## 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 was 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 and add an IMU to improve the odometry. In the process of adding the IMU, I'm also going to replace the Arduino Nano with a Teensy 3.5.

## Adding a LIDAR

The most common method for autonomous navigation in ROS requires topic subscription to both the /scan and /tf topics. The /tf topic is used to obtain the odom transform and we started to broadcast this from the ekf\_localization\_node in the last article. The /scan topic contains data from a laser scan device.

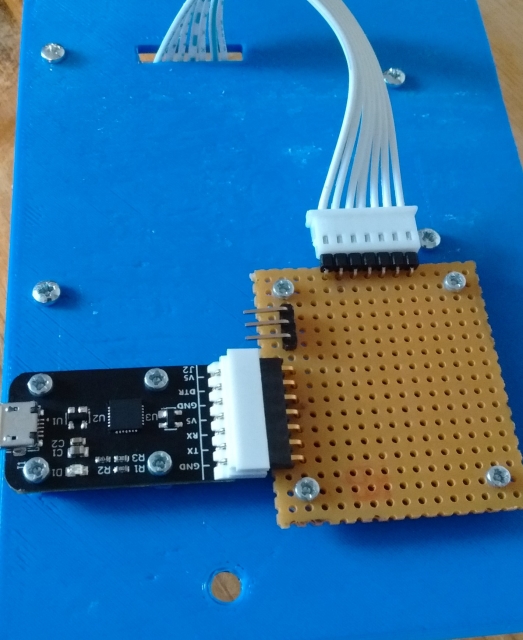
You can see some large and very expensive 360 degree scanning devices on autonomous cars that are under development, but as Rodney is going to be confined to the house, we can considerably step down the price range. Slamtec produce a number of spinning LIDARs and their [RPLidar A1](https://www.slamtec.com/en/Lidar/A1) 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 is transmitting.

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 data 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 signal 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 serial device powered from the Raspberry Pi via the 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 who 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 covering the neck area. We will use it to remove 10 degrees either side of the 180 degree mark.

To add the code for the RPLidar, 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 and will 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.

Hide   Copy Code

scan\_filter\_chain:

- name: angle

type: laser\_filters/LaserScanAngularBoundsFilter

params:

lower\_angle: -2.96706

upper\_angle: 2.96706

The lower and up angles are in radians which will restrict our field from 0 to -170 and 0 to +170 degrees.

To load these parameters onto the parameter server and to start the scan filter, we will add the following lines to our rodney.launch file in the rodney/launch folder. We are going to be making more changes to the launch file so I'll show it in its entirety later in this article.

Hide   Copy Code

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

name="scan\_to\_scan\_filter\_chain" output="screen">

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

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

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

</node>

You can see from the above that we needed 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. This is a common name and can be confusing to the system if you have more than one device with the ttyUSBn name.

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.

Hide   Copy Code

# 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 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 for you.

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

Hide   Copy Code

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

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

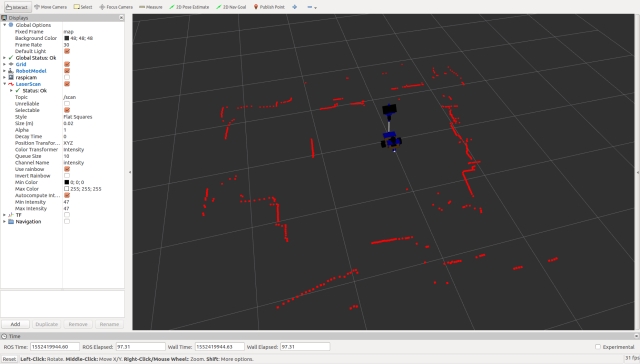
<param name="serial\_baudrate" type="int" value="115200"/>

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

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

</node>

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 visualised in rviz.



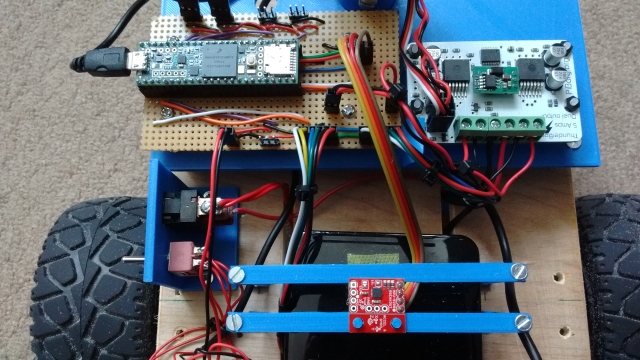
## IMU (Inertial Measurement Unit)

In part 5, we started to broadcast raw odometry data derived from the motor encoders and included the ekf\_localization\_node which we said would be used to fuse the raw odometry with IMU data to improve the odometry of the robot.

