TurtleDog

ME 4451 – Robotics: Final Project Report

Preston Culbertson, Alexander Gil, Orlin Velev

December 7, 2015

Objective

The objective of the TurtleDog project is to use a TurtleBot equipped with a Microsoft Kinect vision system to detect and follow a user. The code for the project runs within the Robot Operating System (ROS) framework, while servo motor "ears" controlled by an Arduino microcontroller are used as a status indicator. A different ear movement pattern is associated with either the "following" or "not following" state of the robot. The ears are programmed to wag while the TurtleDog is "happy" and following its master and droop when the TurtleDog is "sad" because it has been temporarily abandoned by its master.

Another objective aside from the physical TurtleDog project is a tutorial document for using the TurtleBot based on the team's learnings throughout the semester. For those with little or no experience with Linux or ROS, using the TurtleBot can be intimidating. The aim of this tutorial is to facilitate the use of the TurtleBots in future labs and projects in Georgia Tech's ME 4451 robotics course.

Design & Analysis

With all three team members having never worked with ROS before, the first step of the design process was learning what ROS was and how it worked. In ROS, a user creates nodes, which communicate with one another by publishing and subscribing messages to specific user-defined topics. These nodes all run different code. If Node A requires a variable from Node B to execute its code, Node B must publish the variable to a user-defined message topic and Node A then subscribes to that topic in order to execute its code. One extremely powerful quality of ROS is the fact that it is language agnostic. That is, one node may be written in C++ and another in Python and as long as the user includes the correct publishing/subscriber protocol, all of these nodes can work seamlessly together. One instance when this is convenient is when using open source code from multiple sources. Regardless of the language of the open source code, it is fairly simple to integrate it into an existing nodal network.

The design of the code was strongly influenced by these message passing capabilities of ROS. The goal of determining the presence of a user and passing that information along to physical hardware led to the design choice of using a pair of communicating nodes, one of which ran on the TurtleBot's laptop and one of which ran on the Arduino. The laptop based node was chosen to be a modified version of the open source turtlebot_follower program that came with the TurtleBot. The Arduino based node was completely original and designed to work with the hardware available in the lab space. This hardware planned for the project included a Microsoft Kinect, a Dell 210 G1 netbook, a standard Arduino Uno, two Fitec FS90MG servos and a breadboard. Later design changes also necessitated the addition of 4 AA batteries in a battery pack as described in the implementation section of this report.

Almost immediately after the TurtleDog project was started, the decision was made to move the Kinect from the middle shelf of the TurtleBot to its top shelf. This provided the vision system with a less obstructed view of the robot's environment by eliminating the field-of-view constraint above the camera present when mounted on the middle shelf. Because the main following objective of the

project relies heavily on the Kinect's ability to see and identify users, the team wanted to provide the Kinect with as complete a picture of its surroundings as possible. Additionally, the adjusted placement created a convenient mounting location for the servo motor ears. By also mounting the Arduino, breadboard, and battery pack on the top shelf, the wiring of the servo motors was very simple.

Implementation

The team's goal was to explore the capabilities of the TurtleBot and ROS and include as many of these capabilities as possible in the final project. The TurtleBot comes with a large number of packages pre-installed. The team used the turtlebot_follower program code as the basis for the final TurtleDog code. The turtlebot_follower code identifies distinct objects in the Kinect's field of view and maintains a specific distance from the centroid of the largest object. This code was modified to publish a "0" when the robot could not identify a user and, therefore, was not following and a "1" when the robot was following a user. This "0" or "1" value was published to a topic named "ears". The variable type used was uint16_t because it is a standard C type variable that could be sent between the follower node and the Arduino node.

Another feature of ROS demonstrated by the TurtleDog project is its ability to interface with external hardware. In this case, an Arduino microcontroller and Microsoft Kinect were both used. The Arduino was coded such that it functioned as an autonomous ROS node. Details on how this was done can be found in the tutorial prepared by the team. As a ROS node, the Arduino is able to subscribe to the "ears" topic and use the uint16_t value published by the follower code to determine the motion of the servo motors. The Arduino communicates with the laptop via a serial port connection. The rosserial_python package downloaded from the internet allows the Arduino and laptop to communicate via USB. This is a perfect example of ROS's ability to handle code in different languages. The serial connection is opened by code written in Python to allow two nodes written in C++ to pass messages.

