HSL Crazyflie Documentation

David Fridovich-Keil

November 2017

Contents

1 Introduction

Welcome to the Hybrid Systems Lab's Crazyflie documentation! This manual includes all the information you'll need to download, build, use, and improve the Crazyflie code base. Most of the rest of this manual is structured as a quick start guide, with step-by-step instructions to follow; here though we'll indulge in a few preliminaries that are intended to help users figure out some of the philosophy behind the way the code base is structured.

1.1 ROS — Robot Operating System

ROS, or Robot Operating System, is the glue that holds this repository together. For a detailed set of tutorials, please refer to the online documentation. The key functionality that we will use is the ROS *computation graph*, in which "nodes" are dedicated to performing particular computational tasks, the results of which are "published" as "messages" on "topics," to which other nodes may "subscribe." Nodes can also provide "services" to each other, which effectively lets nodes make calls to functions that live in other nodes. All of this happens asynchronously since each node is literally a separate computational process, so timing can be important (more on that later).

ROS is an open source project that keeps coming out with new versions every year. For stability, this repository is built with ROS Indigo/Jade on Ubuntu 14.04. However, as you will see the code only relies on core ROS functionality which really should not change much in newer versions of ROS.

1.2 Language

This code base is almost entirely written in C++. If you are not familiar with C++, there are lots of great online resources for learning the basics and it might be a good idea to consult with them before attempting to add new functionality. However, one of the cool things about the way ROS works and the way this code base is constructed is that many of the key pieces which you may want to improve and extend can be written in Python, since the interface with other nodes is handled by ROS and does not require direct function calls or inheritance or anything like that. Concretely, for example, if you wanted to write your own state feedback controller in Python you could simply write a node in Python that subscribes to a topic where the state estimator is publishing new state messages, and publishes control messages on a designated topic.

1.3 Style

As with any large programming project, style is very important. I'll give a list of general style pointers here and throughout this manual, but I strongly encourage you to take a look at any of the following resources if you have specific questions: the ROS C++ style guide, the Google C++ style guide, and the Google Python style guide.

- Use *inheritance* and *encapsulation* whenever possible. Inheritance is a good idea especially for classes you might need multiple versions of. For example, it makes sense for different dynamics models (each of which is its own class) to inherit from a single parent class that specifies a single unified interface. Encapsulation is the other key to effective abstraction, since it lets users of a particular class not have to worry about how particular functions are implemented or data are stored.

- *Modularity* is the key to effective software design. If you have a big complicated class or even just a function with lots of moving parts, chances are you can make your (and everyone else's) life easier by breaking it up into multiple different pieces that do one or maybe two things, and do them reliably.

- *Unit testing* is another important way to ensure that modules behave reliably in a large project. The main idea behind writing good unit tests is to test the key functionality of each class in isolation to make sure it behaves as expected.
- Comments are what let yourself and other users figure out what's going on without having to parse the code itself. Comment early and comment often.
- Classes, functions, and variables should all be named clearly and consistently. For example, throughout this repository class and function names are `CamelCased`, but variable names have `under_scores`. In C++, member variables end in an underscore, and in Python, member variables begin with an underscore.
- In compiled languages like C++, it's good to get in the habit of using the `const` specifier whenever possible, both for variables and for a class' member functions.

2 Organization

The repository is organized as a single ROS workspace that contains many different packages, as shown below. Everything is based on the original SDK (that lives in the `crazyflie_ros` metapackage), which itself has not been changed. There are a number of new packages:

