**Introduction**

The Rodney Robot project is a hobbyist robotic project to design and build an autonomous house-bot using ROS (Robot Operating System). This article is the sixth in the series describing the project.

**Background**

In [part 1](https://www.codeproject.com/Articles/1249436/Rodney-A-long-time-coming-autonomous-robot-Part-1), to help define the requirements for our robot, we selected our first mission and split it down into a number of Design Goals to make it more manageable.

The mission was taken from the article, [Let's build a robot!](https://www.codeproject.com/Articles/1115414/Lets-build-a-robot) and was: *Take a message to... - Since the robot will [have] the ability to recognize family members, how about the ability to make it the 'message taker and reminder'. I could say 'Robot, remind (PersonName) to pick me up from the station at 6pm'. Then, even if that household member had their phone turned on silent, or were listening to loud music or (insert reason to NOT pick me up at the station), the robot could wander through the house, find the person, and give them the message.*

The design goals for this mission were:

* Design Goal 1: To be able to look around using the camera, search for faces, attempt to identify any people seen and display a message for any identified
* Design Goal 2: Facial expressions and speech synthesis. Rodney will need to be able to deliver the message
* Design Goal 3: Locomotion controlled by a remote keyboard and/or joystick
* Design Goal 4: Addition of a laser ranger finder or similar ranging sensor used to aid navigation
* Design Goal 5: Autonomous locomotion
* Design Goal 6: Task assignment and completion notification

In the last part we added motor control and odometry feedback to complete Design Goal 3. In this part, I'm going to add a spinning LIDAR (light detection and ranging) to complete Design Goal 4, include existing ROS packages for SLAM (simultaneous localization and mapping), a probabilistic localization system,  global and local navigational planning, all to put us well on our way to completing Design Goal 5. I'm also going to include an option to replace the Arduino Nano with a Teensy 3.5 to improve data update rates and to give room for future expansion.

**Adding a LIDAR**

The most common method for autonomous navigation in ROS require topic subscription to the both the */scan* and */tf* topics. The */tf* topic is used to obtain the odom transform, which we started to broadcast from the *thunderborg* node in the last article and the */scan* topic contains data from a laser scan device.

Now you see some large and very expensive 360 degree scanning devices on autonomous cars that are under development, but as Rodney is going to be confined to the house we can considerably step down the price range. Slamtec produce a number of spinning LIDARs and their [RPLidar A1](https://www.slamtec.com/en/Lidar/A1) is reasonably priced at around £100 (GBP). The RPLidar has a range of  12 meters, gives a full 360 degree scan and uses a serial interface for communication. But wait it gets better, they have even developed a ROS node available for download from the [Slamtec GitHub page](https://github.com/slamtec/rplidar_ros). We can therefore plug this device straight into our ROS enabled robot and start receiving messages on the */scan* topic without having to figure out what serial data it requires and transmits.

Below is a photo of the RPLidar installed on Rodney.

The version of the RPLidar I purchased is the Development Kit and comes with a USB serial device and cable for connecting this device to the LIDAR. As well as the Tx/Rx lines it supplies the LIDAR motor and LIDAR core with 5V. There is also a PWM input to the core which can be used to control the speed of the motor, if you use the supplied USB serial device this line is connected to the DTR and can be used to enable/disable the device.

As I'm trying to share the power load between the different supplies I have on Rodney, I decided to supply the motor and the core from different supplies and have only the USB device powered from the Raspberry Pi USB power. I therefore built a simple breakout board so that I could feed the power to the LIDAR separately. The image below shows this breakout board and the USB serial device mounted on the rear of the platform where the LIDAR is installed. The separate power comes in on the 3 pins which are not connected in the photo.  The stl files for the platform and standoffs are available in the 3D print zip file included with this article.

Now some of you that have been following these articles from the start may have spotted a problem. Although I have mounted the RPLidar high enough so that the electronics on the rear platform are below its laser, Rodney's neck is going to be in the way of the laser and thus always result in detection in the 360 degree field. There is available an existing ROS package called [laser filters](http://wiki.ros.org/laser_filters) which will allow us to use a plugin called *LaserScanAngularBoundsFilter*to remove the points from the message. We will use it to remove 10 degrees either side of the 180 degree mark.

To add the code for the RPLidar and for the navigation functionality we are going to edit some of the packages created in the previous parts of this article, namely rodney and rodney\_missions.

The laser filter node will be launched from our *rondey.launch* file but is will also require a configuration file which we will load into the ROS parameter server. We therefore need to add the following file named *laser\_filter\_config.yaml* to the *rodney/config* folder.

scan\_filter\_chain:

- name: angle

type: laser\_filters/LaserScanAngularBoundsFilter

params:

lower\_angle: -2.70526

upper\_angle: 2.70526

The lower and up angles are in radians which will restrict our field to -170 to +170 degrees.

To load these parameters into the parameter server we will add the following line near the beginning of our *rodney.launch* file in the *rodney/launch* folder. We are going to be making quite a few changes to the launch file so I'll show it in its entirety at later in this article.

<rosparm command="load" file="$(find rodney)/config/laser\_filter\_config.yaml" ns="scan\_to\_scan\_filter\_chain"/>

To launch the node we will add the following to the rodney.launch file.

<node pkg ="laser\_filters" type="scan\_to\_scan\_filter\_chain" name="scan\_to\_scan\_filter\_chain" output="screen">

<remap from="scan" to="scan\_filter\_input"/>

<remap from="scan\_filtered" to="scan"/>

</node>

We need to do some remapping of the topic names, the output topic of the node is *scan\_filtered*, but the navigation packages will be subscribing to scan. We will also be remapping the RPLidar published topic from scan to scan\_filter\_input.

Next we need to launch the RPLidar node provided by Slamtec. A slight problem here is that the serial device provided is identified by the Linux system as */dev/ttyUSBn* as is the Arduino Nano. One will be ttyUSB0 and the other will be ttyUSB1 but we can't be sure which one will be which.

We can avoid this uncertainty by adding some udev rules which will create a set of symbolic links which we will use in the launch file. In the *rodney/scripts* folder I have created the udev rules in the *rodney\_udev.rules* file.

# Set the udev rules.

#

# Arduino Nano

KERNEL=="ttyUSB\*", ATTRS{idVendor}=="1a86", ATTRS{idProduct}=="7523", MODE:="0777", SYMLINK+="nano"

#

# Teensy

KERNEL=="ttyACM\*", ATTRS{idVendor}=="16c0", ATTRS{idProduct}=="0483", MODE:="0777", SYMLINK+="teensy"

#

# RPLidar

KERNEL=="ttyUSB\*", ATTRS{idVendor}=="10c4", ATTRS{idProduct}=="ea60", MODE:="0777", SYMLINK+="rplidar"

Using the a sudo copy command, copy this file to the */etc/udev/rules.d* folder on the Raspberry Pi. I have also created a script file called *create\_udev\_rules.sh*, which will do the copying.

Next we can add the command to launch the RPLidar node to the *rodey.launch* file.