For the IMU, we are going to use a SparkFun MPU-9250 breakout board.

I want to connect the IMU to the microcontroller and not directly to the Raspberry Pi. As the Arduino Nano is already running out of memory, we need an alternative solution. You could add a second Nano to the project or use one of the larger Arduino boards, but I have decided on my project I'm going to replace the Nano with a Teensy 3.5. 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. I chose a Teensy 3.5 as unlike the faster 3.6, the digital inputs are 5V tolerant.

The image below shows the IMU (MPU-9250 breakout board) in the lower part of the image and the Teensy is shown top left.



### Updates to the Arduino Sketch

Since the Teensy is faster, we will also take the opportunity to increase the baud rate of the serial interface to the Raspberry Pi. As the robot develops, we may also require larger message between the Teensy and the Pi. In order to make the changes to the buffer 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.

The sketch still incudes the code for the servos and the hall sensors but has been updated to include the IMU functionality.

In the setup function, we now also ensure we can talk to the IMU and call setup procedures from the SparkFun MPU-9250 9 DOF IMU Breakout library which I installed into the Arduino IDE with the Library Manager. Note here that we also setup the magnetometer part of the IMU although we are not currently broadcasting the mag data on any topic. At the end of the setup function, we now only turn on the on-board LED if we successfully setup the IMU.

In the loop function, I have added code to log a message if we failed to set up the IMU. In the past, I found that any log calls made in the setup part of the sketch don't get logged. During each call to loop, we check to see if the IMU registers contain new data, if so we read the accelerometer, gyro and magnetometer data. If it's time to publish the IMU data, we form a sensor\_msgs/Imu [message](http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Imu.html)and broadcast it on the imu/data\_raw topic.

rodney\_control.ino

Hide   Shrink Image 5 for Rodney - A Long Time Coming Autonomous Robot (Part 6)   Copy Code

#include <PWMServo.h> // Use PWMServo on Teensy

#include <MPU9250.h>

*// 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>

#include <sensor\_msgs/Imu.h>

#include <sensor\_msgs/MagneticField.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

*// 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 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;

#define G\_TO\_MS2 9.80665

#define I2Cclock 400000

#define I2Cport Wire

#define MPU9250\_ADDRESS MPU9250\_ADDRESS\_AD0

MPU9250 myIMU(MPU9250\_ADDRESS, I2Cport, I2Cclock);

tacho\_msgs::tacho tachoMsg;

sensor\_msgs::Imu imuMsg;

ros::NodeHandle nh;

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

ros::Publisher imuPub("imu/data\_raw", &imuMsg);

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

bool imuTestPassed;

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;

char imu\_link[] = "imu";

void setup()

{

Wire.begin();

nh.initNode();

nh.advertise(tachoPub);

nh.advertise(imuPub);

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);

imuTestPassed = true;

*// Read the WHO\_AM\_I register of the IMU, this is a good test of communication*

byte c = myIMU.readByte(MPU9250\_ADDRESS, WHO\_AM\_I\_MPU9250);

if(c == 0x71) *// WHO\_AM\_I should always be 0x71*

{

*// Start by performing self test*

myIMU.MPU9250SelfTest(myIMU.selfTest);

for(int i = 0; i < 6; i++)

{

if(abs(myIMU.selfTest[i]) > 14.0f)

{

imuTestPassed = false;

}

}

*// Calibrate gyro and accelerometers, load biases in bias registers*

myIMU.calibrateMPU9250(myIMU.gyroBias, myIMU.accelBias);

*// Initialize device for active mode read of acclerometer, gyroscope, and temperature*

myIMU.initMPU9250();

*// Read the WHO\_AM\_I register of the magnetometer, this is a good test of communication*

byte d = myIMU.readByte(AK8963\_ADDRESS, WHO\_AM\_I\_AK8963);

if(d == 0x48)