- The core functionality of state estimation and control are contained in the `crazyflie_state_estimator` and `crazyflie_lqr` packages, respectively.
- The `crazyflie_simulator` package provides a simple physics simulator.
- The `crazyflie_control_merger` package converts control signals (from potentially multiple controllers) to the proper form for sending over the Crazyflie's radio.
- The `crazyflie_takeoff` node provides custom takeoff and landing services.
- All of these packages rely on a set of custom ROS messages that are defined in the `crazyflie_msgs` package, and many also take advantage of helper utilities in the `crazyflie_utils` package.

```
crazyflie_clean
  LICENSE
  README.md
  ros                             # ROS workspace
    build
    devel
      setup.bash                  # Tells ROS where to find our  packages.
    src
      crazyflie_control_merger    # Package for least restrictive control.
      crazyflie_examples          # Package with software/hardware examples.
      crazyflie_lqr               # Package for LQR controllers.
      crazyflie_msgs              # Package that defines custom messages.
      crazyflie_ros               # Metapackage containing original SDK.
        crazyflie
        crazyflie_controller
        crazyflie_cpp
        crazyflie_demo
        crazyflie_description
        crazyflie_driver
        crazyflie_tools
      crazyflie_simulator        # Package for simple physics simulation.
      crazyflie_state_estimator  # Package containing all state estimators.
      crazyflie_takeoff          # Package providing takeoff/landing services.
      crazyflie_utils            # Package with random utilities.
```

Figure 1 shows the network of ROS nodes and topics that are in operation during the software demo presented in Sec. 4. Each oval represents a node; nodes are connected to rectangular topics by arrows, which represent messages; and sometimes groups of topics and/or nodes share the same namespace, which is represented by a large rectangle containing them.

Let's parse Fig. 1 a little more closely, starting from the node marked `/simulator`, which comes from the `crazyflie_simulator` package. The `/simulator` listens for control messages on the `/control/final` topic that are being published by the `/takeoff_control_filter` mode. The simulator then updates its internal state and broadcasts its position and orientation to the `/tf`

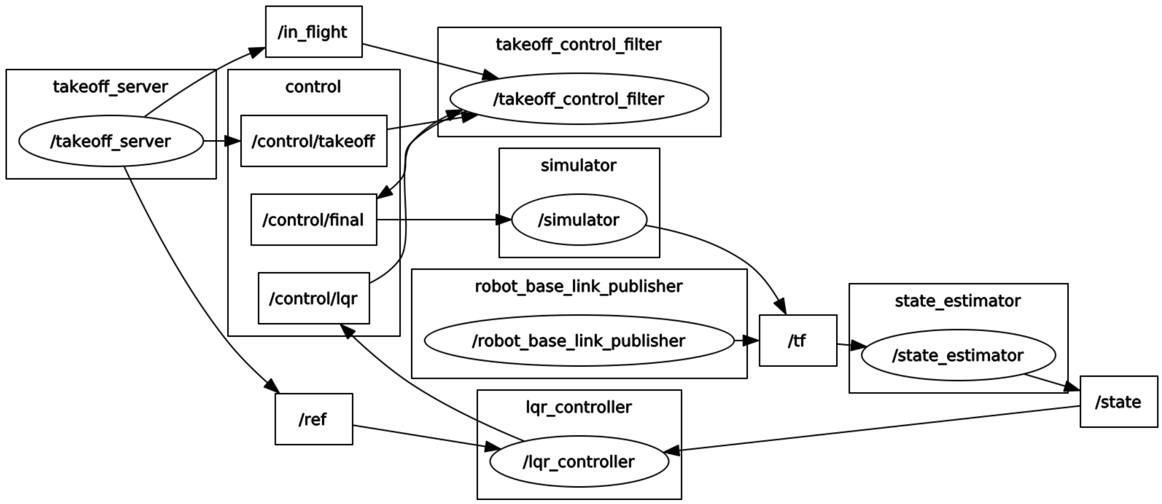

Figure 1: Network of ROS nodes and topics in operation during the software demo in Sec. 4.

topic.[1] Moving along, the state estimator reads from `/tf` and publishes to a topic called `/state`. The `/lqr_controller` node subscribes to the `/state` topic, whose messages contain position, velocity, and yaw information. In general, we could have more than one controller operating in different state spaces, each of which would require its own state estimator. The `/lqr_controller` publishes messages to the `/control/lqr` topic, which are then either passed through or filtered out by the `/takeoff_control_filter` node, depending on whether the vehicle is in flight or not.