<node pkg="rplidar\_ros" type="rplidarNode" name="rplidar\_node" output="screen">

<param name="serial\_port" type="string" value="/dev/rplidar"/>

<param name="serial\_baudrate" type="int" value="115200"/>

<param name="frame\_id" type="string" value="laser"/>

<remap from="scan" to="scan\_filter\_input"/>

</node>

For the *rosserial\_python* launch update the "/dev/ttyUSB0" to "/dev/nano" when using the Arduino Nano.

We will build and run all the code in the "Using The Code" section later in the article, but the image below shows the laser scan message visulised in rviz.

**Navigation "Stack"**

In older versions of ROS packages could be organised into ROS stacks, this is an outdated term but sometimes you will still see the term stack used especially the "Nav Stack". This now refers to a number of packages that can be used together for autonomous navigation. Like the node used for the LIDAR we will be not writing new code but using existing ROS packages. In this section I'll explain hwo to configure and launch these packages, however some of them contain many configurable parameters and you should refer to the ROS Wiki to get an understanding of each package.

**Mapping**

In order for our robot to navigate it will need a map of its world. The node we are going to use to create the map will be run on a Linux workstation and constructs the map from data recorded from the robot sensors. The ROS package is called [gmapping](http://wiki.ros.org/gmapping) and provides laser-based SLAM (Simultaneous Localization and Mapping). The following image shows a map created by gmapping from Rodney's sensors displayed in rviz. Its no perfect but it is possible to use if for navigation.

The map is stored for later use in two files, a .yaml and a .pgm file. The .pgm file can be edited in a number of image editing tools so you can always tidy up the map and block off areas you may not wish your robot to visit.

You can create maps live but most tutorials will tell you to record the sensor data and then produce the map so that you can adjust the parameters and reproduce the map. I prefer a mixture of the two, record the data but also visualise the map in rviz as it is created.

You create the map by driving your robot slowly in manual mode. To improve the map quality as well as driving slowly it is best to visit a location more than once. I'll show how to record the data and how to create and save the map in the "Using the Code" section.

Once we have a map the robot will need access to the map and as usual with ROS this is done using topics. The node providing these topics is part of the [map\_server](http://wiki.ros.org/map_server) package. Since the map may be large the topics are not continually published but are latched and any new node requiring the map will be passed a copy. As well as containing the *map\_server* node the package includes another node called *map\_saver* which will be used to save the map created by gmapping to disk.

To launch the *map\_server* on the robot hardware we will include the following in the *rodney.launch* file.

<arg name="map\_file" default="second\_floor"/>

<node pkg="map\_server" type="map\_server" name="map\_server" args="$(find rodney)/maps/$(arg map\_file).yaml" output="screen"/>

This implies that we will store our map files in a new folder in the rodney package called *maps*. If when we call the launch file we don't provide a *map\_file* parameter the default value of second\_floor will be used.

**Robot Localisation**

The next part of this navigation jig-saw puzzle we require is Robot Localisation. As well as keeping track of where it is in the world it will also need to know its starting place. Now the odom and laser sensors are not perfect so the robot can't tell exactly where it is but will maintain an array of candidate poses of possible locations. A pose is a position and orientation in the world. As the robot moves and it narrows down where it thinks it is it will discard the poses that are less likely to be its position.

For this functionality we will use the [amcl package](http://wiki.ros.org/amcl). AMCL stands for Adaptive Monte Carlo Localization, luckily all we need to do is to configure it ans launch it, if you wish to understand it more then may be this [Wikipedia page](https://en.wikipedia.org/wiki/Monte_Carlo_localization) is a good starting place.

As is normal with ROS we will configure the node by loading some configuration data onto the parameter server. This data will be stored in the *amcl\_config.yaml* file stored in the *rodney/config* folder. This is my version of the configuration file but as with these navigation packages they are highly configurable so consult the ROS Wiki for the package.

amcl:

# Overall filter parameters

min\_particles: 500

max\_particles: 3000

kld\_err: 0.05

kld\_z: 0.99

update\_min\_d: 0.2

update\_min\_a: 0.5

resample\_interval: 1

transform\_tolerance: 0.5

recovery\_alpha\_slow: 0.0

recovery\_alpha\_fast: 0.0

gui\_publish\_rate: 10.0

# Laser model parameters

laser\_max\_beams: 30

laser\_z\_hit: 0.5

laser\_z\_short: 0.05

laser\_z\_max: 0.05

laser\_z\_rand: 0.5

laser\_sigma\_hit: 0.2

laser\_lambda\_short: 0.1

laser\_likelihood\_max\_dist: 2.0

laser\_model\_type: beam

# Odometry model parameters

odom\_model\_type: diff

odom\_alpha1: 0.2

odom\_alpha2: 0.2

odom\_alpha3: 0.8

odom\_alpha4: 0.2

odom\_alpha5: 0.1

odom\_frame\_id: odom

 We will need to add the the following line near the top the launch file to load the file into the parameter sever.

<rosparam command="load" file="$(find rodney)/config/amcl\_config.yaml"/>

  The following added to the launch file will launch the node.

<node pkg="amcl" type="amcl" name="amcl" output="screen"/>

It's worth noting that most of the parameters I have changed for from defaults for the nav stack relate to transform tolerances and and sample frequencies, we are asking a lot of the Raspberry Pi here!

**Planning a Route**

Now we are getting to the business end of the process. If we want to get from point A to point B (or pose A to pose B) using the map something needs to plan the route and command the robot to move. For this we are going to use the [move\_base](http://wiki.ros.org/move_base) package. This package is going to have to use the amcl data, the map, the odom data and the laser data to plan the best route. Not only that it needs to allow for things changing since the map was constructed. If a person or a family pet walking into the path of the robot it will need to come up with a new route. Again to gain some understanding of this package read the ROS Wiki page on it.

One of the things to note is that it use two costmaps to plan routes. A costmap shows good and bad places for a robot to be. A good place is out in the open and a bad place would be against a wall. One costmap is the Global costmap and is based on the map and remains static. This map is used to plan the global route, but as said before what if things have changed from the map. This is were the Local costmap comes in. It's updated as the robot moves and where as the Global costmap covers the entire map the Local costmap covers only the immediate location around the robot.

A problem I have encountered on Rodney is that the Local costmap is in th odom frame and although I have tuned the odom parameters (wheel circumference and wheel distance) there are times when the Local costmap can become rotated in axis to the Global one and hence routes through narrow places are blocked off. I can get around this problem by resetting the costmaps before a new nav goal is set and I'll explain this in the Using The Code section.

The odom is always going to be an estimate position but in the future I plan to look at including an IMU (Inertial Measurement Unit) and a Extended Kalman filter to fuse the odom and IMU data to produce a more accurate odom.

We need a number of configuration files for the *move\_base* which will be loaded into the parameter sever. Each one will be located in the rodney/config folder.