As previously mentioned, one challenge that the team encountered involved the voltage supplied to the two servo motors. The specifications for the Fitec FS90MG servos recommend an operating voltage of 6V. The highest voltage obtainable via a pin on the Arduino is 5V. Initially, the servos were powered using the 5V pin on the Arduino. The Arduino pulls 5V over the USB cable in order to power its processing chip. However, with the servos attached to the 5V pin on the Arduino, the voltage being sent to the Arduino via the cable was being used to operate the servos, leaving an insufficient voltage available for the processing chip. Because the voltage being sent to the processing chip was too low, the code on the Arduino was only able to run momentarily before crashing and displaying an error. To test the theory that insufficient voltage supply was the cause of the error in the Arduino code, the team plugged an external 6VDC power supply into the Arduino from a wall outlet. The power wire for the servos was then moved from the 5V pin to the Vin pin on the Arduino while leaving the USB connection intact. This allowed for the servos to receive voltage from the 6V source without interfering with the 5V needed to power the Arduino. After making these changes, the Arduino and servos functioned

properly and never errored but a portable solution was needed as the wall outlet connection limited the TurtleBot's ability to follow a user around the room. The final solution that the team developed was to use a battery holder capable of holding 4 AA batteries (1.5V each) to externally supply the servos with 6V. A breadboard was used to connect both servo's power wires to the positive terminal of the battery pack and their ground wires to the negative terminal of the battery pack. The signal wires of the servos were connected to two of the output pins on the Arduino.

Results & Discussion

The TurtleDog successfully completes all of the objectives the team set out to accomplish at the beginning of the project. The goal of the project was explore as many of the TurtleBot's and ROS's capabilities as possible and include as much of what was learned as possible. The TurtleDog demonstrates knowledge of ROS's architecture, something completely foreign to the team at first. It successfully publishes messages to a topic from one node and subscribes to this message topic from another node. This publishing and subscribing behavior is at the core of ROS and essential to understand in order to effectively utilize ROS. Not only can TurtleDog's nodes publish and subscribe to one another, but the subscriber node in the TurtleDog project is an Arduino microcontroller. This exemplifies ROS's ability to interface with external hardware. The TurtleDog also demonstrates the ability to use open source code as a means to complete an objective. For example, the code used to establish a serial connection between the follower node and Arduino node is available online as part of the rosserial package.

Several issues presented themselves throughout the semester, the most troubling being the instability of the Kinect drivers. Apple Inc. recently acquired the company that maintained and supported these drivers and made them closed source. Newer, stable versions of the drivers are no longer available online. This is an inherent risk associated with using open source software. If the challenges with the Kinect drivers had not persisted, the team planned to move forward with gesture recognition, allowing the TurtleDog to perform "tricks" based on user gestural inputs. Another future plan would be to utilize open source "player" recognition software to specifically track individuals in a crowd. The current modified turtlebot_follower code uses blob recognition to identify different objects and simply follows the centroid of the largest objects. This occasionally leads to the TurtleDog recognizing a wall, door, or other inanimate object as larger than the user at which point it begins following that object. The advantage of player recognition software is that it recognizes only humans, not inanimate objects, and can be programmed to follow specific humans.

As previously referenced, a tutorial document was prepared as an additional deliverable alongside the TurtleDog. This tutorial is intended to be a launch pad for future use of the TurtleBots in Georgia Tech's ME 4451 robotics class. The tutorial condenses solutions to some of the challenges the team faced throughout the semester into a single document. It could prove useful for either lab or project use of the TurtleBots. It makes using the TurtleBot a realistic

possibility for those students or groups without an extensive computer science background.