{

*// Get magnetometer calibration from AK8963 ROM*

*// Initialize device for active mode read of magnetometer*

myIMU.initAK8963(myIMU.factoryMagCalibration);

*// Get sensor resolutions, only need to do this once*

myIMU.getAres();

myIMU.getGres();

myIMU.getMres();

}

else

{

imuTestPassed = false;

}

}

else

{

imuTestPassed = false;

}

if(imuTestPassed == true)

{

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

pinMode(LED\_PIN, OUTPUT);

digitalWrite(LED\_PIN, HIGH);

}

}

void loop()

{

static bool setup = false;

if(setup == false)

{

*// Log only gets reported in loop*

nh.loginfo("Teensy code started");

if(imuTestPassed == false)

{

nh.loginfo("IMU self test failed");

}

setup = true;

}

*// 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;

}

*// IMU*

if(imuTestPassed == true)

{

*// Check to see if all data registers have new data*

if (myIMU.readByte(MPU9250\_ADDRESS, INT\_STATUS) & 0x01)

{

myIMU.readAccelData(myIMU.accelCount); *// Read the x/y/z adc values*

*// Now we'll calculate the accleration value into actual g's*

*// This depends on scale being set*

myIMU.ax = (float)myIMU.accelCount[0] \* myIMU.aRes;

myIMU.ay = (float)myIMU.accelCount[1] \* myIMU.aRes;

myIMU.az = (float)myIMU.accelCount[2] \* myIMU.aRes;

myIMU.readGyroData(myIMU.gyroCount); *// Read the x/y/z adc values*

*// Calculate the gyro value into actual degrees per second*

*// This depends on scale being set*

myIMU.gx = (float)myIMU.gyroCount[0] \* myIMU.gRes;

myIMU.gy = (float)myIMU.gyroCount[1] \* myIMU.gRes;

myIMU.gz = (float)myIMU.gyroCount[2] \* myIMU.gRes;

myIMU.readMagData(myIMU.magCount); *// Read the x/y/z adc values*

*// Reading mag data but not currently publishing it*

*// Calculate the magnetometer values in milliGauss*

*// Include factory calibration per data sheet and user environmental corrections*

*// Get actual magnetometer value, this depends on scale being set*

myIMU.mx = (float)myIMU.magCount[0] \* myIMU.mRes

\* myIMU.factoryMagCalibration[0] - myIMU.magBias[0];

myIMU.my = (float)myIMU.magCount[1] \* myIMU.mRes

\* myIMU.factoryMagCalibration[1] - myIMU.magBias[1];

myIMU.mz = (float)myIMU.magCount[2] \* myIMU.mRes

\* myIMU.factoryMagCalibration[2] - myIMU.magBias[2];

}

*// Is it time to publish IMU data*

myIMU.delt\_t = millis() - myIMU.count;

if (myIMU.delt\_t > 50)

{

*// IMU*

imuMsg.header.frame\_id = imu\_link;

imuMsg.header.stamp = nh.now();

*// We are not providing orientation so the*

*// first element of the this matrix should be -1*

imuMsg.orientation\_covariance[0] = -1;

imuMsg.angular\_velocity.x = myIMU.gx \* DEG\_TO\_RAD;

imuMsg.angular\_velocity.y = myIMU.gy \* DEG\_TO\_RAD;

imuMsg.angular\_velocity.z = myIMU.gz \* DEG\_TO\_RAD;

*// angular velocity covariance*

imuMsg.angular\_velocity\_covariance[0] = 0.003;

imuMsg.angular\_velocity\_covariance[4] = 0.003;

imuMsg.angular\_velocity\_covariance[8] = 0.003;

imuMsg.linear\_acceleration.x = myIMU.ax \* G\_TO\_MS2;

imuMsg.linear\_acceleration.y = myIMU.ay \* G\_TO\_MS2;

imuMsg.linear\_acceleration.z = myIMU.az \* G\_TO\_MS2;

*// linear acceleration covariance*

imuMsg.linear\_acceleration\_covariance[0] = 0.1;

imuMsg.linear\_acceleration\_covariance[4] = 0.1;

imuMsg.linear\_acceleration\_covariance[8] = 0.1;

imuPub.publish(&imuMsg);

myIMU.count = millis();

}

}

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

Hide   Copy Code

#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

Hide   Shrink Image 6 for Rodney - A Long Time Coming Autonomous Robot (Part 6)   Copy Code

#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

### Serial Node Baud Rate

In the ArduinoHardware.h file, we have changed the baud rate of the serial port to 500000 baud. We need to reflect this change at the Raspberry Pi end when we launch the ROS serial node. I have made another change to the rodney.launch file to set the baud rate.