*base\_local\_planner\_params.yaml*

move\_base:

controller\_frequency: 1.4

TrajectoryPlannerROS:

holonomic\_robot: false

meter\_scoring: true

max\_vel\_x: 0.25

min\_vel\_x: 0.17

escape\_vel: -0.2

max\_vel\_theta: 1.5

min\_vel\_theta: -1.5

  min\_in\_place\_vel\_theta: 1.2

Here I had to slow down the controller frequency, the holonomic\_robot false value basically states that we are a two wheeled robot using differential drive foe steering. As this node will be responsible for issuing velocity commands we also have to give it some bounds for those velocities.

The costmaps have some common parameters and some which are set for the two different costmaps.

*costmap\_common\_params.yaml*

obstacle\_range: 2.5

raytrace\_range: 2.0

footprint: [[0.170, 0.090], [0.025, 0.090], [0.025, 0.145], [-0.025, 0.145], [-0.025, 0.090], [-0.170, 0.090], [-0.170, -0.090], [-0.025, -0.090], [-0.025, -0.145], [0.025, -0.145], [0.025, -0.090], [0.170, -0.090]]

observation\_sources: laser\_scan\_sensor

laser\_scan\_sensor:

sensor\_frame: laser

data\_type: LaserScan

topic: scan

marking: true

clearing: true

This gives a footprint for the base including the wheels that stick out from the side and information on how to find the topic with our filtered laser ranges.

*global\_costmap\_params.yaml*

move\_base:

global\_costmap:

global\_frame: map

robot\_base\_frame: base\_link

static\_map: true

transform\_tolerance: 0.6

update\_frequency: 1.0

*local\_costmap\_params.yaml*

move\_base:

local\_costmap:

global\_frame: odom

robot\_base\_frame: base\_link

static\_map: false

rolling\_window: true

transform\_tolerance: 0.7

update\_frequency: 1.0

  publish\_frequency: 0.5

Like the main the map the costmaps are latch topics and not continually published. So that I can see the local costmap update in rviz I have changed the publish\_frequency from 0.0 to 0.5 so that is published every 2 seconds. Once the navigation is working it will be prudent to go back to the default so that the map is not continually published.

These files will be loaded into the parameter server with the inclusion of the following lines in the *rodney.launch* file.

<rosparam command="load" file="$(find rodney)/config/base\_local\_planner\_params.yaml"/>

<rosparam command="load" file="$(find rodney)/config/local\_costmap\_params.yaml"/>

<rosparam command="load" file="$(find rodney)/config/global\_costmap\_params.yaml"/>

Note that this does note load the costmap common parameters, this file will be loaded twice into different name spaces for the local and global costmaps when the nodes are launched.

To launch the *move\_base* node I have added the following to the *rodney.launch* file.

<node pkg="move\_base" type="move\_base" name="move\_base" respawn="false" output="screen">

<rosparam command="load" file="$(find rodney)/config/costmap\_common\_params.yaml" ns="global\_costmap"/>

<rosparam command="load" file="$(find rodney)/config/costmap\_common\_params.yaml" ns="local\_costmap"/>

<remap from="cmd\_vel" to="demand\_vel"/>

</node>

The published topic from move\_base is normally *cmd\_vel* , which is the topic which our thunderborg node subscribes to. Since we wish to be able to drive the robot manually our *rodney*node was set up to publish on the *cmd\_vel* topic either data for manual driving or autonomous depending on the current mode of the robot. We therefore remap the topic name here from *cmd\_vel* to *demand\_vel*.

**Were is the Laser?**

We need to add just one more missing piece to the launch file. The navigation stack requires odom and laser scan data but is also requires knowledge of where the laser is fitted to the robot. The centre of our robot is in the middle of the base between the wheels, this is known as the base\_link and the navigation is centre around this point. However the RPLidar node will be reporting ranges from its frame known as the laser frame, so we need to give the navigation stack information on how to transform the laser data to be meaningful for the robot base.

We could write a node which broadcasts this transform and there is an [article in the ROS Wiki here on how to do that](http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF). However, there is another way were we use an existing package that will do the work for us. We can launch it from our launch file and pass the data required to it. The RPLidar is 85mm forward of and 107mm above the centre of the robot. So by adding the following line this transform will be broadcast every 100ms (10Hz).

<node pkg="tf" type="static\_transform\_publisher" name="laser\_link\_broadcaster" args="0.085 0 0.107 0 0 0 base\_link laser 100"/>

**Launch File**

Another change I have made to the *rodney.launch* file is to remove the

output="screen"

from each of the nodes which are no longer under development or test. This reduces the log messages displayed in the terminal that we launch the robot code from. However, you can still use *rqt\_console* on a remote terminal to monitor all the log message.

The full *rodney.launch* file now looks like this:

<?xml version="1.0" ?>

<launch>

<!-- Load each of the config files into the parameter server -->

<rosparam command="load" file="$(find rodney)/config/config.yaml"/>

<rosparam command="load" file="$(find rodney)/config/laser\_filter\_config.yaml" ns="scan\_to\_scan\_filter\_chain"/>

<rosparam command="load" file="$(find rodney)/config/amcl\_config.yaml"/>

<rosparam command="load" file="$(find rodney)/config/base\_local\_planner\_params.yaml"/>

<rosparam command="load" file="$(find rodney)/config/local\_costmap\_params.yaml"/>

<rosparam command="load" file="$(find rodney)/config/global\_costmap\_params.yaml"/>

<rosparam command="load" file="$(find pan\_tilt)/config/config.yaml"/>

<rosparam command="load" file="$(find face\_recognition)/config/config.yaml"/>

<rosparam command="load" file="$(find head\_control)/config/config.yaml"/>

<rosparam command="load" file="$(find rodney\_missions)/config/config.yaml"/>

<rosparam command="load" file="$(find thunderborg)/config/config.yaml"/>

<!-- Launch the camera node from one of its launch files -->

<include file="$(find raspicam\_node)/launch/camerav2\_1280x960.launch" />

<!-- Start all the nodes that make up Rondey -->

<!-- Starting with those written for the project -->

<node pkg="pan\_tilt" type="pan\_tilt\_node" name="pan\_tilt\_node"/>

<node pkg="face\_recognition" type="face\_recognition\_node.py" name="face\_recognition\_node"/>

<node pkg="head\_control" type="head\_control\_node" name="head\_control\_node"/>

<node pkg="speech" type="speech\_node" name="speech\_node"/>

<node pkg="rodney\_missions" type="rodney\_missions\_node.py" name="rodney\_missions" output="screen"/>

<node pkg="rodney" type="rodney\_node" name="rodney\_node" output="screen" />

<node pkg="thunderborg" type="thunderborg\_node.py" name="thunderborg"/>

<!-- The Arduino or Teensy.

For Arduino Nano use /dev/ttyUSB0 (or nano if dev rules updated) and 57600.