Learning Experiences

The primary overarching learning experience of this project was the knowledge gained about using ROS and how to make effective use of its capabilities. The most valuable of these capabilities was the ability to communicate information simply with through publishing and subscribing to topics. By using online tutorials and example code, the team was able to learn the proper syntax for embedding publishing and subscribing code into individual ROS nodes. This allowed for much easier communication not only between programs on the computer but also with external hardware including the Arduino and servo motors. Outside of a ROS environment, serial communication over USB often requires relatively specialized knowledge of communication protocols between the computer and device. Learning how to use the tools and packages built into ROS to support communication was integral to the success of the project.

Another important component of ROS that had to be learned and implemented was the use of the catkin build system to make packages and compile code. While the code for the Arduino portion of the project was written and compiled in an integrated development environment, the portion of the code that ran on the laptop was written and edited with Linux's built in text editors and so relied on ROS's catkin system to be compiled into executable code. The catkin_make command that performed these actions requires that all code is written properly and that all dependencies are properly specified in the CMakeLists.txt and Package.xml files that accompany every workspace. Significant amounts of time were spent debugging catkin related errors, which provided useful insight into common errors made by users in ROS. By looking through the files that were being compiled for this project and files that were successfully compiled for other programs, it was possible to determine where dependencies were missing or specified incorrectly. This reinforced the importance of understanding the resources and code base that the nodes were being written from.

Another aspect of the TurtleBot is that its laptop uses Ubuntu, a Linux based operating system, as its OS. Only one member of the TurtleDog team had extensive experience with Linux prior to the project, which proved to be valuable throughout the project, allowing the other members to watch and learn. As time progressed, all members of the team had developed a working proficiency with Linux, which allowed the team to work as both a collective unit and individually as schedules permitted. Using a Linux based computer is very different from using a computer running Windows or Mac OS. Opening a Terminal window and typing nearly everything takes some getting used to. Simply navigating file directories was something that had to be learned. The team has included some basics of using ROS and Linux in the tutorial document. The aim is that through the team's documentation of their own learning experience, future students will have an easier learning curve if using these technologies for the first time, leading to the possibility of undertaking more ambitious ROS projects.

Interfacing the TurtleBot with an external piece of hardware such as the Arduino demonstrates the flexibility of ROS. All team members had ample experience using Arduino microcontrollers prior to the TurtleDog project but had never incorporated one as a standalone node in a ROS nodal network. It became clear relatively early on in the project that the Arduino could be programmed as any other ROS node in C++ if the correct measures were taken to enable connectivity with the other TurtleDog nodes. This process is detailed further in the tutorial document. The most valuable lesson learned regarding the Arduino, however, was the issue related to insufficient voltages for all system components. It was initially believed that connecting both servos to the 5V pin on the Arduino would not have a detrimental effect on Arduino functionality. As previously described, the voltage required by the servos was drawing voltage away from the Arduino's processing chip, causing the code to err after less than five seconds. This necessitated the inclusion of an external 6V power source provided by 4 AA batteries. This experience provides valuable intuition regarding voltage distribution in a system for future projects, especially those involving a computing chip requiring a minimum voltage to operate. In mechanical engineering projects, it is often easy to focus attention on the mechanical actuators in a system, in this case the servo-driven ears. It is equally important, though, to remember that the computing components of a system also require power and that considerations must be made to supply them with the necessary power.

Another learning experience came from the problems that arose from the software driver for the Kinect vision system. The turtlebot_follower program and much of the code that ships with the turtlebot runs on the OpenNI code base. While researching how to use it, it was found that OpenNI had been purchased by Apple in 2013 which ended open source development of the original OpenNI framework. This was a problem because OpenNI still had some small bugs in the code when development stopped. Among these issues was a tendency for the Kinect to lose communication over usb when other usb devices were active on the machine. This presented a problem for this project, since one of the primary objectives was configuring serial communications with an Arduino. Trying to initialize the Kinect and Arduino without sufficient time in between would cause the Kinect to stop responding for several minutes. The software cause for this issue is likely buried deep in low level code, and was far beyond the scope of this project to troubleshoot. Instead, the group had to adapt to the limitations presented by these crashes and be careful about allowing plenty of time for the serial interface with the Kinect to be initialized before attempting to connect the Arduino. While this problem was frustrating, it provided an interesting insight into the drawbacks of having to use an unsupported software package and the problems with open source code being made proprietary. Nevertheless, like any other roadblock in the development of this project, it was overcome and the final version of the TurtleDog was able to perform a stable demonstration without any software crashes.