The purpose of the separate `/control/takeoff` topic is to provide an overriding control signal during takeoff and landing. When the user sends the command to takeoff (via the `/takeoff` service), the `/takeoff_server` node applies a short burst of thrust slightly greater than g on the `/control/takeoff` topic, and then publishes a user-specified hover point to the `/lqr_controller`. After a short wait to ensure convergence to the hover point, the `/takeoff_server` publishes an empty message to the `/in_flight` topic, indicating that the vehicle is ready for normal flight. When the user triggers the landing routine (via a call to the `/land` service), the `/takeoff_server` publishes again to the `/in_flight` topic (toggling it off) and slowly ramps down the thrust on the `/control/takeoff` topic. Whenever the vehicle is not in flight, the `/takeoff_server` periodically publishes a zero thrust command to refresh and make sure the quadrotor stays on the ground.

To review, there are a couple of key nodes in this example. Each of them lives in the corresponding ROS package:

- `/simulator`, which reads in controls and publishes poses to `/tf`
- `/state_estimator`, which reads from `/tf` and publishes state messages
- `/lqr_controller`, which reads state messages and publishes control messages
- `/takeoff_server`, which provides the `/takeoff` and /texttt/land services and publishes to the `/in_flight` and `/control/takeoff` topics
- `/takeoff_control_filter`, which decides whether to apply LQR control or the custom takeoff control

[1]The `/tf` topic is special, since ROS automatically designates it to keep track of the time-indexed positions and orientations of different frames of reference with respect to one another. There are a bunch of useful utilities that ROS provides — for example, the `tf2_ros` package provides a class that automatically keeps track of all the action on `/tf` and lets you query it for the most up-to-date relative transform between two arbitrary frames of reference.

3 Installation

This section provides step-by-step instructions for installing the code base on your own computer. For best results, you should be running Ubuntu 14.04 and either ROS Indigo or Jade. If you do not want to install Ubuntu natively on your machine, often a virtual machine (e.g. on Mac, using VM Fusion) works just fine and shouldn't require too much disk space, memory, or processing power.

3.1 Dependencies

The only real dependencies you'll need to install before you can build this code base are Eigen (one of the most popular C++ linear algebra toolkits) and gtest (Google's C++ unit testing suite), which may be installed easily using `apt-get` if you don't already have them. If you're not sure whether you have them or not, you can just move on and try to build the workspace, and if you get an error message complaining about a missing dependency you'll know what you're missing....

3.2 Building the workspace

You can build the workspace as you would any other ROS workspace: navigate to the top level directory (in this case, the `ros/` directory) and type in the command

```
catkin_make
```

If you get any errors that say things like `Could not find X` then you'll know you need to install dependency `X`. You may also see weird compiler out-of-memory issues if you're on an under-provisioned virtual machine — one way to avoid those problems is to build with a single thread: `catkin_make -j1`. Once you're done building the code base, you're all set to use it!

3.3 How to use this code base in another project

The most common way to use this code base is to write your own application in another ROS workspace and then tell that workspace about this one so that the ROS build system can find these packages. The way to do this is to "source" this workspace in any and all terminal windows you want to use to build and interact with your new workspace. The command to run (from the `ros/` directory) is

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

In general, you'll need to run this command for any workspace you want ROS to know about. If you want to make your life easier (and you plan to be using this workspace a lot), it might make sense to stick this command at the end of your `.bashrc` file (which lives in your home directory).

Now, in order to actually use any of the individual packages in some new package you're writing you'll need to declare them as dependencies in both the `package.xml` and `CMakeLists.txt` files for your new package. For examples of how to do this, please refer to the online ROS tutorials and/or similar files in this code base. If you don't do this, then any references you make to this code base in the new package's source code will be incomprehensible to ROS.