For the Teensy use the defaults /dev/ttyACM0 (or teensy if dev rules updated) and 500000 -->

<arg name="serial\_port" default="/dev/teensy"/>

<arg name="baud\_rate" default="500000"/>

<node pkg="rosserial\_python" type="serial\_node.py" name="serial\_node" output="screen">

<param name="port" value="$(arg serial\_port)"/>

<param name="baud" value="$(arg baud\_rate)"/>

</node>

<!-- The RPLidar and laser filter node

Have created symbolic link for /dev/ttyUSBn to be rplidar -->

<node pkg="rplidar\_ros" type="rplidarNode" name="rplidar\_node">

<param name="serial\_port" type="string" value="/dev/rplidar"/>

<param name="serial\_baudrate" type="int" value="115200"/>

<param name="frame\_id" type="string" value="laser"/>

<remap from="scan" to="scan\_filter\_input"/>

</node>

<node pkg ="laser\_filters" type="scan\_to\_scan\_filter\_chain" name="scan\_to\_scan\_filter\_chain">

<remap from="scan" to="scan\_filter\_input"/>

<remap from="scan\_filtered" to="scan"/>

</node>

<!-- The robot face -->

<node pkg="homer\_robot\_face" type="RobotFace" name="RobotFace"/>

<!-- Static transform for the laser -> baselink -->

<node pkg="tf" type="static\_transform\_publisher" name="laser\_link\_broadcaster" args="0.085 0 0.107 0 0 0 base\_link laser 100"/>

<!-- Navigation -->

<arg name="no\_nav" default="false"/>

<group unless="$(arg no\_nav)">

<arg name="map\_file" default="second\_floor"/>

<node pkg="map\_server" type="map\_server" name="map\_server" args="$(find rodney)/maps/$(arg map\_file).yaml" output="screen"/>

<node pkg="amcl" type="amcl" name="amcl" output="screen"/>

<node pkg="move\_base" type="move\_base" name="move\_base" respawn="false" output="screen">

<rosparam command="load" file="$(find rodney)/config/costmap\_common\_params.yaml" ns="global\_costmap"/>

<rosparam command="load" file="$(find rodney)/config/costmap\_common\_params.yaml" ns="local\_costmap"/>

<remap from="cmd\_vel" to="demand\_vel"/>

</node>

</group>

</launch>

You can see for the navigation I have included a number of parameters that can be passed to the *roslaunch* when using this file.

When driving the robot around manually to create a map we don't want the map server publishing a map so we can exclude the navigation stack with the following:

$ roslaunch rodney rodney.launch no\_nav:=True

By default if the navigation stack is included my *second\_floor* map will be used. You can pass a different map with the following:

$ roslaunch rodney rodney.launch map\_file:=ground\_floor

**Minor Code Changes**

So we have added a number of ROS nodes to the system but have up to now written any new code or updated any of the old code. In this part of the articles I'm not going to call the nav stack programmatically, we will do that in the next part and use the tool rviz in this part to examine the data and to set nav goals to move the base. I have however made some minor changes to the code mostly around been able to enable/disable the LIDAR when the robot is in manual mode. I have also made changes so that we can use either the Arduino Nano or Teensy 3.5 in the system.

**Updates to the thunderborg Node**

This is in fact a very small change but that came about from using the navigation stack. As mentioned above the base\_link frame is the centre of the robot base and in the *thunderborg*node we broadacst the transform between this frame and the odom frame. This is the x and y displacement and the rotation around z between these frames. However, the base\_link is also a fixed distance above odom due to the fact it is sitting on its wheels. I have therefore added this fixed distance along the z axis to both the odom message and transform broadcast. The base\_link is 90mm above the odom.

...

# Send the transform

self.\_\_odom\_broadcaster.sendTransform((self.\_\_odom\_x, self.\_\_odom\_y, 0.09),

odom\_quat,

current\_time,

'base\_link',

'odom')

...

# The pose is specified to the odom co-ordinate frame

odom.pose.pose = Pose(Point(self.\_\_odom\_x, self.\_\_odom\_y, 0.09), Quaternion(\*odom\_quat))

**Updates to the rodney\_missions Node**

In the next part of these articles we will changing the *rodney\_missions* node to programatically start autonomous navigation to a new pose. Here I have made small changes so that this node can enable/disable the LIDAR by starting and stopping the LIDAR motor. The request come from the *rodney*node but it's important the control is done here so that this node can ensure the LIDAR is running when in the future it request autonomous navigation.

The starting and stopping of the motor is done via ROS service calls to the LIDAR node. The first of the changes are in the \_\_init\_\_ function of the MissionsHelper class. We wait for the services to become available, create proxy calls so that we can access the service and then call the LidarEnable helper function to ensure the LIDAR is currently running.

...

  # RPLidar services to start and stop the motor

rospy.wait\_for\_service('stop\_motor')

rospy.wait\_for\_service('start\_motor')

self.\_\_rplidar\_stop\_motor\_srv = rospy.ServiceProxy('stop\_motor', std\_srvs.srv.Empty)

self.\_\_rplidar\_start\_motor\_srv = rospy.ServiceProxy('start\_motor', std\_srvs.srv.Empty)

# LIDAR should be running but make sure

self.LidarEnable()

Next I have added three helper functions to MissionHelper class. They call the service in question and keep track of the current state of the LIDAR motor.

...

# Function to enable the RPLidar

def LidarEnable(self):

self.\_\_rplidar\_start\_motor\_srv()

self.\_\_lidar\_on = True

# Function to disable the RPLidar

def LidarDisable(self):

self.\_\_rplidar\_stop\_motor\_srv()

self.\_\_lidar\_on = False

# Function to Toggle RPLidar on/off

def ToggleLidar(self):

if(self.\_\_lidar\_on == True):

self.LidarDisable()

else:

self.LidarEnable()

The final change is in the Prepare class and adds a new elif construct to the execute function. If the job received is of id 'J4' then the request is to toggle the current state of the LIDAR.

...

elif parameters[0] == 'J4':

# Simple job to toggle the LIDAR on/off

self.\_\_helper\_obj.ToggleLidar()

return retVal

**Updates to the rodney Package**

We need to make some changes to the *rodney*node so that we can use the joystick and/or keyboard attached to the remote workstation to enable/disable the LIDAR when in manual mode.

On occasions when I have been manually controlling the robot, my home network has dropped out for a few seconds. The problem here is that the robot will carry on using the velocities created from the last joystick or keyboard input. I have therefore added a new node (*remote\_heartbeat\_node*) to the *rodney*package which is run on the remote workstation and publishes an heartbeat message. The *rodney*node running on the robot hardware monitors this message and if the robot is in manual mode and the message is not received for a one second the velocities will be set to zero.

Changes to the *rodney\_node.cpp* file.

We can now enable/disable the LIDAR from the joystick so we need a way to configure which button controls this functionality. Add the following line to the RodneyNode constructor.

nh\_.param("/controller/buttons/lidar\_enable", lidar\_enable\_select\_, 2);

In the joystickCallback function add the following "if" construct to request the LIDAR motor state be toggled if the joystick button in question is pressed.

