```
###################################
##  wall_follow Node YAML File  ##
###################################

## Values are in meters

# User Defined Variables and Constants for IR
wall_follow/ir/min_ir_val:  0.5
wall_follow/ir/max_ir_val:  4.0

# Robot Dimensions
wall_follow/dimensions/tick_rev:     90
wall_follow/dimensions/dia_wheel:    0.087
wall_follow/dimensions/width_wheel:  0.215

# wall follow characteristics
wall_follow/wall_follow/des_wall_dist:     1
wall_follow/wall_follow/min_front_dist:    0.5
wall_follow/wall_follow/max_front_dist:    1.0
wall_follow/wall_follow/min_side_dist:     0.5
wall_follow/wall_follow/max_side_dist:     1.0
wall_follow/wall_follow/right_array_val:   12
wall_follow/wall_follow/center_array_val:  17

# controller
wall_follow/controller/kp:  0.9

# Topics
wall_follow/odometry_topic:    "/wall_follow/odometry"
wall_follow/laser_scan_topic:  "/camera/scan"
wall_follow/traj_angle_topic:  "/wall_follow/traj_angle"

# state
wall_follow/state: false
```

```
<!-- ROS LAUNCH FILE -->
<launch>

  <!-- Get parameters from .yaml file -->
  <rosparam file="$(find wall_follow)/launch/wall_follow.yaml"/>

  <!-- Launch nodes -->
  <node
        pkg="wall_follow"
        type="wall_follow_node"
        name="wall_follow_node"
        output="screen"
        required="true"
  />


<!--
  <node pkg="rosserial_python" type="serial_node.py" name="serial_node"
output="screen" required="true">
      <param name="~port" value="/dev/ttyACM0" />
      <param name="~baud" value="115200" />
  </node>
 -->
</launch>
```

```
/*****************************************************************
 * Header File:
 *    Arduino Wall Follow : Allow an RC car to read IR values in an array
 *                          and output motors controls to an arduino
 * Author:
 *    Helmut Neher
 * Summary:
 *    Read IR Values from ROS and YAML file of parameters, and odometry
 *    from odometry (encoders/kinect) to calulate heading and speed and
 *    publish using ROS to arduino.
 *****************************************************************/

#ifndef WALL_FOLLOW_H
#define WALL_FOLLOW_H

#include <ros/ros.h>
#include <ros/console.h>
#include <std_msgs/Bool.h>
#include <std_msgs/String.h>
#include <std_msgs/Float32.h>
#include <std_msgs/Float32MultiArray.h>
#include <stdio.h>
#include <iostream>
#include <sensor_msgs/LaserScan.h>


/*****************************************
 * ARDUINOWALLFOLLOW
 * Wall Following Class which interfaces with
 * arduino
 *****************************************/
class wall_follow
{
public:
   // constructor
   wall_follow();

   // destructor
   ~wall_follow()          { }

private:
   void irCb  (const sensor_msgs::LaserScan::ConstPtr& msg);
   void odomCb(const std_msgs::Float32MultiArray::ConstPtr& msg);
   void getParam(ros::NodeHandle nh);
   void getTrajAngle();

  // ROS variables
  ros::NodeHandle nh_;
  ros::Subscriber ir_sub;
  ros::Subscriber odom_sub;
  ros::Publisher  traj_angle;

  // wall_follow variables:
  //ubfloat ir [];                      // store IR values
  double min_ir_val, max_ir_val;            // max and min ir values can be read
  int ticks_l,ticks_r, ticks_r_old,
  ticks_l_old, dTicks_l, dTicks_r;          // change in ticks on left and right
encoders   // subscribing to arduino
```

```
  int tick_rev;                                // ticks for revolution
  double dia_wheel, width_wheel;               // diameter of wheel and width between
wheels
  double left_dist, right_dist, center_dist; // distanced traveled from left and
right wheel and robot center
  double x,y,phi;                              // current estimate of robot
(m,m,radians)
  double phi_des, v_des, w_des;               // desired robot variables
  int v_left, v_right;                        // velocity of left and right encoder
  int kp;                                     // prop. conrol gain
  double pi;                                  // pi = 3.1459

  int min_dist, obj_front, obj_left,
  obj_right, x0_rel, y0_rel, phi_rel;        // avoidance behavor characteristics

  double des_wall_dist, min_front_dist,       // wall follow variables
  max_front_dist, min_side_dist,
  max_side_dist;

  int right_array_val, center_array_val;
  double right_laser_scan, center_laser_scan;

  double uF, uS,phi_w;                        // forward, side vector and angle

  std::string odometry_topic, laser_scan_topic, traj_angle_topic;

  sensor_msgs::LaserScan laser_msg;
  std_msgs::Float32 traj_angle_msg;
};


#endif // ARDUINOWALLFOLLOW_H
```

```
/*****************************************************************
 * Source File:
 *    Arduino Wall Follow : Allow an RC car to read IR values in an array
 *                          and output motors controls to an arduino
 * Author:
 *    Helmut Neher
 * Summary:
 *    Read IR Values from ROS and YAML file of parameters, and odometry
 *    from odometry (encoders/kinect) to calulate heading and speed and
 *    publish using ROS to arduino.
