R3-URC18 Documentation

Release 1.0.0

mincrmatt12

Jul 18, 2018

Project structure

Welcome to the somewhat hidden documentation for R3-URC18! Here you might actually find useful information. I hope so, anyways.

CHAPTER 1

Structure

1.1 rover_cameras

The rover_cameras package contains the launch files and code to handle the rover's cameras.

There is one launch file to launch all the camera nodes. This also contains the camera switcher node.

1.2 rover_description

The rover_description package contains all the urdf related things. It's a quite simple package There is only one special folder in it, `urdf`

1.2.1 `urdf`

In here are all the xacro files for the rover urdf. Attachments like the arm or autonomous should be put in separate files and be parametrized so that different robot configurations are easily modeled.

1.3 rover_gazebo

`rover_gazebo` is the package that contains all of the gazebo stuff, like worlds and gazebo-specific textures/models. It has a few folders, organizing the above gazebo-relating things.

There is no code in the `rover_gazebo package` at this time, although should we need gazebo plugins they will be placed in another package, probably called `rover_gazebo_plugins`

1.3.1 `world`

The `world` folder contains all of the gazebo worlds, in `.world` format. The current worlds are:

Filename	Content/function
outdoor_world.gazebo	An outdoor world with a heightmap. Good for large-scale testing where a natural-ish landscape is important

Each world *should* have a launch file associated with it, which should spawn the rover with the same parameters as the (upcoming) master launch file. An easy way to create this is to use the `empty_world.launch` file, which allows for customizing the world location.

1.3.2 `media`

The `media` folder is where all of the gazebo "media" is: textures, models, heightmaps, etc. Any objects should be in their own folder, for example the "height" object (which is the heightmap used in the outdoor world) is in the folder `media/height`. Textures should be in a folder named `media/<object name>/textures`. Models should be in a folder named `media/<object name>/models`. Any other files the object needs, including its `.sdf` file, should be located in the root folder for that object.

Here is a table summarizing this info:

Folder location	Files located there
`media/<obj name>`	Files for the object named `obj_name`.
`media/<obj name>/textures`	Textures for the object named `obj_name`.
`media/<obj name>/models`	Models for the object named `obj_name`.

1.4 rover_gui

The rover_gui package contains nearly all of the guis for the rover, excluding the map (see see `rover_jsmap`)

The package contains one special directory, the `rqt` directory.

This directory contains each of the predefined rqt perspectives for the rover. You can load them by going into rqt and importing the perspective files.

See the usage docs for information on what each of them does.

1.5 rover_hw

`rover_hw` is the package which contains the HardwareInterface implementation for our rover. For more information on that, see either the internal documentation and/or the documentation for ros_control.

This package contains only one file defining the interface, but its structure is important

1.5.1 Hardware Extension Definition

Everytime we add a new hardware component with ros_control to the rover, a new hardware_interface "extension" is created in its package. For example, `rover_drive` has its implementation in `src/drive_hw`. This folder is built into a library, and that library needs to provide a class with the following methods:

Methods

`init(hardware_interface::RobotHW *hw)`

This method is called to initialize the hardware, as well as register any needed interfaces (e.g. `JointStateInterface`) on the RobotHW instance.

`read()`

This method should *only* update the interfaces with new values from the hardware, like encoders or limit switches. It is called *before* updating the controllers

`write()`

This method should only *write* to the hardware with new values from the interfaces. It is called *after* updating the controllers.

1.5.2 `hw_node.cpp` structure

This file contains the implementation of HardwareInterface. The convention for creating extensions to it was defined above.

For each extension you want to add, you must add the containing package as a dependency in the `package.xml` and `CMakeLists.txt` file.

In the denoted sections in the file, the declaration of your class and calling of its methods should be added, see how the rover_drive package does it as an example.

1.6 rover_navigation

`rover_navigation` is our implementation / configuration of a move_base stack.

1.6.1 `config`

The config folder contains all of the yaml configuration for `move_base`. There are three files in it.

`planner.yaml`

`planner.yaml` contains all of the configuration for the planners (global and local). For documentation on what the parameters in it mean, see the docs for `base_local_planner` and `navfn`.

`local_costmap.yaml` and `global_costmap.yaml`

These files contain the configuration for the local and global costmaps respectively. The documentation for them is in the *costmap_2d* package.