// Button on controller selects to enable/disable the lidar function

if((manual\_locomotion\_mode\_ == true) && (msg->buttons[lidar\_enable\_select\_] == 1))

{

std\_msgs::String mission\_msg;

// Toggle the LIDAR on/off

mission\_msg.data = "J4";

mission\_pub\_.publish(mission\_msg);

last\_interaction\_time\_ = ros::Time::now();

}

Likewise if the 'l' key is pressed on the keyboard the motor state is toggled. Add the following "else if" construct to the keyboardCallBack function.

else if((msg->code == keyboard::Key::KEY\_l) && ((msg->modifiers & ~RodneyNode::SHIFT\_CAPS\_NUM\_LOCK\_) == 0))

{

if(manual\_locomotion\_mode\_ == true)

{

std\_msgs::String mission\_msg;

// Toggle the LIDAR on/off

mission\_msg.data = "J4";

mission\_pub\_.publish(mission\_msg);

last\_interaction\_time\_ = ros::Time::now();

}

}

Next the node needs to subscribe to the new heartbeat message by adding the following to the RodneyNode constructor.

remote\_heartbeat\_sub\_ = nh\_.subscribe("remote\_heartbeat", 1, &RodneyNode::remHeartbeatCallback, this);

 A callback function for when the message on this topic is received will store the time of the message.

// Callback for remote heartbeat

void RodneyNode::remHeartbeatCallback(const std\_msgs::Empty::ConstPtr& msg)

{

// Remote heartbeat received store the time

remote\_heartbeat\_time\_ = ros::Time::now();

}

In the sendTwist function we need to set the velocities to zero if its been more than one second since the last heartbeat message was received. Below is the complete new version of the sendTwist function.

void RodneyNode::sendTwist(void)

{

geometry\_msgs::Twist target\_twist;

// If in manual locomotion mode use keyboard or joystick data

if(manual\_locomotion\_mode\_ == true)

{

// Only allow stored keyboard or joystick values to set

// the velocities if the remote heartbeat is running

if((ros::Time::now() - remote\_heartbeat\_time\_).toSec() < 1.0)

{

// Publish message based on keyboard or joystick speeds

if((keyboard\_linear\_speed\_ == 0) && (keyboard\_angular\_speed\_ == 0))

{

// Use joystick values

target\_twist.linear.x = joystick\_linear\_speed\_;

target\_twist.angular.z = joystick\_angular\_speed\_;

}

else

{

// use keyboard values

target\_twist.linear.x = keyboard\_linear\_speed\_;

target\_twist.angular.z = keyboard\_angular\_speed\_;

}

}

else

{

// Lost connection with remote workstation so zero the velocities

target\_twist.linear.x = 0.0;

target\_twist.angular.z = 0.0;

}

}

else

{

// Use mission demands (autonomous)

target\_twist.linear.x = linear\_mission\_demand\_;

target\_twist.angular.z = angular\_mission\_demand\_;

}

// If not using the PID ramp to the target value.

if (false == pid\_enabled\_)

{

ros::Time time\_now = ros::Time::now();

// Ramp towards are required twist velocities

last\_twist\_ = rampedTwist(last\_twist\_, target\_twist, last\_twist\_send\_time\_, time\_now);

last\_twist\_send\_time\_ = time\_now;

// Publish the Twist message using the ramp value

twist\_pub\_.publish(last\_twist\_);

}

else

{

// Publish the Twist message using the target value

twist\_pub\_.publish(target\_twist);

}

}

Next we will add the *remote\_heartbeat\_node.cpp* file to the *rodney/src folder*. Remember this node will not be run on the robot hardware but on a remote workstation being used to manually drive the robot. This simple node just broadcasts the heartbeat message at 5Hz.

// This heartbeat node is not to be run on the robot platform but on a remote worksation

// when either the keyboard or joystick nodes are being used to teleop the robot. If the

// message sent by this node is missed for 1 second the robot will stop using the keyboard

// and joystick stored values to drive the motors.

#include <ros/ros.h>

#include <std\_msgs/Empty.h>

int main(int argc, char \*\*argv)

{

ros::init(argc, argv, "remote\_heartbeat");

ros::NodeHandle n;

ros::Publisher remote\_heartbeat\_pub = n.advertise<std\_msgs::Empty>("remote\_heartbeat", 1);

std::string node\_name = ros::this\_node::getName();

ROS\_INFO("%s started", node\_name.c\_str());

ros::Rate r(5); // 5Hz

std\_msgs::Empty beat;

while(ros::ok())

{

remote\_heartbeat\_pub.publish(beat);

ros::spinOnce();

r.sleep();

}

return 0;

}

One final change to the *rodney*package is to add a launch file that can be used to launch  the keyboard, joystick and heartbeat nodes all together. The file name is *remote.launch* and it is in the *rodney/launch* folder.

<?xml version="1.0" ?>

<launch>

<node pkg="joystick" type="joystick\_node" name="joystick\_node"/>

<node pkg="keyboard" type="keyboard" name="keyboard"/>

<node pkg="rodney" type="remote\_heartbeat\_node" name="remote\_heartbeat\_node" output="screen"/>

</launch>

**Updates to the Arduino Sketch**

When I started to think about the LIDAR I thought that it would be connected to the Arduino but with memory already running out in that device I decided to switch to a [Teensy](https://www.pjrc.com/teensy/). Teensy's are faster and contain much more memory than Arduino's but are compatible with Arduino software and libraries. You can download a plugin so that you can continue to use the Arduino IDE for development.

Although I didn't in the end connect the LIDAR to the Arduino ot Teensy I have upgraded Rodney with a Teensy for future expansion and so that I can make use of the faster microcontroller.

The sketch is now written in such a way that it can be used on either an Arduino Nano or a Teensy 3.5. The 3.5 was chosen as unlike the faster 3.6 the digital inputs are 5V tolerant.

The functionality of the sketch has not currently changed but I have made the following improvements.

When the old sketch was compiled for the Arduino Nano a low memory warning was issued. This low memory warning can be removed by making the message buffers smaller. This is only done if the sketch is compiled for the Arduino, if it is compiled for the Teensy the buffer sizes are left as they were.

Since the Teensy is faster I have increased the baud rate if the code is compiled for a Teensy and increased to rate at which the tacho topic is published.

In order to make the changes to the buffer and sizes and baud rate you need to make changes to the ros.h and ArduinoHardware.h files which are part of the ROS serial library. You could make the changes directly in the library but you would then subsequently lose the changes if you recompiled the library for example if you added a new message to the library. I have therefore recreated these files within the sketch folder.

*rodney\_control.ino*

/\*

\* This version:

\* - Controls upto four RC Servos on the servo topic

\* - Publishes the tacho on the tacho topic monitoring two motors with Hall sensors.

\*

\* The node subscribes to the servo topic and acts on a rodney\_msgs::servo\_array message.

\* This message contains two elements, index and angle. Index references the servos 0-3 and

\* angle is the angle to set the servo to 0-180.