 *****************************************************************/
#include <wall_follow/wall_follow.h>

/***************************************
 * ArduinoWallFollow : Callback
 * Initializes subscriber and begins wall
 * following
 ***************************************/
 wall_follow::wall_follow() : ticks_r_old(0),ticks_l_old(0),x(0),y(0),phi(0)
{
   // get Parameters from Ros server
   ROS_INFO("Getting Parameters");
   getParam(nh_);

   // Subscribe to values
   ROS_INFO("Subscribing to topics");
   ir_sub   = nh_.subscribe(laser_scan_topic, 10, &wall_follow::irCb, this);     //
subscribe to ir values
   odom_sub = nh_.subscribe(odometry_topic, 10, &wall_follow::odomCb, this);   //
subscribe to odom values
   traj_angle = nh_.advertise<std_msgs::Float32>(traj_angle_topic,10);

 //  image_pub_ = it_.advertise("/image_converter/output_video", 1);
}

/***************************************
 * Wall Follow : Get Param
 * Gets all parameters from Ros server and
 * provides default if unable to get values
 ***************************************/
 void wall_follow::getParam(ros::NodeHandle nh)
 {

   if (nh.getParam("wall_follow/ir/min_ir_val", min_ir_val))
      ROS_INFO_STREAM("Got param min_ir_val: " << min_ir_val);
   else{
      ROS_INFO("Failed to get wall_follow/ir/min_ir_val");
      min_ir_val = 0.5;
   }

   if (nh.getParam("wall_follow/ir/max_ir_val", max_ir_val))
      ROS_INFO_STREAM("Got param max_ir_val: " << max_ir_val);
   else{
      ROS_INFO("Failed to get wall_follow/ir/max_ir_val");
      max_ir_val = 4.0;
   }

   if (nh.getParam("wall_follow/dimensions/tick_rev", tick_rev))
```

```
   ROS_INFO_STREAM("Got param tick_rev: " << tick_rev);
else{
   ROS_INFO("Failed to get wall_follow/dimensions/tick_rev");
   tick_rev = 128;
}

if (nh.getParam("wall_follow/dimensions/dia_wheel", dia_wheel))
   ROS_INFO_STREAM("Got param dia_wheel: " << dia_wheel);
else{
   ROS_INFO("Failed to get wall_follow/dimensions/dia_wheel");
   dia_wheel = 0.1;
}

if (nh.getParam("wall_follow/dimensions/width_wheel", width_wheel))
   ROS_INFO_STREAM("Got param width_wheel: " << width_wheel);
else{
   ROS_INFO("Failed to get wall_follow/dimensions/width_wheel");
   width_wheel = 0.3;
}

if (nh.getParam("wall_follow/wall_follow/des_wall_dist", des_wall_dist))
   ROS_INFO_STREAM("Got param des_wall_dist: " << des_wall_dist);
else{
   ROS_INFO("Failed to get wall_follow/wall_follow/des_wall_dist");
   des_wall_dist = 0.55;
}

if (nh.getParam("wall_follow/wall_follow/min_front_dist", min_front_dist))
   ROS_INFO_STREAM("Got param min_front_dist: " << min_front_dist);
else{
   ROS_INFO("Failed to get wall_follow/wall_follow/min_front_dist");
   min_front_dist = 0.5;
}

if (nh.getParam("wall_follow/wall_follow/max_front_dist", max_front_dist))
   ROS_INFO_STREAM("Got param max_front_dist: " << max_front_dist);
else{
   ROS_INFO("Failed to get wall_follow/wall_follow/max_front_dist");
   max_front_dist = 0.8;
}

if (nh.getParam("wall_follow/wall_follow/min_side_dist", min_side_dist))