4 Software Example

Begin by running the built-in software example:

```
roslaunch crazyflie_examples sw_hover.launch
```

This should open up an RViz window where you'll see a set of coordinate axes sitting at the origin of a grid. There's actually a few sets of axes, but they're all on top of one another. In order to start the virtual quadrotor, you'll have to send the signal to takeoff. In another terminal window (with this workspace appropriately sourced), run:

```
rosservice call /takeoff
```

You should see one set of coordinate axes lift off the ground and hover a meter off the ground. If you zoom in far enough, you'll see the virtual Crazyflie displayed. When you're ready to land the quadrotor, you can run:

```
rosservice call /land
```

4.1 Arguments

Let us examine the `sw_hover.launch` file in the `launch/` directory of the `crazyflie_examples` package. Open it in a text editor. The top part of the file defines a few arguments to all of the nodes we'll need to launch. These are defined using the `<arg>` tag, since we'd like to specify a default value in the file and allow users to change them on the command line. They all look something like this:

```
<arg name="reference_topic" default="/ref" />
```

Once we've set all the arguments (and there are quite a few...) the only thing left is to tell ROS which nodes we would like to launch, and set their internal parameters appropriately. We'll go one by one.

4.2 Static transform publisher

```
<node pkg="tf"
      type="static_transform_publisher"
      name="robot_base_link_publisher"
      args="0 0 0 0 0 0 1 $(arg robot_frame) $(arg robot_frame)/base_link 100">
</node>
```

This node is a built-in ROS node that lives in the `tf` package. As the name suggests, it publishes a rigid body transformation between two frames of reference at a specified rate. In this case, it publishes the identity transform (given by the first seven numbers in the `args` field, the first three of which are position, and the last four of which are an orientation in quaternion form), between the robot frame and one called `base_link` that lives within the robot frame's namespace, at a rate of 100 Hz. The only reason to do this is so that we can display a URDF (universal robot descriptor file) of the Crazyflie in RViz (the ROS 3D visualization tool), since robot descriptors need to be referenced to a frame called `base_link`.

4.3 Simulator

```
<node name="simulator"
      pkg="crazyflie_simulator"
      type="near_hover_simulator_node"
      output="screen">

    <param name="time_step" value="$(arg simulator_dt)" />
    <param name="init/x" value="$(arg sim_init_x)" />
    <param name="init/y" value="$(arg sim_init_y)" />
    <param name="init/z" value="$(arg sim_init_z)" />

    <param name="frames/fixed" value="$(arg fixed_frame)" />
    <param name="frames/robot" value="$(arg robot_frame)" />

    <param name="topics/control" value="$(arg merged_control_topic)" />
</node>
```

This node is named `simulator`, but it is actually of type `near_hover_simulator_node` and it lives in the `crazyflie_simulator` package. As you can see, this node's internal parameters are set by evaluating the `args` defined above; the parameters' names are not arbitrary! They must align with the names that the node expects. For a clear example of how such paramters are loaded into the node at run-time, please examine the `LoadParameters` function in the `LinearFeedbackController` class, which is the parent class for all LQR controllers.

4.4 State estimator

```
<node name="state_estimator"
      pkg="crazyflie_state_estimator"
      type="position_velocity_yaw_state_estimator_node"
      output="screen">

    <param name="x_dim" value="$(arg x_dim)" />
    <param name="time_step" value="$(arg estimator_dt)" />

    <param name="frames/fixed" value="$(arg fixed_frame)" />
    <param name="frames/robot" value="$(arg robot_frame)" />

    <param name="topics/state" value="$(arg state_topic)" />
    </node>
```

Here we use the `state_estimator` node, which publishes position, velocity, and yaw information. The reason we don't need to keep track of roll or pitch information is because the control signals we get to send directly specify roll and pitch (more on that later).