Hide   Copy Code

!-- 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>

### IMU Calibration

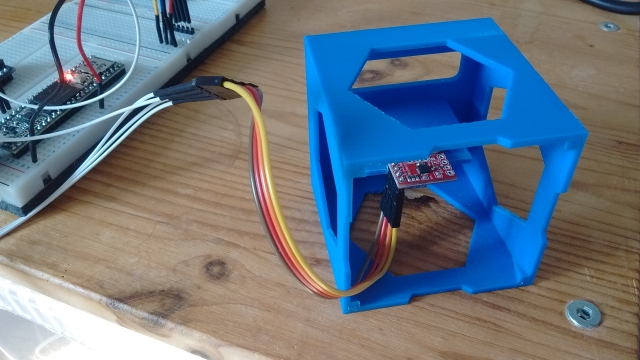
When we set up the IMU, it includes loading some factory calibration values but we can improve the IMU data by including extra calibration.

For this, we are going to use a ROS package available from [this GitHub site](https://github.com/dpkoch/imu_calib). We will also fork the repository and make some code changes of our own. The unchanged package contains two nodes. The first of these computes the accelerometer calibration parameters and saves them to a YAML file. We only need to run this once and as we have to place the IMU into the six orientations with fairly accurate positioning, it is best to do this before fitting the IMU to the robot. The second node uses the calibration file created by the first to apply the calibration to an uncalibrated IMU topic to produce a calibrated IMU topic. The second node can also optionally calculate the gyro biases at startup and then subtract these values from the raw data. Although the setup of the IMU inside the sketch also does this, I found that I received better results if this option was enabled.

In addition, I'm going to add a change to the second node. Although we will have calibrated the IMU, once installed on the robot, it may not be exactly perpendicular to the ground. This will result in accelerometer drift. Now as I only expect to use Rodney in the home and don't expect him to climb or descend any inclines, we can also null out this offset.

To carry out the one time calibration and in order to get the IMU into a fairly accurate position along each of its axes, I installed it into a [calibration cube](https://www.thingiverse.com/thing:2841844) which I 3D printed. Although the cube was for a different model of IMU with a little bit of filling, I was able to install my IMU.

The image shows the IMU installed in the calibration cube. The IMU is being controlled from a spare Teensy but you can use the one installed in the robot.



The calibration node do\_calib expects the IMU data to be broadcast in the topic imu. To launch the ROS serial node and to remap the topic, I have created a launch file, cal\_imu.launch which is in the rodney/launch folder.

Hide   Copy Code

<?xml version="1.0" ?>

<launch>

*<!-- Teensy.*

*Use the defaults /dev/ttyACM0 (or teensy if dev rules updated) and 500000 -->*

<arg name="serial\_port" default="/dev/ttyACM0"/>

<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)"/>

<remap from="/imu/data\_raw" to="imu"/>

</node>

*<!-- now "rosrun imu\_calib do\_calib" in a shell -->*

</launch>

Once the serial node is running, in another terminal, type the following command:

Hide   Copy Code

$ rosrun imu\_calib do\_calib

You then follow the instructions given on the terminal. Once the calibration file is produced, copy it to the rodney/config folder.

The changes I have made to the imu\_calib package are confined to the apply\_calib .cpp and .h files.

To the constructor, I added the reading of parameters from the parameters server so that this functionality can be enabled or disabled.

Hide   Copy Code

nh\_private.param<bool>("null\_accelerometer", null\_accelerometer\_, true);

nh\_private.param<int>("null\_accelerometer\_samples", null\_accelerometer\_samples\_, 100);

In the function rawImuCallback, I added code to calculate the mean accelerometer offsets at startup.