   ROS_INFO_STREAM("Got param min_side_dist: " << min_side_dist);
else{
   ROS_INFO("Failed to get wall_follow/wall_follow/min_side_dist");
   min_side_dist = 0.5;
}

if (nh.getParam("wall_follow/wall_follow/max_side_dist", max_side_dist))
   ROS_INFO_STREAM("Got param max_side_dist: " << max_side_dist);
else{
   ROS_INFO("Failed to get wall_follow/wall_follow/max_side_dist");
   max_side_dist = 0.8;
}

if (nh.getParam("wall_follow/wall_follow/right_array_val", right_array_val))
   ROS_INFO_STREAM("Got param max_side_dist: " << right_array_val);
else{
   ROS_INFO("Failed to get wall_follow/wall_follow/right_array_val");
```

```
      max_side_dist = 12;
   }

   if (nh.getParam("wall_follow/wall_follow/center_array_val", center_array_val))
      ROS_INFO_STREAM("Got param max_side_dist: " << center_array_val);
   else{
      ROS_INFO("Failed to get wall_follow/wall_follow/center_array_val");
      max_side_dist = 17;
   }

   if (nh.getParam("wall_follow/controller/kp", kp))
      ROS_INFO_STREAM("Got param kp: " << kp);
   else{
      ROS_INFO("Failed to get wall_follow/controller/kp");
      kp = 1;
   }

   if (nh.getParam("wall_follow/odometry_topic", odometry_topic))
      ROS_INFO_STREAM("Got param odometry_topic: " << odometry_topic);
   else
      ROS_INFO("Failed to get wall_follow/odometry_topic");


   if (nh.getParam("wall_follow/laser_scan_topic", laser_scan_topic))
      ROS_INFO_STREAM("Got param laser_scan_topic: " << laser_scan_topic);
   else
      ROS_INFO("Failed to get wall_follow/laser_scan_topic");

   if (nh.getParam("wall_follow/traj_angle_topic", traj_angle_topic))
      ROS_INFO_STREAM("Got param laser_scan_topic: " << traj_angle_topic);
   else
      ROS_INFO("Failed to get wall_follow/traj_angle_topic");
 }

/****************************************
 * irCB : Infrared Callback
 * Read IR values and store in array
 ****************************************/
void wall_follow::irCb(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  right_laser_scan  = msg->ranges[right_array_val];
  center_laser_scan = msg->ranges[center_array_val];
  laser_msg = *msg;
  ROS_INFO("I heard laserscan message");
//  ROS_INFO_STREAM("right_array_val  " << right_laser_scan);
//  ROS_INFO_STREAM("center_array_val " << center_laser_scan);
}

/****************************************
 * odomCB : Odometry Callback
 * Read Odometry values and store in array
 ****************************************/
void wall_follow::odomCb(const std_msgs::Float32MultiArray::ConstPtr& msg)
{
  ticks_l = msg->data[0];    // array of encoder values (just 2 values sent from
the arduino)
  ticks_r = msg->data[1];
  ROS_INFO_STREAM("I heard and accepted ticks_l: " << ticks_l << " and ticks_r: "
<< ticks_r);
```

```
 getTrajAngle();
}

/***************************************
 * odomCB : Odometry Callback
 * Read Odometry values and store in array
 ***************************************/
void wall_follow::getTrajAngle()
{

      // calculate encoder counts between each loop.