\*

\*

\* The connections to a Teensy 3.5 are:

\* Pin 0 (INT) -> used for monitoring right motor speed

\* Pin 1 (Input) -> used for sensing right motor direction

\* Pin 3 (INT) -> used for monitoring left motor speed

\* Pin 4 (Input) -> used for sensing left motor direction

\* Pin 20 (PWM) -> servo indexed 3

\* Pin 21 (PWM) -> servo indexed 2

\* Pin 22 (PWM) -> servo indexed 1

\* PIN 23 (PWM) -> servo indexed 0

\*

\* The connections to a Nano are:

\* D2 (INT) -> used for monitoring right motor speed

\* D4 (Input) -> used for sensing right motor direction

\* D3 (INT) -> used for monitoring left motor speed

\* D7 (Input) -> used for sensing left motor direction

\* D10 (PWM) -> servo indexed 3

\* D5 (PWM) -> servo indexed 2

\* D6 (PWM) -> servo indexed 1

\* D9 (PWM) -> servo indexed 0

\*

\*/

#if defined(\_\_MK64FX512\_\_) || defined(\_\_MK66FX1M0\_\_)

#include <PWMServo.h> // Use PWMServo if using a Teensy

#else

#include <Servo.h>

#endif

// Use "ros.h" not <ros.h> so that by using our local version

// we can increase/decrease buffer size if required and

// increased the baud rate on faster boards.

#include "ros.h"

#include <servo\_msgs/servo\_array.h>

#include <tacho\_msgs/tacho.h>

void servo\_cb( const servo\_msgs::servo\_array& cmd\_msg);

void WheelSpeed0();

void WheelSpeed1();

#define LED\_PIN 13 // Onboard LED

#define GEAR\_BOX\_COUNTS\_PER\_REV 1440.0f

#if defined(\_\_MK64FX512\_\_) || defined(\_\_MK66FX1M0\_\_) // Teensy

// Define the period in milliseconds between tacho messages

#define TACHO\_PERIOD\_MS 25 // Publish at 40Hz

// Define the PWM pins that the other servos are connected to

#define SERVO\_0 23

#define SERVO\_1 22

#define SERVO\_2 21

#define SERVO\_3 20

// Define pins used for two Hall sensors

#define ENCODER0\_PINA 0 // Interrupt

#define ENCODER0\_PINB 1 // Digital pin

#define ENCODER1\_PINA 3 // Interrupt

#define ENCODER1\_PINB 4 // Digital pin

PWMServo servo0;

PWMServo servo1;

PWMServo servo2;

PWMServo servo3;

#else // Arduino Nano

// Define the period in milliseconds between tacho messages

#define TACHO\_PERIOD\_MS 50 // Publish at 20Hz

// Define the PWM pins that the other servos are connected to

#define SERVO\_0 9

#define SERVO\_1 9

#define SERVO\_2 5

#define SERVO\_3 10

// Define pins used for two Hall sensors

#define ENCODER0\_PINA 2 // Interrupt 0

#define ENCODER0\_PINB 4 // Digital pin

#define ENCODER1\_PINA 3 // Interrupt 1

#define ENCODER1\_PINB 7 // Digital pin

Servo servo0;

Servo servo1;

Servo servo2;

Servo servo3;

#endif

tacho\_msgs::tacho tachoMsg;

ros::NodeHandle nh;

ros::Publisher tachoPub("tacho", &tachoMsg);

ros::Subscriber<servo\_msgs::servo\_array> subServo("servo", servo\_cb);

byte encoder0PinALast;

byte encoder1PinALast;

volatile int encoder0Count; // Number of pulses

volatile int encoder1Count; // Number of pulses

volatile boolean encoder0Direction; //Rotation direction

volatile boolean encoder1Direction; //Rotation direction

unsigned long publisherTime;

unsigned long currentTime;

unsigned long lastTime;

void setup()

{

nh.initNode();

nh.advertise(tachoPub);

nh.subscribe(subServo);

// Attach servos

servo0.attach(SERVO\_0); //attach it to the pin

servo1.attach(SERVO\_1);

servo2.attach(SERVO\_2);

servo3.attach(SERVO\_3);

servo0.write(90);

servo1.write(120);

servo2.write(90);

servo3.write(90);

encoder0Direction = true; // default is forward

encoder1Direction = true;

encoder0Count = 0;

encoder1Count = 0;

pinMode(ENCODER0\_PINB, INPUT);

pinMode(ENCODER1\_PINB, INPUT);

// Attach the interrupts for the Hall sensors

attachInterrupt(digitalPinToInterrupt(ENCODER0\_PINA), WheelSpeed0, CHANGE);

attachInterrupt(digitalPinToInterrupt(ENCODER1\_PINA), WheelSpeed1, CHANGE);

// Turn on the onboard LED to show we are running

pinMode(LED\_PIN, OUTPUT);

digitalWrite(LED\_PIN, HIGH);

}

void loop()

{

// Is it time to publish the tacho message

if(millis() > publisherTime)

{

float deltaTime;

currentTime = micros();

deltaTime = (float)(currentTime - lastTime)/1000000.0;

// Right wheel speed

tachoMsg.rwheelrpm = (((((float)encoder0Count)/2.0f)/deltaTime)/GEAR\_BOX\_COUNTS\_PER\_REV)\*60.0f;

encoder0Count = 0;

// Left wheel speed

tachoMsg.lwheelrpm = (((((float)encoder1Count)/2.0f)/deltaTime)/GEAR\_BOX\_COUNTS\_PER\_REV)\*60.0f;

encoder1Count = 0;

lastTime = currentTime;

tachoPub.publish(&tachoMsg);

publisherTime = millis() + TACHO\_PERIOD\_MS;

}

nh.spinOnce();

}

// Callback for when servo array message received

void servo\_cb( const servo\_msgs::servo\_array& cmd\_msg)

{

/\* Which servo to drive \*/

switch(cmd\_msg.index)

{

case 0:

servo0.write(cmd\_msg.angle); //set servo 0 angle, should be from 0-180

break;

case 1:

servo1.write(cmd\_msg.angle); //set servo 1 angle, should be from 0-180

break;

case 2:

servo2.write(cmd\_msg.angle); //set servo 2 angle, should be from 0-180

break;

case 3:

servo3.write(cmd\_msg.angle); //set servo 3 angle, should be from 0-180

break;

default:

nh.logdebug("Error incorrect servo index");

break;

}

}

// ISR

void WheelSpeed0()

{

int state = digitalRead(ENCODER0\_PINA);

if((encoder0PinALast == LOW) && (state == HIGH))

{

int val = digitalRead(ENCODER0\_PINB);

if(val == LOW && encoder0Direction)

{

encoder0Direction = false; // Reverse

}

else if (val == HIGH && !encoder0Direction)

{

encoder0Direction = true; // Forward

}

}

encoder0PinALast = state;

if(!encoder0Direction)

{

encoder0Count++;

}

else

{

encoder0Count--;

}

}

// ISR

void WheelSpeed1()

{

int state = digitalRead(ENCODER1\_PINA);

if((encoder1PinALast == LOW) && (state == HIGH))

{

int val = digitalRead(ENCODER1\_PINB);

if(val == LOW && encoder1Direction)

{

encoder1Direction = false; // Reverse

}

else if (val == HIGH && !encoder1Direction)

{

encoder1Direction = true; // Forward

}

}

encoder1PinALast = state;

if(!encoder1Direction)

