UNITED STATES MILITARY ACADEMY

LAB 3 REPORT - EMBEDDED SYSTEMS WITH ROS

EE487: EMBEDDED SYSTEMS DEVELOPMENT
SECTION T2

COL KORPELA

BY

CDT NICHOLAS E. LIEBERS '23, C-2

WEST POINT, NY 10996

04 MAR 2022

___ DOCUMENTATION IDENTIFIES ALL SOURCES USED AND ASSISTANCE RECEIVED IN COMPLETING THIS ASSIGNMENT.

___ NO SOURCES WERE USED OR ASSISTANCE RECEIVED IN COMPLETING THIS ASSIGNMENT.

SIGNATURE: _____

Contents

1. Introduction

1.1. Purpose

The purpose of this lab is to introduce cadets to Robot Operating System (ROS), control and program robots using ROS, and use ROS to interface with other embedded systems. ROS provides the ability for the user to send messages to provide services across a network, USB wire, or on an Arduino. In this lab, ROS communicated across the network and between embedded Arduino devices. The purpose was accomplished, and the following learning objectives were met:

1.2. Learning Objectives

1. Configure nodes to exchange information via topics within publisher/subscriber frame-work of ROS.
2. Write a custom subscriber node in ROS to listen to a topic and perform computation when new information is published to that topic.
3. Learn how to interact and control a small, mobile robot such as the Turtlebot.
4. Network a remote workstation to a mobile robot within the ROS environment so that messages can be wirelessly exchanged between nodes on either computer.
5. Leverage pre-existing hardware drivers and other nodes in the ROS repository and integrate them together with custom-written nodes as part of robot application.

2. Theory

In order to complete this lab, we must define what ROS at a high level, introduce the Turltebot3 robot, and discuss Rosserial.

2.1. What is ROS?

ROS is a high level interface that allows researchers to program and control robots that connect to various forms of equipment that can communicate using one common interface; ROS is the common interface in this case. Figure 1 illustrates how different ROS publishing and subscribing nodes communicate with one another[1].

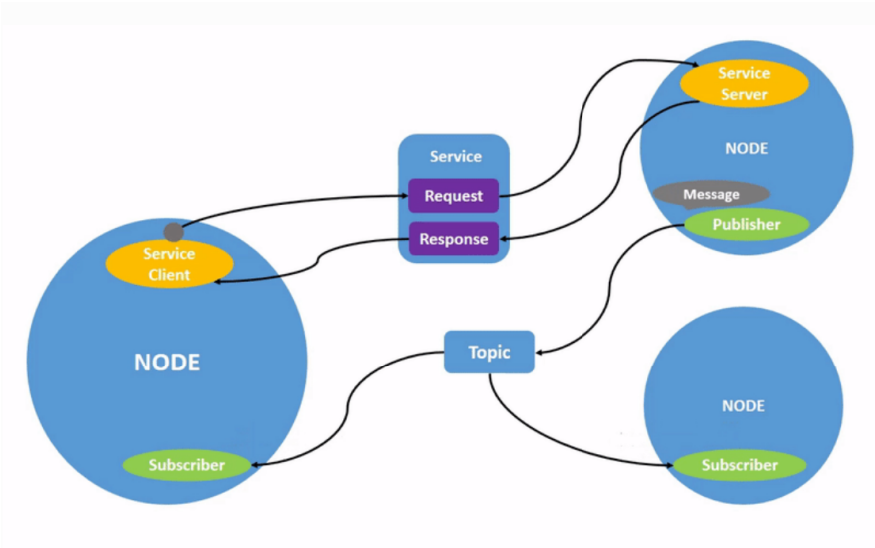

Figure 1: A simple diagram that outlines how Nodes, Topics, Messages, Publishers, and Subscribers relate to one another. The Publishers publish a message over a Topic, and the message is received/read by a subscriber [6].

[1]For a more detailed explanation of ROS and its functions, please refer to Lab2 at `https://usarmywestpoint-my.sharepoint.com/:b:/g/personal/nicholas_liebers_westpoint_edu/ET1MzeZ6AwdMk4yvUkP3h_kB1RR_Q6jM5vNuGU5WSEB6wQ?e=bEyap3`.