      dTicks_l    = ticks_l - ticks_l_old;
      ticks_l_old = ticks_l;

      dTicks_r    = ticks_r - ticks_r_old;
      ticks_r_old =  ticks_r;

      // update pose of the car
      //calculate distance
      left_dist   = pi * dia_wheel * dTicks_l / tick_rev;
      right_dist  = pi * dia_wheel * dTicks_r / tick_rev;
      center_dist = (left_dist + right_dist) / 2;

      // calculate new pose
      x   = x + center_dist * cos(phi);
      y   = y + center_dist * sin(phi);
      phi = phi + (right_dist - left_dist) / width_wheel;
      phi = atan2(sin(phi),cos(phi));

      // calculate forward and side vector and angle
      if (center_laser_scan > max_ir_val)
            center_laser_scan = max_ir_val;
      else if (center_laser_scan < min_ir_val)
            center_laser_scan = min_ir_val;

      if (right_laser_scan > max_ir_val)
            right_laser_scan = max_ir_val;
      else if (right_laser_scan < min_ir_val)
            right_laser_scan = min_ir_val;

      uF    = center_laser_scan;
      uS    = sqrt(2) * des_wall_dist - right_laser_scan;
      phi_w = atan2(uS,uF);

      // set trajectory msg and publish msg
      traj_angle_msg.data = phi_w;
      traj_angle.publish(traj_angle_msg);

      ROS_INFO_STREAM("Angle Tracking COmpleted: " << traj_angle_msg);
}
```

```
/*****************************************************************
 * Main File:
 *    Arduino Wall Follow : Allow an RC car to read IR values in an array
 *                          and output motors controls to an arduino
 * Author:
 *    Helmut Neher
 * Summary:
 *    Main file to initialize Wall Follow Class
 *****************************************************************/

#include <ros/ros.h>
//#include <std_msgs/String.h>
#include <wall_follow/wall_follow.h>


/*************************************
 * getIRValues (array)
 * Gets IR Values from the ROS Callback
 * function or is this the callback function?
 *************************************/
int main(int argc, char **argv)
{
  ros::init(argc, argv, "wall_follow_arduino");
  wall_follow awf;
  ros::spin();

  return 0;
}
```

```
/*************************************************************************
 * Main Arduino File:
 *    Arduino Wall Follow : An arduino file that subscribes to traj_angle
 *           Sketch         and publishes odometry as well as turning
 *                          for feedback. The sketch also outputs the
 *                          turning angle to the steering servo.
 * Authors:
 *    David Christian, Korey Danklefsen, Glenn Mackay, and Helmut Neher
 * Summary:
 *    Main Arduino Sketch to be implemented
 *************************************************************************/
//#define USE_USBCON
#include <ros.h>
#include <std_msgs/Float32MultiArray.h>
#include <std_msgs/Float32.h>
#include <std_msgs/Bool.h>

#include <Servo.h>


// ROS variables
ros::NodeHandle nh;

// Publishing Encoder values
std_msgs::Float32MultiArray encoder_arr;
std_msgs::Float32 turning;
std_msgs::Bool state;
ros::Publisher chatter ("/wall_follow/odometry", &encoder_arr);
ros::Publisher chatter2("/wall_follow/turn", &turning);

//declare variables
float encoder_value[2] = {0,1};   // variables for putting encoder values in
float traj_angle;
bool states;
int x,start,stp;


// subscriber CallBack
void messageCb (const std_msgs::Float32& trajectory_angle){
  traj_angle = trajectory_angle.data;
}

void stateCb (const std_msgs::Bool& message){
  states = state.data;
}



//initialize subscriber
ros::Subscriber<std_msgs::Float32> sub("/wall_follow/traj_angle", &messageCb);
ros::Subscriber<std_msgs::Bool> sub2("/wall_follow/state", &stateCb);




//____________________Set up encoder stuff_____________________//

enum PinAssignments {
  encoderPinA = 2,   // rigth
```

```
  encoderPinB = 3,   // left
  encoderPinC = 18,   // right
  encoderPinD = 19,   // left
};

volatile unsigned int encoderPos = 0;  // a counter for the dial
unsigned int lastReportedPos = 1;   // change management
volatile unsigned int encoderPos1 = 0;  // a counter for the dial
unsigned int lastReportedPos1 = 1;   // change management
static boolean rotating=false;      // debounce management

// interrupt service routine vars
boolean A_set = false;
boolean B_set = false;
boolean A1_set = false;