Hide   Shrink Image 8 for Rodney - A Long Time Coming Autonomous Robot (Part 6)   Copy Code

if(null\_accelerometer\_ == true)

{

ROS\_INFO\_ONCE("Nulling accelerometer; do not move the IMU");

*// Recursively compute mean accelerometer measurements from corrected acceleration readings*

sensor\_msgs::Imu corrected = \*raw;

accel\_sample\_count\_++;

calib\_.applyCalib(raw->linear\_acceleration.x,

raw->linear\_acceleration.y, raw->linear\_acceleration.z,

&corrected.linear\_acceleration.x, &corrected.linear\_acceleration.y,

&corrected.linear\_acceleration.z);

accel\_bias\_x\_ = ((accel\_sample\_count\_ - 1) \*

accel\_bias\_x\_ + corrected.linear\_acceleration.x) / accel\_sample\_count\_;

accel\_bias\_y\_ = ((accel\_sample\_count\_ - 1) \*

accel\_bias\_y\_ + corrected.linear\_acceleration.y) / accel\_sample\_count\_;

accel\_bias\_z\_ = ((accel\_sample\_count\_ - 1) \* accel\_bias\_z\_ +

(corrected.linear\_acceleration.z-9.80665)) / accel\_sample\_count\_;

if (accel\_sample\_count\_ >= null\_accelerometer\_samples\_)

{

ROS\_INFO("Nulling accelerometers complete! (bias = [%.3f, %.3f, %.3f])",

accel\_bias\_x\_, accel\_bias\_y\_, accel\_bias\_z\_);

null\_accelerometer\_ = false;

}

}

These values are then subtracted from the calibration corrected values.

Hide   Copy Code

corrected.linear\_acceleration.x -= accel\_bias\_x\_;

corrected.linear\_acceleration.y -= accel\_bias\_y\_;

corrected.linear\_acceleration.z -= accel\_bias\_z\_;

The updated package is available in the code zip file under the imu\_calib folder.

The following was added to the rodney.launch file to launch the updated node and to pass the location of the calibration file.

Hide   Copy Code

*<!-- Add calibration to raw imu data -->*

<node pkg="imu\_calib" type="apply\_calib" name="imu\_calib" output="screen">

<param name="calib\_file" value="$(find rodney)/config/imu\_calib.yaml"/>

</node>

Now in the previous part of this article, we added the ekf\_localization\_node from the robot\_localization package. This node will fuse the IMU data with the raw odometry data to produce the odometry data which will be used by the navigation system I'll introduce in the next article.

The configuration of this data fusing is contained in the robot\_localization.yaml file which can be found in the rodney/config folder. We briefly discussed this file in the last article but it is worth adding some detail here.

The data in the configuration matrix represents the following:

Hide   Copy Code

[ x position, y position, z position,

roll, pitch, yaw,

x velocity, y velocity, z velocity,

roll velocity, pitch velocity, yaw velocity,

x acceleration, y acceleration, z, acceleration]

A "true" value in the matrix means the data in that position will be used by the Extended Kalman Filter.

Now for the raw odometry data I have:

Hide   Copy Code

odom0\_config: [false, false, false,

false, false, false,

true, true, false,

false, false, true,

false, false, false]

Although the odometry message also contains the position data, it was derived from the same source as the velocity data so there is no point in using the data twice. It could also be argued that since we always set the y velocity to zero that we could set this matrix value to false. If, however, you want to include the y velocity from the IMU, it helps to cancel drift from the IMU by including this parameters in the odom matrix.

Now there are two different configurations that I have tried for the IMU matrix. Firstly one which uses just the yaw velocity from the IMU data.

Hide   Copy Code

imu0\_config: [false, false, false,

false, false, false,

false, false, false,

false, false, true,

false, false, false]

You can also try a configuration which also uses x and y velocity but you need to ensure that you have reduced any noise/error on the data first, or else the odometry will be subject to x and/or y drift.

Hide   Copy Code

imu0\_config: [false, false, false,

false, false, false,

true, true, false,

false, false, true,

false, false, false]

## Static Transforms

In the last part, we introduced a static transform for the height of the robot base from the ground. This latched transform broadcast was setup from the rodney.launch file with the following:

Hide   Copy Code

<node pkg="tf2\_ros" type="static\_transform\_publisher"

name="base\_footprint\_broadcaster" args="0 0 0.09 0 0 0 /base\_footprint /base\_link"/>

Now that we have added the LIDAR and IMU, we need to add more static transforms. Calculations are made from the middle of the robot, so any distances reported in the scan message need to be transformed from the location of the LIDAR to the centre of the robot.

We could add two more static transforms like the one above to the launch file but instead we will remove the existing one and write a simple node, which we will include in the rodney package, to handle all three static transforms.

