

9

Interfacing I/O Boards, Sensors, and Actuators to ROS

In the previous chapters, we discussed different kinds of plugin frameworks that are used in ROS. In this chapter, we are going to discuss the interfacing of some hardware components, such as sensors and actuators, to ROS. We will look at the interfacing of sensors using I/O boards, such as Arduino, Raspberry Pi, and Odroid-XU4 to ROS, and we will discuss interfacing smart actuators, such as DYNAMIXEL, to ROS. The following is the detailed list of topics that we are going to cover in this chapter

- Understanding the Arduino-ROS interface
- Setting up the Arduino-ROS interface packages
- Arduino-ROS examples: Chatter and Talker, blink LED and push button, Accelerometer ADXL 335, ultrasonic distance sensors, and Odometry Publisher
- Interfacing a non-Arduino board to ROS
- Setting ROS on Odroid-XU4 and Raspberry Pi 2
- Working with Raspberry Pi and Odroid GPIOs using ROS
- Interfacing DYNAMIXEL actuators to ROS

Understanding the Arduino-ROS interface

Let's see what Arduino is first. Arduino is one of the most popular open source development boards in the market. The ease of programming and the cost effectiveness of the hardware have made Arduino a big success. Most of the Arduino boards are powered by Atmel microcontrollers, which are available from 8-bit to 32-bit, with clock speeds from 8 MHz to 84 MHz. Arduino can be used for the quick prototyping of robots. The main applications of Arduino in robotics are interfacing sensors and actuators, used for communicating with PCs for receiving high-level commands and sending sensor values to PCs using the `UART` protocol.

There are different varieties of Arduino available in the market. Selecting one board for our purpose will be dependent on the nature of our robotic application. Let's see some boards which we can use for beginners, intermediate, and high-end users:

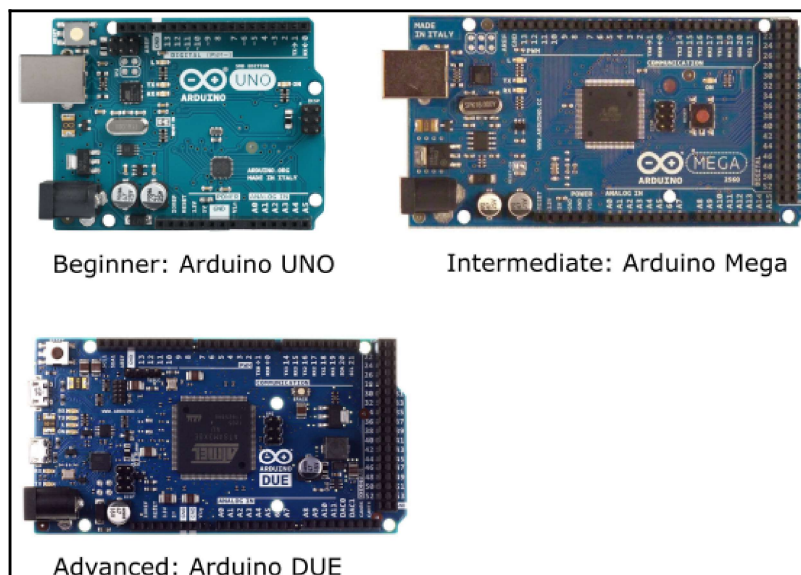


Figure 1: Different versions of the Arduino board

In the following table, we will look at each Arduino board specification in brief and see where it can be deployed:

Boards	Arduino UNO	Arduino Mega 2560	Arduino Due
Processor	ATmega328P	ATmega2560	ATSAM3X8E
Operating/Input Voltage	5V / 7-12 V	5V/ 7-12V	3.3V / 7 - 12 V

CPU Speed	16 MHz	16 MHz	84 MHz
Analog In/Out	6/0	16/0	12/2
Digital IO/PWM	14/6	54/15	54/12
EEPROM[KB]	1	4	-
SRAM [KB]	2	8	96
Flash [KB]	32	256	512
USB	Regular	Regular	2 Micro
UART	1	4	4
Application	Basic robotics and sensor interfacing	Intermediate robotic application-level application	High-end robotics application

Let's look at how to interface Arduino to ROS.

What is the Arduino-ROS interface?

Most of the communication between PCs and I/O boards in robots will be through the UART protocol. When both the devices communicate with each other, there should be some program in both the sides that can translate the serial commands from each of these devices. We can implement our own logic to receive and transmit the data from board to PC and vice versa. The interfacing code can be different in each I/O board because there are no standard libraries to do this communication.

The Arduino-ROS interface is a standard way of communication between the Arduino boards and the PC. Currently, this interface is exclusive for Arduino. We may need to write custom nodes to interface with other I/O boards.

We can use the similar C++ APIs of ROS used in the PC in the Arduino IDE also for programming the Arduino board. Detailed information about the interfacing package follows.