boolean B1_set = false;




//_____________Servo stuff_______________//

Servo throttleServo,steerServo;  //output
#define SERVO_THROTTLE_PIN    10
#define SERVO_STEER_PIN       11
int var = 500;




void setup() {
//Serial.begin(9600);
x=1;
stp = 0;

// initialize nodes and stuff

//nh.getHardware()->setBaud(57600);
nh.getHardware()->setBaud(115200);
nh.initNode();
nh.advertise(chatter);
nh.advertise(chatter2);
nh.subscribe(sub);
nh.subscribe(sub2);


encoder_arr.layout.dim[0].label = "test";
encoder_arr.layout.dim[0].size = 2;
encoder_arr.layout.dim[0].stride = 1*2;
encoder_arr.layout.data_offset = 0;
encoder_arr.data_length = 2;

  pinMode(encoderPinA, INPUT);
  pinMode(encoderPinB, INPUT);
  pinMode(encoderPinC, INPUT);
  pinMode(encoderPinD, INPUT);
 // turn on pullup resistors
  digitalWrite(encoderPinA, HIGH);
  digitalWrite(encoderPinB, HIGH);
```

```
  digitalWrite(encoderPinC, HIGH);
  digitalWrite(encoderPinD, HIGH);

// encoder pin on interrupt 0 (pin 2)
  attachInterrupt(0, doEncoderA, CHANGE);
// encoder pin on interrupt 1 (pin 3)
  attachInterrupt(1, doEncoderB, CHANGE);
// encoder pin on interrupt 3 (pin 18)
  attachInterrupt(3, doEncoderC, CHANGE);
// encoder pin on interrupt 4 (pin 19)
  attachInterrupt(4, doEncoderD, CHANGE);

//__________________Steering & Throttle_________________//

  throttleServo.attach(SERVO_THROTTLE_PIN);
  steerServo.attach(SERVO_STEER_PIN);

}

//ESC SETUP
void setup_ESC()
{
  long tmp_time = millis();
  while ((millis() - tmp_time) <5000)
    throttleServo.writeMicroseconds(1500);
}

void loop() {

nh.spinOnce();


float turn = (1000/3.14)*traj_angle+1500;
turning.data = turn;

steerServo.writeMicroseconds (turn);

// Finding ticks from encoders

rotating = true;  // reset the debouncer

  if (lastReportedPos != encoderPos) {
    //Serial.print("IndexL:");
    //Serial.println(encoderPos, DEC);
    encoder_value[0] = encoderPos;
    lastReportedPos = encoderPos;
  }



  if (lastReportedPos1 != encoderPos1) {
    //Serial.print("IndexR:");
    //Serial.println(encoderPos1, DEC);
    encoder_value[1] = encoderPos1;
    lastReportedPos1 = encoderPos1;
  }
```

```
// assigning values and publishing to ROS
    encoder_arr.data = encoder_value;         // assigning values from encoder to
ROS message
    chatter.publish(&encoder_arr);            // publishing  ROS message to ROS
    chatter2.publish(&turning);
    //delay(1000);

}

// Interrupt on A changing state
void doEncoderA(){
  // debounce
  if ( rotating ) delay (1);  // wait a little until the bouncing is done

  // Test transition, did things really change?
  if( digitalRead(encoderPinA) != A_set ) {  // debounce once more
    A_set = !A_set;

    // adjust counter + if A leads B
    if ( A_set && !B_set )
      encoderPos += 2;

    rotating = false;  // no more debouncing until loop() hits again
  }
}

// Interrupt on B changing state, same as A above
void doEncoderB(){
  if ( rotating ) delay (1);
  if( digitalRead(encoderPinB) != B_set ) {
    B_set = !B_set;
    //  adjust counter - 1 if B leads A
    if( B_set && !A_set )
      encoderPos -= 2;

    rotating = false;
  }
}

// Interrupt on A1 changing state
void doEncoderC(){
  // debounce
  if ( rotating ) delay (1);  // wait a little until the bouncing is done

  // Test transition, did things really change?
  if( digitalRead(encoderPinC) != A1_set ) {  // debounce once more
    A1_set = !A1_set;

    // adjust counter + if C leads D
    if ( A1_set && !B1_set )
      encoderPos1 += 2;

    rotating = false;  // no more debouncing until loop() hits again
  }
}

// Interrupt on B1 changing state, same as A above
void doEncoderD(){
```

```
  if ( rotating ) delay (1);
  if( digitalRead(encoderPinD) != B1_set ) {
    B1_set = !B1_set;
    //  adjust counter - 1 if D leads C
    if( B1_set && !A1_set )
      encoderPos1 += 2;

    rotating = false;
  }
}

int timer(){
  if (x == 1 && stp == 0){
    start = millis();
    stp = start;
    return 1;
  }
  else if(x == 0){
    start = millis();
    if (start - stp >= var+200){
      x = 1;
      stp = start;
      return 1;
    }
    else{
      return 0;
    }
  }
  else if(x == 1){
    start = millis();
    if(start - stp >= var){
      x = 0;
      stp = start;
      return 0;
    }
    else{
      return 1;
    }
  }
}
```