1.6.2 Internal Launch Files

Although the usage documentation only details one launch file, there are a lot of internal ones with more parametrization.

`move_base.launch`

The `move_base.launch` file contains the launch spec for starting move_base itself. It contains two useful parameters, located on lines 6 and 7:

This sets the `base_(global/local)_planner`.

`odometry.launch`

This is one of the most important files in the `rover_navigation` package. It contains the sensor fusion configuration. For documentation on how to configure it, see the documentation for `robot_localization`.

`visual_odom.launch`

This file contains the parameters (but right now mainly just remappings) for the rtabmap visual odometry.

`rtabmap.launch`

This file contains the parameters for rtabmap's main node. Add mapping parameters here.

CHAPTER 2

Usage

2.1 drive_side_controller

drive_side_controller is a controller in the rover_drive package that relays a single velocity command to multiple joints on one side of the rover. Its pluginlib name is rover_drive/DriveSideController. This controller uses the velocity interface.

2.1.1 Parameters

joints

This is the only parameter in drive_side_controller. It is a list and its contents are the joints to use for this side. Example:

```
joints:
    - back_left_wheel
    - front_left_wheel
```

This would make the controller control the *back_left_wheel* and *front_left_wheel* with the same command.

2.1.2 Topics

drive_side_controller/cmd

This topic is subscribed to as a Float64 topic containing the current velocity command for all the joints.

2.2 rover_cameras

The nodes in the `rover_cameras` package are designed to manipulate rover camera streams. The launch files are what launch them to get all the feeds actually running.

2.2.1 Nodes

`image_rotater`

The `image_rotater` node takes in one input stream and rotates it by a constant amount.

Topics

- `~image_rotater/image_in` - image stream to rotate
- `~image_rotater/image_out` - published image stream

Parameters

- `angle` - angle in degrees to rotate by

`camera_diagnostics`

The `camera_diagnostics` node monitors `/dev/video` paths and publishes diagnostic information for them.

Parameters

- `~expected_nums` - video numbers to monitor

`swapper.py`

Runs and manages `usb_cam` nodes to deal with cameras with a ros api. Typically you use the cli to interact with it.

2.2.2 Launch files

The only launch file in *rover_cameras* starts the camera swapper.

2.2.3 CLI

Important: The cli for camera swapping requires you install the *prompt_toolkit* python package.

The cli (node name `swapper_cli.py`) allows you to interact with the swapper. Each camera is just represented as a name and a video device, the node doesn't actually store these things.

The CLI has built in help, (as well as autocomplete) but documentation is also provided here:

Commands

quit

Quits the CLI.

load

Two parameters, first is the camera name, next is the video device (/dev/videoN). Optionally takes a parameter, which is the preffered device to unload if we cannot run this many cameras. Using this parameter allows you to quickly switch cameras.

` > load belly_cam /dev/video0 head_cam ` Swaps head_cam with belly_cam

unload

Unloads the camera (by name) given to it as a parameter.

list

Lists all running cameras by name.

2.3 rover_control

2.3.1 Launch Files

rover_control.launch

This launch file is for loading controller parameters and starting robot_state_publisher. It takes two parameters:

- use_fake - default false - whether or not to load the fake_controllers file.
- controllers - default depends on use_fake - which controllers to load by default.

The launch file also starts the rover_hw node if the use_fake parameter is set to false.

2.4 rover_gazebo

All of the useful functionality in rover_gazebo is in launch files. Since there is no "real" top-level launch file to refer to, all of the launch files used to start gazebo have *no parameters*.

2.4.1 Launch files

As explained in the structure documentation, each world defined there has a launch file corresponding to it. There are no parameters in these launch files. For all the worlds, see the table under worlds in that file.

2.5 rover_hw

The rover_hw package contains the hardware interface and controller node.

2.5.1 `rover_hw_node`

`rover_hw_node` is the node that runs all the controllers. It takes no parameters, as parametrization of hardware interfaces should be done as compile time constants.

Running this node is enough to start a controller manager with proper hardware interfaces.

2.6 rover_ik

2.6.1 `ik_joint_controller`

The `rover_ik::IKJointController` is a controller in ros_control that does inverse kinematics on a set of joints and feeds the result to a PID loop. A full usage example can be seen in rover_control.