4.5 LQR controller

```
<node name="lqr_controller"
      pkg="crazyflie_lqr"
      type="position_velocity_yaw_state_lift_lqr_node"
      output="screen">

    <param name="x_dim" value="$(arg x_dim)" />
    <param name="u_dim" value="$(arg u_dim)" />

    <param name="K_file" value="$(arg K_file)" />
    <param name="u_ref_file" value="$(arg u_ref_file)" />

    <param name="topics/reference" value="$(arg reference_topic)" />
    <param name="topics/state" value="$(arg state_topic)" />
    <param name="topics/control" value="$(arg lqr_control_topic)" />
    <param name="topics/in_flight" value="$(arg in_flight_topic)" />
</node>
```

This is the LQR hover controller. The underlying controller is of type `PositionVelocityYawStateLiftLqr` (which inherits from the `LinearFeedbackController` class), which may seem a little strange. This LQR controller expects incoming state messages to be of type `crazyflie_msgs::PositionVelocityYawStateStamped` (which includes position, velocity, and yaw along with a timestamp). The "Lift" part of the name indicates that the reference signal is of type `crazyflie_msgs::PositionVelocityStateStamped` (which does not include yaw) and the controller must "lift" this lower-dimensional reference state into the higher dimensional true state (in this case, it assumes the reference yaw is zero).

4.6 RViz

```
<node name="rviz"
      pkg="rviz"
      type="rviz"
      args="-d $(find crazyflie_simulator)/rviz/crazyflie_simulator.rviz"
      required="true">

    <param name="robot_description"
          command="$(find xacro)/xacro.py
                   $(find crazyflie_description)/urdf/crazyflie.urdf.xacro" />
</node>
```

This node just launches RViz (the built-in ROS 3D visualization tool) with a custom configuration file and loads up the URDF for the Crazyflie so that it appears in the right frame of reference.

4.7 Takeoff

```
<node name="takeoff_server"
      pkg="crazyflie_takeoff"
      type="takeoff_server_node"
      output="screen">

    <param name="topics/reference" value="$(arg reference_topic)" />
    <param name="topics/control" value="$(arg takeoff_control_topic)" />
    <param name="topics/in_flight" value="$(arg in_flight_topic)" />

    <param name="hover/x" value="$(arg takeoff_hover_x)" />
    <param name="hover/y" value="$(arg takeoff_hover_y)" />
    <param name="hover/z" value="$(arg takeoff_hover_z)" />
</node>

<node name="takeoff_control_filter"
      pkg="crazyflie_takeoff"
      type="takeoff_control_filter_node"
      output="screen">

    <param name="topics/takeoff_control" value="$(arg takeoff_control_topic)" />
    <param name="topics/commanded_control" value="$(arg lqr_control_topic)" />
    <param name="topics/final_control" value="$(arg final_control_topic)" />
    <param name="topics/in_flight" value="$(arg in_flight_topic)" />
</node>
```

It can be important to ensure that the quadrotor is well off the ground before beginning more complicated control maneuvers. The `takeoff_server` node provides a service called `/takeoff` (and another called `/land`). Calling this service initiates a short ramp-up of thrust on the `control/takeoff` topic for several seconds, then sends a reference point for the LQR controller to hover about. After a fixed amount of time (roughly 10 s) the server publishes a message on the `in_flight` topic, indicating that the Crazyflie is now in flight, hovering at the initial start position.

The `takeoff_control_filter` listens on these topics and decides whether to pass through the takeoff control or the LQR control to the vehicle.

5 Hardware Example

You can run the same example on the real quadrotor. First though, you'll need to do the following:

- Turn on the OptiTrack system and make sure it is properly calibrated.
- Turn on the Crazyflie.
- Place the Crazyflie at the desired start location and orientation.

At this point, the motion capture system should be turned on, but its output needs to be converted into ROS messages and `/tf` transforms. To do this, run

```
roslaunch mocap_optitrack mocap.launch
```

Note that this node reads in parameters stored in the `mocap_optitrack/config/mocap.yaml` file. If you need to change any of these parameters — e.g. if you wish to track more than one Crazyflie — then you can adjust them as needed (ideally in a copy of the original file).