{

encoder1Count++;

}

else

{

encoder1Count--;

}

}

*ros.h*

#ifndef \_ROS\_H\_

#define \_ROS\_H\_

// As we are no longer including any .h file in the root of the

// ros\_lib library the IDE can't find ros/node\_handle.h

// Add the dummy.h empty file in the ros\_lib root

#include <dummy.h>

#include <ros/node\_handle.h>

#include "ArduinoHardware.h"

namespace ros

{

#if defined(\_\_MK64FX512\_\_) || defined(\_\_MK66FX1M0\_\_)

// Teensy 3.5 or 6.3

typedef NodeHandle\_<ArduinoHardware, 25, 25, 512, 512> NodeHandle;

#elif defined(\_\_AVR\_ATmega328P\_\_)

// arduino Nano

// 10 publishers, 15 subscribers, 128 bytes input buffer and 256 bytes output buffer

typedef NodeHandle\_<ArduinoHardware, 10, 15, 128, 256> NodeHandle;

#else

typedef NodeHandle\_<ArduinoHardware> NodeHandle; // default 25, 25, 512, 512

#endif

}

#endif

*ArduinoHardware.h*

#ifndef ROS\_ARDUINO\_HARDWARE\_H\_

#define ROS\_ARDUINO\_HARDWARE\_H\_

#if ARDUINO>=100

#include <Arduino.h> // Arduino 1.0

#else

#include <WProgram.h> // Arduino 0022

#endif

#if defined(\_\_MK20DX128\_\_) || defined(\_\_MK20DX256\_\_) || defined(\_\_MK64FX512\_\_) || defined(\_\_MK66FX1M0\_\_) || defined(\_\_MKL26Z64\_\_)

#if defined(USE\_TEENSY\_HW\_SERIAL)

#define SERIAL\_CLASS HardwareSerial // Teensy HW Serial

#else

#include <usb\_serial.h> // Teensy 3.0 and 3.1

#define SERIAL\_CLASS usb\_serial\_class

#endif

#elif defined(\_SAM3XA\_)

#include <UARTClass.h> // Arduino Due

#define SERIAL\_CLASS UARTClass

#elif defined(USE\_USBCON)

// Arduino Leonardo USB Serial Port

#define SERIAL\_CLASS Serial\_

#elif (defined(\_\_STM32F1\_\_) and !(defined(USE\_STM32\_HW\_SERIAL))) or defined(SPARK)

// Stm32duino Maple mini USB Serial Port

#define SERIAL\_CLASS USBSerial

#else

#include <HardwareSerial.h> // Arduino AVR

#define SERIAL\_CLASS HardwareSerial

#endif

class ArduinoHardware {

public:

#if defined(\_\_MK64FX512\_\_) || defined(\_\_MK66FX1M0\_\_)

ArduinoHardware(SERIAL\_CLASS\* io , long baud= 500000){

iostream = io;

baud\_ = baud;

}

ArduinoHardware()

{

#if defined(USBCON) and !(defined(USE\_USBCON))

/\* Leonardo support \*/

iostream = &Serial1;

#elif defined(USE\_TEENSY\_HW\_SERIAL) or defined(USE\_STM32\_HW\_SERIAL)

iostream = &Serial1;

#else

iostream = &Serial;

#endif

baud\_ = 500000;

}

#else // Not a Teensy

ArduinoHardware(SERIAL\_CLASS\* io , long baud= 57600){

iostream = io;

baud\_ = baud;

}

ArduinoHardware()

{

#if defined(USBCON) and !(defined(USE\_USBCON))

/\* Leonardo support \*/

iostream = &Serial1;

#elif defined(USE\_TEENSY\_HW\_SERIAL) or defined(USE\_STM32\_HW\_SERIAL)

iostream = &Serial1;

#else

iostream = &Serial;

#endif

baud\_ = 57600;

}

#endif // defined(\_\_MK64FX512\_\_) || defined(\_\_MK66FX1M0\_\_)

ArduinoHardware(ArduinoHardware& h){

this->iostream = h.iostream;

this->baud\_ = h.baud\_;

}

void setBaud(long baud){

this->baud\_= baud;

}

int getBaud(){return baud\_;}

void init(){

#if defined(USE\_USBCON)

// Startup delay as a fail-safe to upload a new sketch

delay(3000);

#endif

iostream->begin(baud\_);

}

int read(){return iostream->read();};

void write(uint8\_t\* data, int length){

for(int i=0; i<length; i++)

iostream->write(data[i]);

}

unsigned long time(){return millis();}

protected:

SERIAL\_CLASS\* iostream;

long baud\_;

};

#endif

**Robot Hardware**

A full size image of the current circuit diagram is available in the diagrams *zip* folder along with a full size copy of the image from rqt\_graph showing all the nodes and topics.

[??? add optimized circuit image]

A complete bill of material for the project so far is available here. [??? add new link]

In part 1 of the article, I referenced the Ubiquity Robot Image which I use on the Raspberry Pi. Instructions on how to install the image, install extra software and configure it for the project are available here [ ??? new link required].

**Using the Code**

As usual, I'll run the code on the robot hardware and run the test tools and manual control nodes and test on a Linux PC. I'll refer to this PC as the workstation in the details below.

At this stage we will use the tool rviz to set the navigation task goals, in the next part we will add code so that this is done programmatically.

**Building the ROS Packages on the Pi (Robot Hardware)**

If not already done, create a catkin workspace on the Raspberry Pi and initialise it with the following commands:

$ mkdir -p ~/rodney\_ws/src

$ cd ~/rodney\_ws/

$ catkin\_make

Copy the packages face\_recognition, face\_recognition\_msgs, head\_control, pan\_tilt, rodney\_missions, servo\_msgs, speech, thunderborg and tacho\_msgs into the *~/rodney\_ws/src* folder.

Download the packages ros-keyboard (from <https://github.com/lrse/ros-keyboard>) and rplidar\_ros (from <https://github.com/Slamtec/rplidar_ros>) and copy them both into the *~/rodney\_ws/src* folder.

Build the code with the following commands:

$ cd ~/rodney\_ws/

$ catkin\_make

Check that the build completes without any errors.

You will also need to compile and download the sketch to either the Arduino Nano or a Teensy 3.5

**Building the ROS Packages on the Workstation**

On the workstation, we want to run the keyboard, joystick and heartbeat nodes so that we can control the actual robot hardware remotely.

Create a workspace with the following commands:

$ mkdir -p ~/test\_ws/src

$ cd ~/test\_ws/

$ catkin\_make

Copy the packages rodney, joystick, and ros-keyboard(from <https://github.com/lrse/ros-keyboard>) into the *~/test\_ws/src* folder and then build the code with the following commands:

$ cd ~/test\_ws/

$ catkin\_make

Check that the build completes without any errors.

**Tip**

When running ROS code and tools on a workstation and the Raspberry Pi, there can be a lot of repeat typing of commands at a number of terminals. In the next sections, I have included the full commands to type but here are a few tips that can save you all that typing.