In the rodney/src folder, I have added the static\_broadcaster.py file. This Python script will create a node responsible for broadcasting the three static transforms. This message is latched so nodes that require the transform will be passed the data on startup.

Hide   Shrink Image 9 for Rodney - A Long Time Coming Autonomous Robot (Part 6)   Copy Code

*# Rodney robot static transform broadcaster*

import sys

import rospy

import tf

import tf2\_ros

import geometry\_msgs.msg

def main(args):

rospy.init\_node('rodney\_static\_broadcaster', anonymous=False)

rospy.loginfo("rodney static broadcaster node started")

broadcaster = tf2\_ros.StaticTransformBroadcaster()

*# Static transform for the base\_footprint to base\_link*

st1 = geometry\_msgs.msg.TransformStamped()

st1.header.stamp = rospy.Time.now()

st1.header.frame\_id = "/base\_footprint"

st1.child\_frame\_id = "/base\_link"

st1.transform.translation.x = 0.0

st1.transform.translation.y = 0.0

st1.transform.translation.z = 0.09

quat = tf.transformations.quaternion\_from\_euler(0.0, 0.0, 0.0)

st1.transform.rotation.x = quat[0]

st1.transform.rotation.y = quat[1]

st1.transform.rotation.z = quat[2]

st1.transform.rotation.w = quat[3]

*# Static transform for the baselink to laser*

st2 = geometry\_msgs.msg.TransformStamped()

st2.header.stamp = rospy.Time.now()

st2.header.frame\_id = "/base\_link"

st2.child\_frame\_id = "/laser"

st2.transform.translation.x = 0.085

st2.transform.translation.y = 0.0

st2.transform.translation.z = 0.107

quat = tf.transformations.quaternion\_from\_euler(0.0, 0.0, 0.0)

st2.transform.rotation.x = quat[0]

st2.transform.rotation.y = quat[1]

st2.transform.rotation.z = quat[2]

st2.transform.rotation.w = quat[3]

*# Static transform for the baselink to imu*

st3 = geometry\_msgs.msg.TransformStamped()

st3.header.stamp = rospy.Time.now()

st3.header.frame\_id = "/base\_link"

st3.child\_frame\_id = "/imu"

st3.transform.translation.x = 0.0

st3.transform.translation.y = 0.0

st3.transform.translation.z = 0.058

quat = tf.transformations.quaternion\_from\_euler(0.0, 0.0, 0.0)

st3.transform.rotation.x = quat[0]

st3.transform.rotation.y = quat[1]

st3.transform.rotation.z = quat[2]

st3.transform.rotation.w = quat[3]

broadcaster.sendTransform([st1, st2, st3])

*# static transforms are latched so all we need to do here is spin*

rospy.spin()

if \_\_name\_\_ == '\_\_main\_\_':

main(sys.argv)

The node is started in the rodney.launch file:

Hide   Copy Code

*<!-- Static transforms in the system -->*

<node pkg="rodney" type="static\_broadcaster.py" name="static\_broadcaster\_node"/>

## Launch File

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

Hide   Copy Code

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 messages.

The full rodney.launch file now looks like this:

Hide   Shrink Image 10 for Rodney - A Long Time Coming Autonomous Robot (Part 6)   Copy Code

<?xml version="1.0" ?>

<launch>

*<!-- Static transforms in the system -->*

<node pkg="rodney" type="static\_broadcaster.py" name="static\_broadcaster\_node"/>

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

<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"/>

*<!-- 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" output="screen">

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

</node>

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

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

</node>

*<!-- 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" 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>

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

name="scan\_to\_scan\_filter\_chain" output="screen">

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

<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"/>

*<!-- Add calibration to raw imu data -->*

<node pkg="imu\_calib" type="apply\_calib" name="imu\_calib" output="screen">

<param name="calib\_file" value="$(find rodney)/config/imu\_calib.yaml"/>

</node>

*<!-- Node to fuse motor encoder and IMU data for odom -->*

<node pkg="robot\_localization" type="ekf\_localization\_node" name="ekf\_localization\_node">

<remap from="odometry/filtered" to="odom"/>

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

</node>

</launch>

## Minor Code Changes

I have also made some minor changes to the code mostly around being able to enable/disable the LIDAR when the robot is in manual mode and also to ensure we don't get a runaway robot when it is being remotely controlled.