At this point, you can go ahead and launch the hardware example launch file and send the takeoff service call (in different terminal windows, as usual).

```
roslaunch crazyflie_examples hw_hover.launch
rosservice call /takeoff
```

And when you're ready to land:

```
rosservice call /land
```

Open up the `hw_hover.launch` file. You'll notice that we've added a few parameters. Specifically, we have to specify which frame of reference the mocap system is using to refer to the Crazyflie (`mocap_frame`), the name of the radio to use to send commands to the Crazyflie (`uri`), and the name of the joystick in case we want to configure buttons for specific purposes (`joy_dev`).[2]

```
<!-- Frames of reference. -->
<arg name="mocap_frame" default="vicon/cf7/cf7" />

<!-- Crazyflie address. -->
<arg name="uri" default="radio://0/25/2M/E7E7E7E701" />

<!-- Joystick address. -->
<arg name="joy_dev" default="/dev/input/js0" />
```

The first thing we need to do is tie the mocap output to the robot's frame of reference. All this requires is a static transform publisher.

```
<!-- TF static publisher to tie the mocap output to the robot frame. -->
<node pkg="tf"
      type="static_transform_publisher"
      name="mocap_frame_publisher"
      args="0 0 0 0 0 0 1 $(arg mocap_frame) $(arg robot_frame) 100">
</node>
```

[2]Note that we will not actually be using the joystick in this example, but it is included in the launch file to provide a template for future launch files that may require it.

Finally, we need to replace the simulator with a group of nodes to handle communicating with the actual quadrotor, as follows:

```
<!-- Crazyflie interface. -->
<include file="$(find crazyflie_driver)/launch/crazyflie_server.launch" />

<group ns="crazyflie">
  <include file="$(find crazyflie_driver)/launch/crazyflie_add.launch">
    <arg name="uri" value="$(arg uri)" />
    <arg name="tf_prefix" value="$(arg robot_frame)" />
    <arg name="enable_logging" value="True" />
  </include>

<!--
  <node name="joy"
        pkg="joy"
        type="joy_node"
        output="screen">
    <param name="dev" value="$(arg joy_dev)" />
  </node>

  <node name="joystick_controller"
        pkg="crazyflie_demo"
        type="controller.py"
        output="screen">
    <param name="use_crazyflie_controller" value="True" />
  </node>
-->

</group>
```

Here, we have included launch files that start up the `crazyflie_server` and `crazyflie_add` nodes, which are provided by the original SDK for interfacing with the physical quadrotors over the radio. We also launch two nodes to manage the joystick in case future demos need them (we won't use the joystick here).

One last little detail: we need a node to convert the control signals output by our LQR controller into a form that the `crazyflie_server` node can read. The `cmd_vel_converter` node does this, like so:

```
<!-- Converter for custom command velocity messages. -->
<node name="cmd_vel_converter"
      pkg="crazyflie_control_merger"
      type="cmd_vel_converter_node"
      output="screen">

  <param name="topics/control" value="$(arg control_topic)" />
  <param name="topics/cmd_vel" value="/$(arg robot_frame)/cmd_vel" />
</node>
```

6 Anatomy of a ROS Node

All nodes in this code base are constructed essentially the same way. Even if you are familiar with ROS, it's important that you understand the way *these* nodes are organized so that when you write nodes of your own you can maintain a consistent style. This structure is very intentional; it is meant to facilitate modularity and reliability — for example, nodes will automatically shut down on launch (rather than sometime later once the quadrotor is airborne) if parameters are not loaded properly.

6.1 Executable

The node file itself is the one that gets turned into an *executable* by the compiler. Within each package, these files are stored in the `exec/` directory, and are named `*_node.cpp`.[3] They all look essentially the same, so let's look at one for an example. Open up the `near_hover_simulator.cpp` file in the `crazyflie_simulator` package. The code is shown below.

The first two lines specify header files that this executable depends on — in this case, the only dependencies are ROS and the `near_hover_simulator.h` file which declares the `NearHoverSimulator` class. We'll examine that class later; for now, focus on how it gets used.