2.2. What is the TurtleBot3?

Turtlebot3 is a ROS robot that was developed in 2017 that is designed for use in education, research, and product prototyping. Turtlebot3 is simultaneous localization and mapping (SLAM) capable and is webcam capable. As seen in Figure 2, the Turtlebot3 comes equipped with an OPCR1.0 for power management and a Rasberry Pi that runs an Ubnutu OS. Also, the Turtlebot3 design and core components are open source[2] [3].

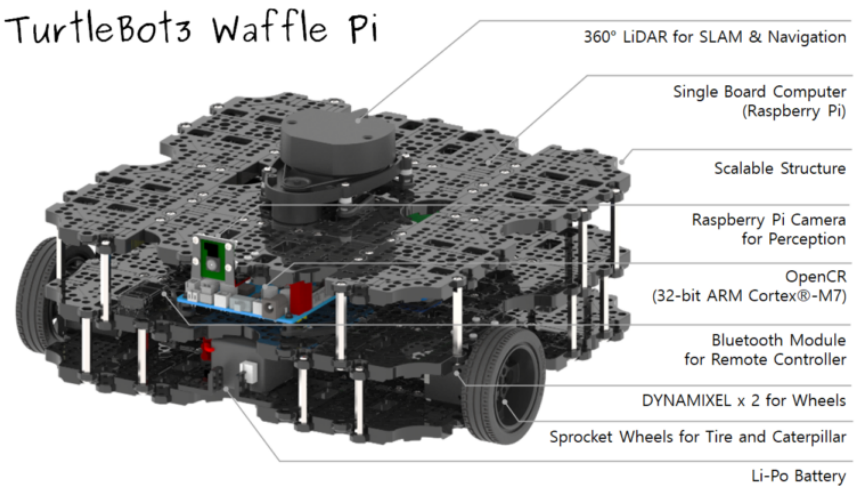

Figure 2: This is the TurtleBot3 with labeled components Pi [3].

2.3. Rosserial Introduction

Rosserial is a protocol that packages ROS messages and information onto a serial port for distribution. Using Rosserial_python, we can set up a node that can publish and subscribe for our rosserial device–an Arduino Uno in our case.

To run a Rosserial on a file over a port, the following command is run:

```
rosrun rosserial_python sampleFile.py /dev/ttyACM1
```

[1, 7]

[2]See `https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications` for open source 3D models.

3. Experimental Setup and Procedure

3.1. Required Equipment

1. Computer running Linux with proper software installed to interface with Turtlebot3[3].
2. A Turtlebot3.
3. An Arduino Uno.
4. SHARP IR Range Finder [2].
5. USB cables and wires for Arduino and range finder.

3.2. Software Setup

This lab runs ROS on a computer, the Turtlebot3, and the Arduino.

3.2.1. ROS Setup for Computer

The following command is required to install the ROS packages for the TurtleBot3:[4]

```
$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
  ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
  ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
  ros-noetic-rosserial-python ros-noetic-rosserial-client \
  ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
  ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
  ros-noetic-compressed-image-transport ros-noetic-rqt* ros-noetic-rviz \
  ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-markers
```

[4]

To run ROS on the computer, the following command is used:

```
# roscore
```

[3] see `https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/#pc-setup` for required packages.
[4] see `https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/#pc-setup` for guided tutorial.

To enable control the movement of the Turtlebot3 from the computer, run the following command:

```
# export TURTLEBOT3_MODEL=waffle
# roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

[4]

To view the incoming camera footage from the Turtlebot3, run the following command:[5]

```
# rqt_image_view
```

[4]

3.2.2. ROS Setup on the TurtleBot3

The Turtlebot3 comes fully equipped with ROS; however, some configuration was required in order to get full functionality.

The TurtleBot3 Setup guide on the West Point Robotics Github offers configuration details for how we setup our robot [5].

While SSHing into the Turltebot3, we can run the bringup file. The bringup ensures all of the correct nodes are up and running on the Turtlebot3. Run the following command:

```
# roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

[4]

To run the camera node, run the following command:

```
# roslaunch raspicam_node camerav2_1280x960.launch
```

[5]

3.2.3. Rosserial Installation

Rosserial is required to be installed on the TurtleBot3 in order to run the Rosserial program from the embedded system.