Parameters

joints

This parameter is a list of joints to control, each one should provide an `EffortJointInterface`

gains

Each subparameter in this parameter corresponds to PID gains for a joints, for example:

```
gains:
  arm_base_to_post: {p: 7, i: 1, d: 0, i_clamp: 1}
```

This sets the gains for joint `arm_base_to_post` to P = 7, I = 1, D = 0 and I_Clamp = 1.

base_link and *tip_link*

This correspond to the *links* at each end of the joint chain, base being the start and tip being the end.

Topics & Services

controller_name/target

Controller listens on this topic for the target pose of the *tip_link*, as defined in the parameters. The coordinates for this pose are relative to the `base_link`. (testing required)

controller_name/request_position

This service can be called to return the current position of the arm as calculated via forward kinematics. Passing this back into `target` should cause no effect to the arm.

2.7 rover_navigation

To use the rover_navigation package to start a working move_base stack, run the `rover_navigation.launch`. That's it. This launch file will start the entirety of the navigation stack. To use move_base, see its documentation. When gps_goal and friends are implemented they'll be put here.

2.8 rover_teleoperation

2.8.1 Launch Files

`simple_drive_teleop.launch`

This launch file starts a `joy` node and a drive teleoperation node. It takes one parameter, `dev` which is the joystick to open.

Important: The teleoperation node expects an XBox controller (or compatible), otherwise the mappings will not be correct

---

2.8.2 Nodes

`arm_ik_joy_teleop`

This node (currently without a launch file) is for commanding the ik_joint_controller with a joystick.

It listens on the `joy` topic for joystick input and outputs to the `arm_ik_controller/target` topic.

`simple_drive_joy_node`

This node commands drive from a diff drive joystick.

It listens on the `joy` topic for joystick input and outputs to the `/left_wheels_controller/cmd` and `/right_wheels_controller/cmd` topics.

2.9 rover_description

2.9.1 `urdf`

Our urdf is contained in the `rover_description` package. Various different configurations for it are available (todo: actually add them), which can be specified with launch file parameters to the `description.launch` file. Right now the current configuration parameters are:

<none>

The launch file places the compiled urdf into `/robot_description` on the parameter server.

CHAPTER 3

Hardware

3.1 Arduino Protocol (eml_uberdriver)

The eml_uberdriver protocol documentation is now hosted elsewhere due to it being a seperate library.

Find it here: http://r3enamel.readthedocs.io/en/latest

CHAPTER 4

Tutorials

4.1 Starting the rover

Starting the rover is not a very complicated procedure. First, you need to ssh into the TX2, then start rover.launch. This process is detailed below.

4.1.1 SSH-ing to rover

To ssh into the rover, use `$ ssh ubuntu@<ip>`

The IP depends on which TX2 is currently being used. If it is marked on top of the TX2, use it otherwise refer to this table.

TX2 Desc	Ip Address
On Rover	192.168.137.213

Once on the rover, navigate to `~/URC-18`

4.1.2 Starting rover launchfile

First, ensure ros is properly sourced with `$ source /opt/ros/kinetic/setup.bash` Next, source the rover's setup file with `$ source rosws/devel/setup.bash`

Finally, start the rover software with `roslaunch rover rover.launch`. To start autonomous nodes, add `start_auton:=true` to the end of the previous command.

This procedure will start the rover's software, but to control it from a basestation you need to read on.

4.2 Controlling the rover

Now that you've started the rover, you need to control it. The first thing you need to do is start the watchdog client.

Note: The watchdog automatically stops the rover if the basestation is not connected for more than 1 second.

To start the watchdog client, use `$ rosrun rover wclient.py`.

4.2.1 Drive

Teleoperating the drivetrain is very easy. Start by running `$ roslaunch rover_teleoperation simple_drive_teleop.launch`. If you get an error about unable to open joysticks, you may need to find the right joystick device under `/dev/input/js<N>` and add `dev:=/dev/input/js<N>` to the end of the command.

The controls are simple (on an xbox style controller):

- left & right stick - tank controls
- dpad up & down - gear up/down (speed levels)

4.2.2 Camera

Cameras can be viewed using the rqt utility. Adding the image view plugin under visualization allows you to view image topics. To control cameras, you need to use the camera swapper cli. Information on how it works and the commands it supports can be found at ../usage/rover_cameras.rst.

Using rqt, you can view the images by clicking the refresh icon in the image view plugin, then selecting the topic ending with "compressed" and with the name of the camera you set in the cli.