Inside the `main` function, we initialize a new node called `simulator`. Then we create a new variable of type `NearHoverSimulator`. Notice that we did not pass in any arguments to the constructor — this is important, because we'd like all the parameters for this (and any other) class to be loaded at run-time from the ROS parameter server.

Next, we call the `Initialize` function within the `NearHoverSimulator` class. We'll take a closer look at this function later, but it does things like load parameters from the parameter server, register publishers and subscribers, etc. If it succeeds, then it will return `true` and the node will sit and wait for any callbacks to get triggered (e.g. by incoming messages). When you send an interrupt signal to the node (by pressing `Control-C` in the terminal where the node is running, for example), then `ros::spin()` will exit and the node will close down without error. On the other hand, if the `Initialize` function returned `false`, e.g. because a parameter was not loaded properly, then an error message prints out and the node shuts down.

```
#include <ros/ros.h>
#include <crazyflie_simulator/near_hover_simulator.h>

int main(int argc, char** argv) {
  ros::init(argc, argv, "simulator");
  ros::NodeHandle n("~");

  crazyflie_simulator::NearHoverSimulator simulator;

  if (!simulator.Initialize(n)) {
    ROS_ERROR("%s: Failed to initialize near_hover_simulator.",
              ros::this_node::getName().c_str());
    return EXIT_FAILURE;
  }

  ros::spin();

  return EXIT_SUCCESS;
}
```

[3] Python executables are a little different. Since this code base is all in C++, we will ignore Python for now; however, all the main ideas in this section have clear analogues in Python.

6.2 Initialization

The key to all of this is proper initialization of the `NearHoverSimulator` class. Let's look at the `Initialize` function inside that class:

```
bool NearHoverSimulator::Initialize(const ros::NodeHandle& n) {
  name_ = ros::names::append(n.getNamespace(), "near_hover_simulator");

  // Set state and control to zero initially.
  x_ = VectorXd::Zero(7);
  u_ = VectorXd::Zero(4);

  if (!LoadParameters(n)) {
    ROS_ERROR("%s: Failed to load parameters.", name_.c_str());
    return false;
  }

  if (!RegisterCallbacks(n)) {
    ROS_ERROR("%s: Failed to register callbacks.", name_.c_str());
    return false;
  }

  // Set initial time.
  last_time_ = ros::Time::now();

  initialized_ = true;
  return true;
}
```

All such `Initialize` functions are structured similarly. All begin by setting the `name_` member variable, which is used to label debug messages as coming from this particular class. In this case, we then set state `x_` and control `u_` — they will be reset later.

Next, we try to load parameters from the ROS parameter server and register any callbacks (for subscribers and services). If either of these steps fails, we error out and return `false`, which as we saw above causes the entire node to crash on start. Assuming everything works properly, we set the `initialized_` flag to `true` so that any other functions that may be called later will be able to confirm that this class was properly initialized.

Finally, let's see how to load parameters properly. Here's the same class' `LoadParameters` function. As you can see, we begin by creating a copy of the input `NodeHandle`, then we attempt to load each new parameter and return false if we encounter an error.

```
bool NearHoverSimulator::LoadParameters(const ros::NodeHandle& n) {
  ros::NodeHandle nl(n);

  // Frames of reference.
  if (!nl.getParam("frames/fixed", fixed_frame_id_)) return false;
  if (!nl.getParam("frames/robot", robot_frame_id_)) return false;

  // Time step for reading tf.
  if (!nl.getParam("time_step", dt_)) return false;

  // Control topic.
  if (!nl.getParam("topics/control", control_topic_)) return false;

  // Get initial position.
  double init_x, init_y, init_z;
  if (!nl.getParam("init/x", init_x)) return false;
  if (!nl.getParam("init/y", init_y)) return false;
  if (!nl.getParam("init/z", init_z)) return false;

  x_(0) = init_x;
  x_(1) = init_y;
  x_(2) = init_z;

  return true;
}
```