Appendix A contains the code that is installed on the Arduino and run to range finding process. To run the file with Rosserial, the following command was used:

```
# rosrun rosserial_python sampleFile.py /dev/ttyACM1
```

[1, 7]

[5] see `https://github.com/westpoint-robotics/usma_turtlebot` for installation process on desktop and Turltebot3.

3.3. Physical Setup

The physical setup for this lab was simple since the Turltebot3 is already built for us. The only physical installation required was to connect the range finder to the Arduino Uno and plug in the Ardiuno Uno into the Turltebot3 via USB. Figure 3 shows how to setup the Arduino Uno to be connected to the IR rangefinder.

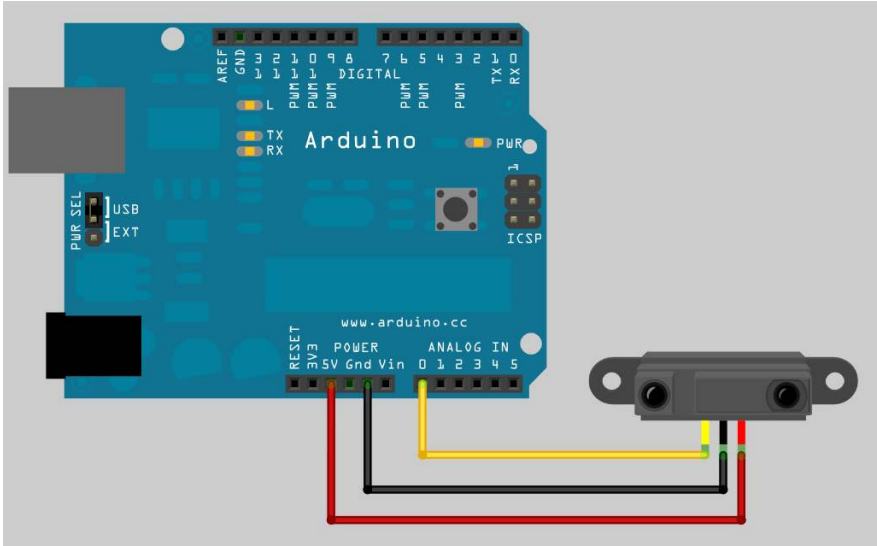

Figure 3: This figure illustrates how the range finder was connected to the Ardunio Uno [7] .

Figure 4 is the final setup of the Turtlebot3 and the Arduino Uno connected and operating together.

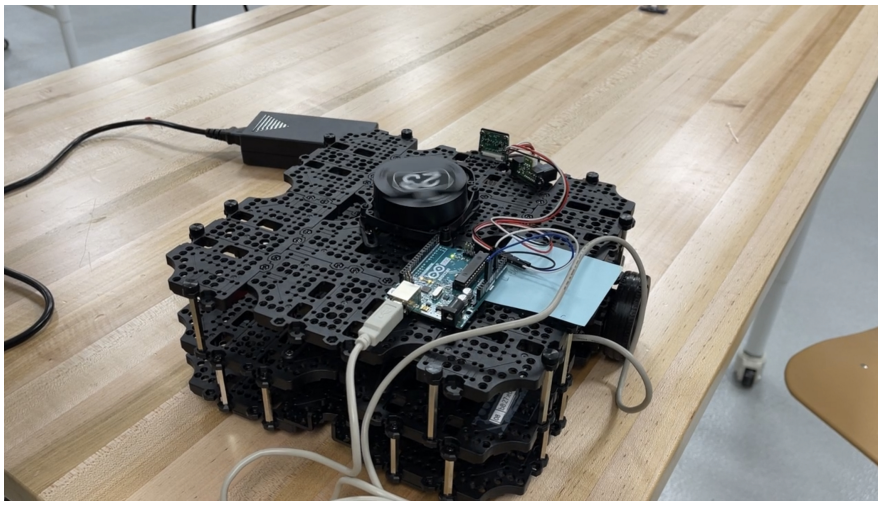

Figure 4: This is the final experimental setup where the Arduino Uno is connected to the Turtlebot3 via USB.

4. Results and Discussion

The Turtlebot3 was able to move using a keyboard, transmit video, and echo/display range finding data to the terminal. These results are repeatable.