On the Raspberry Pi, to save typing "*source devel/setup.bash*", I have added it to the *.bashrc* file for the Raspberry Pi.

$ cd ~/

$ nano .bashrc

Then add "*source /home/ubuntu/rodney\_ws/devel/setup.bash*" to the end of the file, save and exit.

When running test code and tools on the workstation, it also needs to know where the ROS master is so I have added the following to the *.bashrc* file for the workstation.

alias rodney='source ~/test\_ws/devel/setup.bash; \

export ROS\_MASTER\_URI=http://ubiquityrobot:11311'

Then by just typing "rodney" at a terminal, the two commands are run and a lot of typing is saved.

You can also save some typing as some ROS tools support TAB completion. For example, type rosrun rosserial\_ and then press the tab key to auto complete rosrun rosserial\_python.

**Creating a Map**

On the robot hardware, run the following commands to start all the current nodes in the system with the exception of the navigation stack:

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch no\_nav:=True

On the workstation, run the following commands to start the remote control nodes:

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ roslaunch rodney remote.launch

A small window whose title is "**ROS keyboard input**" should be running. When entering keyboard strokes ensure the small window has the focus.

Next we will start recording the transforms and laser scan messages so that we can show how to create a map from recorded data. In a terminal start the recording with the following commands:

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rosbag record -O data.bag /scan /tf

Here we are going to start slam\_gmapping so that we can see a live map being created. I'll also limit the size of the map. In a terminal on the workstation run the following commands:

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rosparam set slam\_gmapping/xmax 10

$ rosparam set slam\_gmapping/ymax 10

$ rosparam set slam\_gmapping/xmin -10

$ rosparam set slam\_gmapping/ymin -10

$ rosparam set slam\_gmapping/delta 0.05

$ rosrun gmapping slam\_gmapping

In another terminal on the workstation start rviz with the following commands:

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ roslaunch rodney rviz.launch

Configure rviz so that:

* Fixed frame is map
* LaserScan is displaying the /scan topic
* TF is displaying the base\_link
* Map is displaying the /map topic

Using the joystick and/or keyboard enter manual mode, ensure the LIDAR motor is running and manually drive the robot around its world. Go slow and visit each location at least twice. The map being created should be visible on rviz.

Once you have created the map press **Ctrl-C** in the terminal running rosbag to finish storing the message.

We can now save the map that is visible in rviz, to disk. With slam\_gmapping still running enter the following command in the terminal that was running rosbag:

$ rosrun map\_server map\_saver -f my\_first\_map

This will result in two files been saved, *my\_first\_map.yaml* and *my\_first\_map.pgm*

You can at this point if you wish regenerate the map from the rosbag file with different gmapping parameters. Here we don't want any of the existing nodes running on the workstation so shutdown and close all the terminals. You can also shutdown the robot.

At the workstation we need a ROS master running (in the previous setup this was running automatically on the robot hardware), enter the following in a terminal:

$roscore

In another terminal use rosparam to set the required gmapping parameters and then type the following:

$ rosparam set use\_sim\_time\_true

$ rosrun gmapping slam\_gmapping

In another terminal playback the rosbag recorded previosusly with the following commands:

$ rosbag play --clock data.bag

Then sit bag whilst gmapping creates the map. If you wish you could start rviz (without the export ROS\_MASTER\_URI command) and watch the map being created. Once all the messages have been played back store the new map to disk with the following command:

$ rosrun map\_server map\_saver -f my\_second\_map

You can play with the gampping config parameters here as much as you like and keep playing back the recoded bag file to see the differences the parameters have on the generated map.

The following video shows map creation

**Autonomous Navigation**

Now the bit we have all been waiting for, autonomous navigation.

 On the robot hardware, run the following commands to start all the current nodes in the system including the navigation stack. I'm going to use my default map but you can set the map by adding "map\_file:=my\_first\_map" to the end of the roslaunch command.

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch

On the workstation, run the following commands to start the remote control nodes:

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ roslaunch rodney remote.launch

A small window whose title is "**ROS keyboard input**" should be running. When entering keyboard strokes ensure the small window has the focus.

In another terminal on the workstation start rviz with the following commands:

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ roslaunch rodney rviz.launch

**Localising the Robot**

Configure rviz to display the robot model, laser scan, map and pose estimates. Also ensure that the map is the fixed frame.

We can see from the display that the laser scan does not match the map, the pose estimates are spread around and I know the model is not facing the same direction as the real robot. So before we give the robot a navigational goal we need to improve its localisation.

The image below shows a poorly localised robot. The red lines are the laser scan and the green arrows are the pose estimates.

The first operation we will carry out is to give the robot an improved localisation using rviz. Click the "2D Pose Estimate" button, estimate the real location and pose of the robot and click/drag the large green arrow on the map to set the initial pose. You can keep doing this until you get the laser scan close to matching the map.

We now have a good initial pose but the pose estimates are still out. We can improve these by driving the robot around in manual mode. Spinning on the spot is a good manoeuvrer to conduct. Whilst moving the robot you should see the pose estimates converging on the robots position.

**Setting a Navigation Goal**

We are now ready to send the robot on its way but lets first take a look at the costmaps that will be used for planning the route. Select Global Planning in rviz to display the Global Costmap. For the Global Costmap I like to select "Draw Behind" so that the map is washed out behind the main map.

From the costmap you can see the open spaces the planner will try and use and the risker places like up against a wall.

Now the Global Costmap was constructed from the main map and will be used to plan an ideal route, but the actual movement of the robot will be governed by the Local Costmap and this will be generated on the fly as the sensor data arrives. This will allow the robot to avoid objects that weren't there when the map was created, i.e. a sleeping family pet.

Select Local Planning in rviz to display the Local Costmap. I like to have this map superimposed on top of the main map.

Now there is a route which is possible to navigate from the room at the bottom of the map to the room in the top of the map, but I can see that the costmap has become distorted by the errors in the odom from when I moved the robot to improve the pose estimates.

Open a new terminal on the wokstation and type the following commands to reset the costmaps:

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rosservice call /move\_base/clear\_costmaps

A new more accurate Local Costmap will be displayed in rviz.

Now set the target goal pose by clickin the "2D Nav Goal" button, click/drag the large green arrow on the map to set the goal. Now I have deliberately left the robot in manual mode so it will not move and this gives us a chance to examine the global plan which is shown as a thin green line in the image below.

We can put the robot into to autonomous mode by giving the **ROS keyboard input** window the focus and pressing the "1" key (not on the numeric keypad). This is the request to run mission 1 which is currently empty so all it does is take the robot out of manual mode and means that the velocities generated by the navigation stack and not the joystick/keyboard will be sent to the thunderorg node which controls the motors.

Hopefully the actual robot will navigate to the goal pose and you can monitor the progress on rviz. In the image below the robot has reached the selected goal.

Now if you want to give it another goal to move to reset the costmaps first.

??? video

**Points of Interest.**

???

**History**

??? don't forget zip files for code, 3D prints, full size circuit and full size ros graph

??? don't forget links to BOM

Keep a running update of any changes or improvements you've made here.