Appendix

Follower Code:

```
/*
 * Copyright (c) 2011, Willow Garage, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 *     * Redistributions of source code must retain the above copyright
 *       notice, this list of conditions and the following disclaimer.
 *     * Redistributions in binary form must reproduce the above copyright
 *       notice, this list of conditions and the following disclaimer in the
 *       documentation and/or other materials provided with the distribution.
 *     * Neither the name of the Willow Garage, Inc. nor the names of its
 *       contributors may be used to endorse or promote products derived from
 *       this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE
          * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include <ros/ros.h>
#include <pluginlib/class_list_macros.h>
#include <nodelet/nodelet.h>
#include <geometry_msgs/Twist.h>
```

```
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <visualization_msgs/Marker.h>
#include <turtlebot_msgs/SetFollowState.h>
#include <std_msgs/UInt16.h>

#include "dynamic_reconfigure/server.h"
#include "turtlebot_follower/FollowerConfig.h"


namespace turtlebot_follower
{
typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;

//* The turtlebot follower nodelet.
/**
 * The turtlebot follower nodelet. Subscribes to point clouds
 * from the 3dsensor, processes them, and publishes command vel
 * messages.
 */
class TurtlebotFollower : public nodelet::Nodelet
{
public:
  /*!
   * @brief The constructor for the follower.
   * Constructor for the follower.
   */
  TurtlebotFollower() : min_y_(0.1), max_y_(0.5),
                        min_x_(-0.2), max_x_(0.2),
                        max_z_(0.8), goal_z_(0.6),
                        z_scale_(1.0), x_scale_(5.0)
  {

  }

  ~TurtlebotFollower()
  {
    delete config_srv_;
  }

private:
  double min_y_; /**< The minimum y position of the points in the box. */
  double max_y_; /**< The maximum y position of the points in the box. */
  double min_x_; /**< The minimum x position of the points in the box. */
  double max_x_; /**< The maximum x position of the points in the box. */
  double max_z_; /**< The maximum z position of the points in the box. */
```

```
  double goal_z_; /**< The distance away from the robot to hold the centroid */
  double z_scale_; /**< The scaling factor for translational robot speed */
  double x_scale_; /**< The scaling factor for rotational robot speed */
  bool   enabled_; /**< Enable/disable following; just prevents motor commands */

  // Service for start/stop following
  ros::ServiceServer switch_srv_;

  // Dynamic reconfigure server
  dynamic_reconfigure::Server<turtlebot_follower::FollowerConfig>* config_srv_;

  /*!
   * @brief OnInit method from node handle.
   * OnInit method from node handle. Sets up the parameters
   * and topics.
   */
  virtual void onInit()
  {
    ros::NodeHandle& nh = getNodeHandle();
    ros::NodeHandle& private_nh = getPrivateNodeHandle();

    private_nh.getParam("min_y", min_y_);
    private_nh.getParam("max_y", max_y_);
    private_nh.getParam("min_x", min_x_);
    private_nh.getParam("max_x", max_x_);
    private_nh.getParam("max_z", max_z_);
    private_nh.getParam("goal_z", goal_z_);
    private_nh.getParam("z_scale", z_scale_);
    private_nh.getParam("x_scale", x_scale_);
    private_nh.getParam("enabled", enabled_);



    cmdpub_ = private_nh.advertise<geometry_msgs::Twist> ("cmd_vel", 1);