4.0.1. Errors and Issues

The largest issue that occurred while completing this lab was installing Rossieral onto the Arduino Uno so it could run the rangefinder program. When installing, the Arduino would "hang" and froze up. As a result, I attempted to delete some files in the catkin_ws folder to fix the issues; however, I ended up corrupting the ROS project.

4.0.2. Solutions to Errors

To fix my issue, I grabbed an unused Turtlebot3 and copied its catkin_ws folder onto my Turltebot3. After doing this, I repeated parts of the tutorial, downloaded any packages if they were missing, and rebuilt the correct catkin_ws folder. Afterwards, I created a backup of my catkin_ws folder and attempted to reinstall the Rosserial using a different method from the tutorial [7]. This different method was successful.

4.0.3. Incremental Software Building Process

This project consists of three parts:

1. The Computer.
2. The Arduino Uno with range finder.
3. The Turtlebot3.

I first configured my computer environment to support ROS and interface with the Turtlebot3. Various, smaller, problems arose during this process; however, by solely focusing on this part, I was able to focus my time on getting the basic Turtlebot3 functions to work.

Next, I began testing the range finder functionality. I used the desktop computer to program the Arduino and test before connecting to the Turltebot3. There were some small issues I encountered–such as faulty wires

disrupting the range finder's transmissions–that were quickly solved since my focus was set on the Ardunio Uno. After getting the range finder program to run, I then moved the Ardiuno Uno to the Turtlebot3.

Since I conquered this lab in those two parts, I was able to find and solve difficult bugs without dealing with too many different devices.

5. Conclusion and Recommendations

In conclusion, this lab was successful. I was able to complete the required objectives for this Lab. An important take way for this lab is that ROS is highly dependant on it being configured correctly. While ROS gives users a high level ability to program robots and read different data from sensors, if its underlying dependencies are not configured correctly, difficult error will arise and the whole implementation will fall apart. In this lab, I spent an enormous amount of time ensuring that the proper dependencies were installed so that the final implementation would be error free.

6. References

[1] Code listing - Overleaf, Online LaTeX Editor.

[2] Infrared Proximity Sensor Short Range - Sharp GP2Y0A41SK0F - SEN-12728.

[3] TurtleBot3, 2022.

[4] TurtleBot3 Setup and Tutorials, 2022.

[5] Turtlebot3 Setup West Point Robotics, 2 2022.

[6] Understanding ROS 2 Nodes, 2022.

[7] Phil Glau. Rosserial_arduino/Tutorials/IR Ranger - ROS Wiki, 2 2017.

Appendix A - Launch File

0.1. Rosserial Code

```
1
2  #include <ros.h>
3  #include <ros/time.h>
4  #include <sensor_msgs/Range.h>
5
6  ros::NodeHandle  nh;
7
8
9  sensor_msgs::Range range_msg;
10 ros::Publisher pub_range( "range_data", &range_msg);
11
12 const int analog_pin = 0;
13 unsigned long range_timer;
14
15 float getRange(int pin_num){
16     int sample;
17     // Get data
18     sample = analogRead(pin_num)/4;
19     // if the ADC reading is too low,
20     //   then we are really far away from anything
21     if(sample < 10)
22         return 254;     // max range
23     // Magic numbers to get cm
24     sample= 1309/(sample-3);
25     return sample; //convert to meters
26 }
27
28 char frameid[] = "/ir_ranger";
29
30 void setup()
```

```
{
  nh.initNode();
  nh.advertise(pub_range);

  range_msg.radiation_type = sensor_msgs::Range::INFRARED;
  range_msg.header.frame_id =  frameid;
  range_msg.field_of_view = 0.01;
  range_msg.min_range = 0.03;
  range_msg.max_range = 0.4;

}

void loop()
{
  // publish the range value every 50 milliseconds
  //   since it takes that long for the sensor to stabilize
  if ( (millis()-range_timer) > 50){
    range_msg.range = getRange(analog_pin);
    range_msg.header.stamp = nh.now();
    pub_range.publish(&range_msg);
    range_timer =  millis() + 50;
  }
  nh.spinOnce();
}
```

NOTE: This code is copied from the source. There were very few required additions to this code. Therefore, all but one or two lines are exactly from the source [7].