### Updates to the rodney\_missions Node

I have made small changes so that this node can enable/disable the LIDAR by starting and stopping the LIDAR motor. The request comes from the rodney node but it's important that the control is done here so that this node can ensure the LIDAR is running when in the future, it requests 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.

Hide   Copy Code

*# 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.

Hide   Copy Code

*# 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.

Hide   Copy Code

elif parameters[0] == 'J4':

*# Simple job to toggle the LIDAR on/off*

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

### 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.

Also, 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 a 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 one second, then 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.

Hide   Copy Code

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.

Hide   Copy Code

*// 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.

Hide   Copy Code

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 RodneyNodeconstructor.

Hide   Copy Code

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 last message.

Hide   Copy Code

*// 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 it's been more than one second since the last heartbeat message was received. Below is the complete new version of the sendTwist function.

Hide   Shrink Image 11 for Rodney - A Long Time Coming Autonomous Robot (Part 6)   Copy Code

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.

Hide   Shrink Image 12 for Rodney - A Long Time Coming Autonomous Robot (Part 6)   Copy Code

*// 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.

Hide   Copy Code

<?xml version="1.0" ?>

<launch>

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

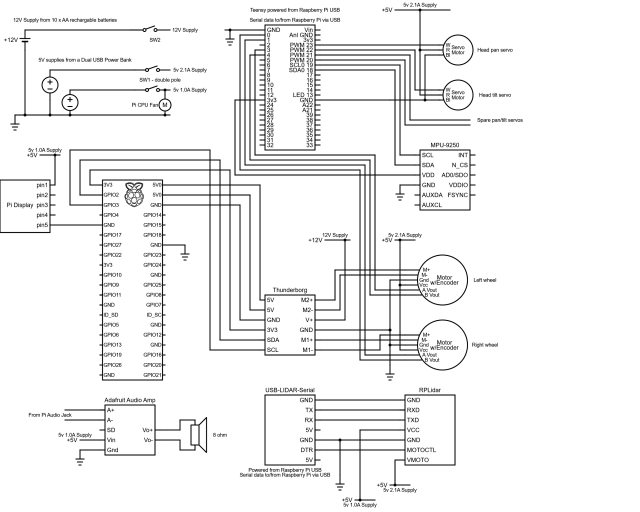
<node pkg="keyboard" type="keyboard" name="keyboard"/>

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

</launch>

## 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.



A complete bill of material for the project so far is [available here](https://github.com/phopley/rodney-project/blob/master/Articles/Part06/bom.md).

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](https://github.com/phopley/rodney-project/blob/master/Articles/Part06/pi_image.md).

## Using the Code

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

### 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:

Hide   Copy Code

$ mkdir -p ~/rodney\_ws/src

$ cd ~/rodney\_ws/

$ catkin\_make

Copy the packages face\_recognition, face\_recognition\_msgs, head\_control, imu\_calib, pan\_tilt, rodney, rodney\_missions, ros-keyboard, rplidar-ros, servo\_msgs, speech, tacho\_msgs and thunderborg into the ~/rodney\_ws/src folder.

Build the code with the following commands:

Hide   Copy Code

$ cd ~/rodney\_ws/

$ catkin\_make

Check that the build completes without any errors.

You will also need to compile and download the sketch to the 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:

Hide   Copy Code

$ mkdir -p ~/test\_ws/src

$ cd ~/test\_ws/

$ catkin\_make

Copy the packages rodney, joystick, odom\_test and ros-keyboard into the ~/test\_ws/src folder, and then build the code with the following commands:

Hide   Copy Code

$ 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.

Hide   Copy Code

$ 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.

Hide   Copy Code

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.

### Starting the Code

On the robot hardware, run the following commands to start all the current nodes in the system:

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch

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

Hide   Copy Code

$ 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:

Hide   Copy Code

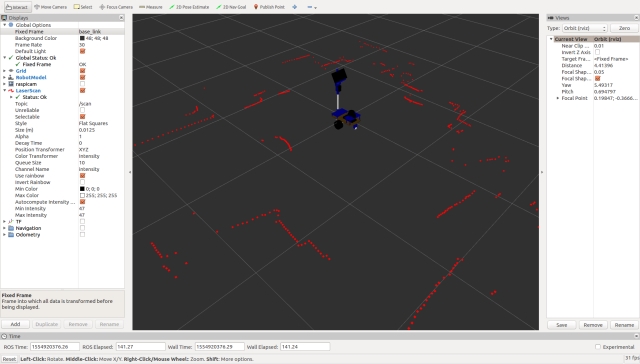
$ source test\_ws/devel/setup.bash

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

$ roslaunch rodney rviz.launch

### Testing the LIDAR

Configure rviz so that the Fixed Frame is base\_link and so that LaserScan is displaying the /scan topic. You should see a visual representation of the scan message within rviz.