    markerpub_ = private_nh.advertise<visualization_msgs::Marker>("marker",1);
    bboxpub_ = private_nh.advertise<visualization_msgs::Marker>("bbox",1);
    ears_ = private_nh.advertise<std_msgs::UInt16> ("ears", 1);
    sub_= nh.subscribe<PointCloud>("depth/points", 1,
&TurtlebotFollower::cloudcb, this);

    switch_srv_ = private_nh.advertiseService("change_state",
&TurtlebotFollower::changeModeSrvCb, this);

    config_srv_ = new
dynamic_reconfigure::Server<turtlebot_follower::FollowerConfig>(private_nh);
```

```
dynamic_reconfigure::Server<turtlebot_follower::FollowerConfig>::CallbackType f
=
      boost::bind(&TurtlebotFollower::reconfigure, this, _1, _2);
  config_srv_->setCallback(f);
 }

 void reconfigure(turtlebot_follower::FollowerConfig &config, uint32_t level)
 {
  min_y_ = config.min_y;
  max_y_ = config.max_y;
  min_x_ = config.min_x;
  max_x_ = config.max_x;
  max_z_ = config.max_z;
  goal_z_ = config.goal_z;
  z_scale_ = config.z_scale;
  x_scale_ = config.x_scale;
 }

 /*!
  * @brief Callback for point clouds.
  * Callback for point clouds. Uses PCL to find the centroid
  * of the points in a box in the center of the point cloud.
  * Publishes cmd_vel messages with the goal from the cloud.
  * @param cloud The point cloud message.
  */
 void cloudcb(const PointCloud::ConstPtr&  cloud)
 {
  //X,Y,Z of the centroid
  float x = 0.0;
  float y = 0.0;
  float z = 1e6;
  //Number of points observed
  unsigned int n = 0;
  //Iterate through all the points in the region and find the average of the position
  BOOST_FOREACH (const pcl::PointXYZ& pt, cloud->points)
  {
   //First, ensure that the point's position is valid. This must be done in a
seperate
   //if because we do not want to perform comparison on a nan value.
   if (!std::isnan(x) && !std::isnan(y) && !std::isnan(z))
   {
    //Test to ensure the point is within the aceptable box.
    if (-pt.y > min_y_ && -pt.y < max_y_ && pt.x < max_x_ && pt.x > min_x_ &&
pt.z < max_z_)
    {
```

```
        //Add the point to the totals
        x += pt.x;
        y += pt.y;
        z = std::min(z, pt.z);
        n++;
      }
    }
  }
  std_msgs::UInt16 msg;
  //If there are points, find the centroid and calculate the command goal.
  //If there are no points, simply publish a stop goal.
  if (n>4000)
  {
    x /= n;
    y /= n;
    if(z > max_z_){
      ROS_DEBUG("No valid points detected, stopping the robot");

        msg.data = 0;
      ears_.publish(msg);

      if (enabled_)
      {
        cmdpub_.publish(geometry_msgs::TwistPtr(new
geometry_msgs::Twist()));

      }
      return;
    }

    ROS_DEBUG("Centroid at %f %f %f with %d points", x, y, z, n);
    publishMarker(x, y, z);

    if (enabled_)
    {
      geometry_msgs::TwistPtr cmd(new geometry_msgs::Twist());
      cmd->linear.x = (z - goal_z_) * z_scale_;
      cmd->angular.z = -x * x_scale_;
      cmdpub_.publish(cmd);
        msg.data = 1;
      ears_.publish(msg);
    }
  }
  else
  {
    ROS_DEBUG("No points detected, stopping the robot");
```

```
    publishMarker(x, y, z);
    msg.data = 0;
    ears_.publish(msg);

    if (enabled_)
    {
      cmdpub_.publish(geometry_msgs::TwistPtr(new geometry_msgs::Twist()));
    }
  }

  publishBbox();
}

bool changeModeSrvCb(turtlebot_msgs::SetFollowState::Request& request,
                     turtlebot_msgs::SetFollowState::Response& response)
{
  if ((enabled_ == true) && (request.state == request.STOPPED))
