRobotics/)
Twitter (https://www.twitter.com/SlateRobotics/)
LinkedIn (https://www.linkedin.com/company/24790837/)
Instagram (https://www.instagram.com/SlateRobotics/)
YouTube (https://www.youtube.com/channel/UC1xko_FNwN6H8PL3MLtJ1UQ)

CONTACT

zach@slaterobots.com (mailto:zach@slaterobots.com)
(417) 849-3612 (tel:+14178493612)

Slate Robotics, Inc.
210 W Sunshine St., Suite C
Springfield, MO 65807

GET SLATE ROBOTICS UPDATES

Stay up to date with all product and company information via our email updates