It should also be possible to use either the joystick controller or keyboard to remotely control the robot avoid obstacles that are picked up by the laser.

In a new terminal on the workstation, enter the following commands to disable to LIDAR:

Hide   Copy Code

$ source test\_ws/devel/setup.bash

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

$ rosservice call /stop\_motor

The LIDAR should stop spinning and the scan display in rviz should not be updated.

In the terminal, enter the following command to enable the LIDAR:

Hide   Copy Code

$ rosservice call /start\_motor

The LIDAR should start spinning and the scan display in rviz should be updated.

### Testing the IMU

To ensure both the raw and fused odometry are zero, close the terminal on the robot hardware that launched the code.

On the robot hardware, re-start with the following commands in a new terminal:

Hide   Copy Code

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch

Configure rviz so that:

* the Fixed Frame is odom
* The Transforms (TF) base\_link and odom axes are displayed
* The odometry /raw\_odom and /odom are displayed in different colours

Mark the position of the robot on the floor. Use the keyboard or joystick to move the robot.

In a terminal on the workstation, type the following to view the current odometry message:

Hide   Copy Code

$ source test\_ws/devel/setup.bash

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

$ rostopic echo /odom

From the message, note the current pose position, x and y. Measure the distance the robot has moved from the marked position and confirm that it is a good approximation.

Use the keyboard or joystick to rotate the robot. The pose orientation is not so easy to compare as it is a Quaternion. You can see the orientation in rviz, but you can also use the test script odom\_test, to confirm the angle in degrees.

In the terminal on the workstation, press **Ctrl-C** and then enter the following command:

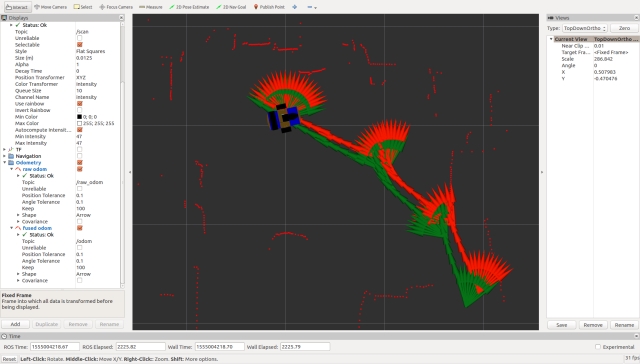
Hide   Copy Code

$ rosrun odom\_test odom\_test1\_node.py

Once per second, the node will print out the pose orientation in degrees for both the raw and fused odometry message.

On the rviz display, you will see two sets of arrows showing the raw and fused odometry. Also, when the robot is returned to the place it started, the base\_link and odom transforms should be fairly close.

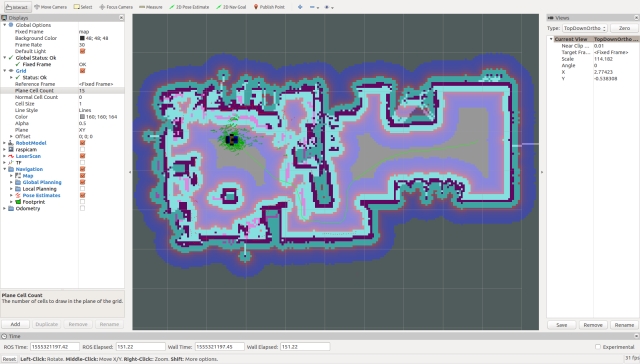
Image shows the trace of the raw odometry (red arrows) and the fused odometry (green arrows):



## Points of Interest

In this part, we have completed Design Goal 4 and also added an IMU to improve the odometry.

In the next article, we add packages that use the LIDAR for autonomous navigation, including creating a map from the LIDAR and odom transform data. We will use rviz to set target poses and the robot will autonomously navigate to the set location.



## History

* 2019/04/15: Initial release