  {
    ROS_INFO("Change mode service request: following stopped");
    cmdpub_.publish(geometry_msgs::TwistPtr(new geometry_msgs::Twist()));
    enabled_ = false;
  }
  else if ((enabled_ == false) && (request.state == request.FOLLOW))
  {
    ROS_INFO("Change mode service request: following (re)started");
    enabled_ = true;
  }

  response.result = response.OK;
  return true;
}

void publishMarker(double x,double y,double z)
{
  visualization_msgs::Marker marker;
  marker.header.frame_id = "/camera_rgb_optical_frame";
  marker.header.stamp = ros::Time();
  marker.ns = "my_namespace";
  marker.id = 0;
  marker.type = visualization_msgs::Marker::SPHERE;
  marker.action = visualization_msgs::Marker::ADD;
  marker.pose.position.x = x;
  marker.pose.position.y = y;
  marker.pose.position.z = z;
  marker.pose.orientation.x = 0.0;
  marker.pose.orientation.y = 0.0;
```

```
  marker.pose.orientation.z = 0.0;
  marker.pose.orientation.w = 1.0;
  marker.scale.x = 0.2;
  marker.scale.y = 0.2;
  marker.scale.z = 0.2;
  marker.color.a = 1.0;
  marker.color.r = 1.0;
  marker.color.g = 0.0;
  marker.color.b = 0.0;
  //only if using a MESH_RESOURCE marker type:
  markerpub_.publish( marker );
}

void publishBbox()
{
  double x = (min_x_ + max_x_)/2;
  double y = (min_y_ + max_y_)/2;
  double z = (0 + max_z_)/2;

  double scale_x = (max_x_ - x)*2;
  double scale_y = (max_y_ - y)*2;
  double scale_z = (max_z_ - z)*2;

  visualization_msgs::Marker marker;
  marker.header.frame_id = "/camera_rgb_optical_frame";
  marker.header.stamp = ros::Time();
  marker.ns = "my_namespace";
  marker.id = 1;
  marker.type = visualization_msgs::Marker::CUBE;
  marker.action = visualization_msgs::Marker::ADD;
  marker.pose.position.x = x;
  marker.pose.position.y = -y;
  marker.pose.position.z = z;
  marker.pose.orientation.x = 0.0;
  marker.pose.orientation.y = 0.0;
  marker.pose.orientation.z = 0.0;
  marker.pose.orientation.w = 1.0;
  marker.scale.x = scale_x;
  marker.scale.y = scale_y;
  marker.scale.z = scale_z;
  marker.color.a = 0.5;
  marker.color.r = 0.0;
  marker.color.g = 1.0;
  marker.color.b = 0.0;
  //only if using a MESH_RESOURCE marker type:
  bboxpub_.publish( marker );
```

```
  }

  ros::Subscriber sub_;
  ros::Publisher cmdpub_;
  ros::Publisher markerpub_;
  ros::Publisher bboxpub_;
  ros::Publisher ears_;
};

PLUGINLIB_DECLARE_CLASS(turtlebot_follower, TurtlebotFollower,
turtlebot_follower::TurtlebotFollower, nodelet::Nodelet);

}
```

Ear Servo Control Code

```
#if (ARDUINO >= 100)
 #include <Arduino.h>
#else
 #include <WProgram.h>
#endif

#include <Servo.h>
#include <ros.h>
#include <std_msgs/UInt16.h>

uint16_t state;

void messageRead(const std_msgs::UInt16& msg) {
  state = msg.data;
}

ros::NodeHandle nh;

Servo servoLeft;
Servo servoRight;

ros::Subscriber<std_msgs::UInt16>sub("turtlebot_follower/ears",&messageRead;

void setup() {
  Serial.begin(9600);
  servoLeft.attach(9);
  servoRight.attach(10);
  nh.initNode();
  nh.subscribe(sub);
}
```

```
void loop() {
  if(state==1)
  {
    dogFollow();
  }
  else if(state==0)
  {
    dogStay();
  }
  nh.spinOnce();
  delay(1);
}

void dogFollow() {
  servoLeft.write(159);
  servoRight.write(74);
  delay(250);
  servoLeft.write(151);
  servoRight.write(82);
  delay(250);
}

void dogStay() {
  servoLeft.write(175);
  servoRight.write(45);
}

void dogRollOver() {
  servoLeft.write(120);
  servoRight.write(100);
}
```