Understanding the roserial package in ROS

The `roserial` package is a set of standardized communication protocols implemented for communicating from ROS to character devices, such as serial ports, and sockets, and vice versa. The `roserial` protocol can convert the standard ROS messages and services data types to embedded device equivalent data types. It also implements multi-topic support by multiplexing the serial data from a character device. The serial data is sent as data packets by adding header and tail bytes on the packet. The packet representation is shown next:

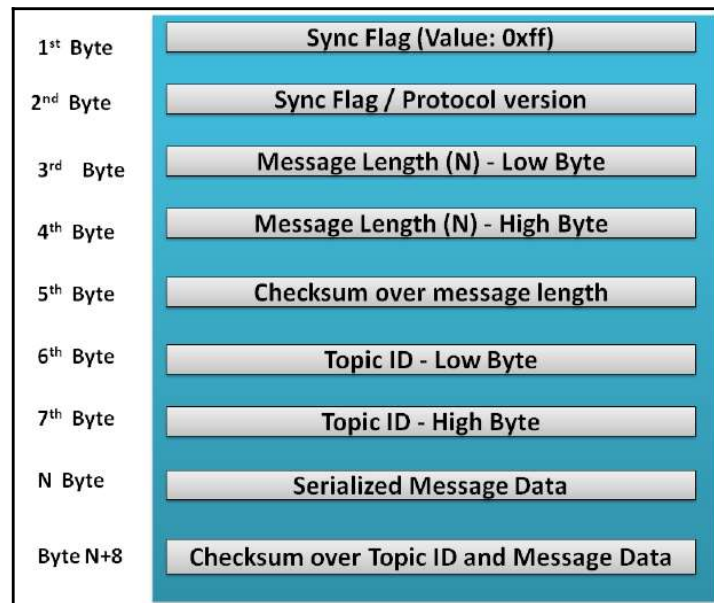


Figure 2: roserial packet representation

The function of each byte follows:

- **Sync Flag:** This is the first byte of the packet, which is always `0xff`
- **Sync Flag/Protocol version:** This byte was `0xff` on ROS Groovy and after that it is set to `0xfe`
- **Message Length:** This is the length of the packet
- **Checksum Over Message Length:** This is the checksum of length for finding packet corruption
- **Topic ID:** This is the ID allocated for each topic; the range 0–100 is allocated for the system-related functionalities

- **Serialized Message data:** This is the data associated with each topic
- **Checksum of Topic ID and Message data:** This is the checksum for the topic and its serial data for finding the packet, `corruption`

The checksum of length is computed using the following equation:

$$\text{Checksum} = 255 - ((\text{Topic ID Low Byte} + \text{Topic ID High Byte} + \dots \text{data byte values}) \% 256)$$

The ROS client libraries, such as `roscpp`, `rospy`, and `roslisp`, enable us to develop ROS nodes that can run from various devices. One of the ports of the ROS clients that enables us to run a ROS node from the embedded devices, such as Arduino and embedded Linux based boards, is called the `roserial_client` library. Using the `roserial_client` libraries, we can develop the ROS nodes from Arduino, embedded Linux platforms, and Windows. The following is the list of `roserial_client` libraries for each of these platforms:

- `roserial_arduino`: This `roserial_client` works on Arduino platforms, such as Arduino UNO, Leonardo, Mega, and Due series for advance robotic projects
- `roserial_embeddedlinux`: This client supports embedded Linux platforms, such as VEXPro, Chumby alarm clock, WRT54GL router, and so on
- `roserial_windows`: This is a client for the Windows platform

In the PC side, we need some other packages to decode the serial message and convert to exact topics from the `roserial_client` libraries. The following packages help in decoding the serial data:

- `roserial_python`: This is the recommended PC-side node for handling serial data from a device. The receiving node is completely written in Python.
- `roserial_server`: This is a C++ implementation of `roserial` in the PC side. The inbuilt functionalities are less compared to `roserial_python`, but it can be used for high-performance applications.

We are mainly focusing on running the ROS nodes from Arduino. First, we will see how to set up the `roserial` packages, and then discuss how to set up the `roserial_arduino` client in the Arduino IDE.

Installing roserial packages on Ubuntu 16.04

To operate with Arduino on Ubuntu 16.04, we must install `roserial` ROS packages and then set up the Arduino client, installing the libraries needed to communicate with ROS. We can install the `roserial` packages on Ubuntu using the following commands:

1. Install the `roserial` package binaries, using `apt-get`:

```
$ sudo apt-get install ros-kinetic-roserial-arduino ros-kinetic-  
roserial-embeddedlinux ros-kinetic-roserial-windows ros-kinetic-  
roserial-server ros-kinetic-roserial-python
```

2. For installing the `roserial_client` library called `ros_lib` in Arduino, we must download the latest Arduino IDE for Linux 32/64 bit. The following is the link for downloading the Arduino IDE: <https://www.arduino.cc/en/main/software>. Here we download the Linux 64-bit version and copy the Arduino IDE folder to the Ubuntu desktop. Arduino requires Java runtime support to run it. If it is not installed, we can install it using the following command:

```
$ sudo apt-get install java-common
```

3. After installing Java runtime, we can switch the `arduino` folder using the following command:

```
$ cd ~/Desktop/arduino-1.8.5
```

4. Start Arduino, using the following command:

```
$ ./arduino
```

Shown next is the Arduino IDE window:



Figure 3: Arduino IDE

5. Go to **File | Preference** for configuring the sketchbook folder of Arduino. The Arduino IDE stores the sketches to this location. We created a folder called `Arduino1` in the user home folder and set this folder as the **Sketchbook location**:

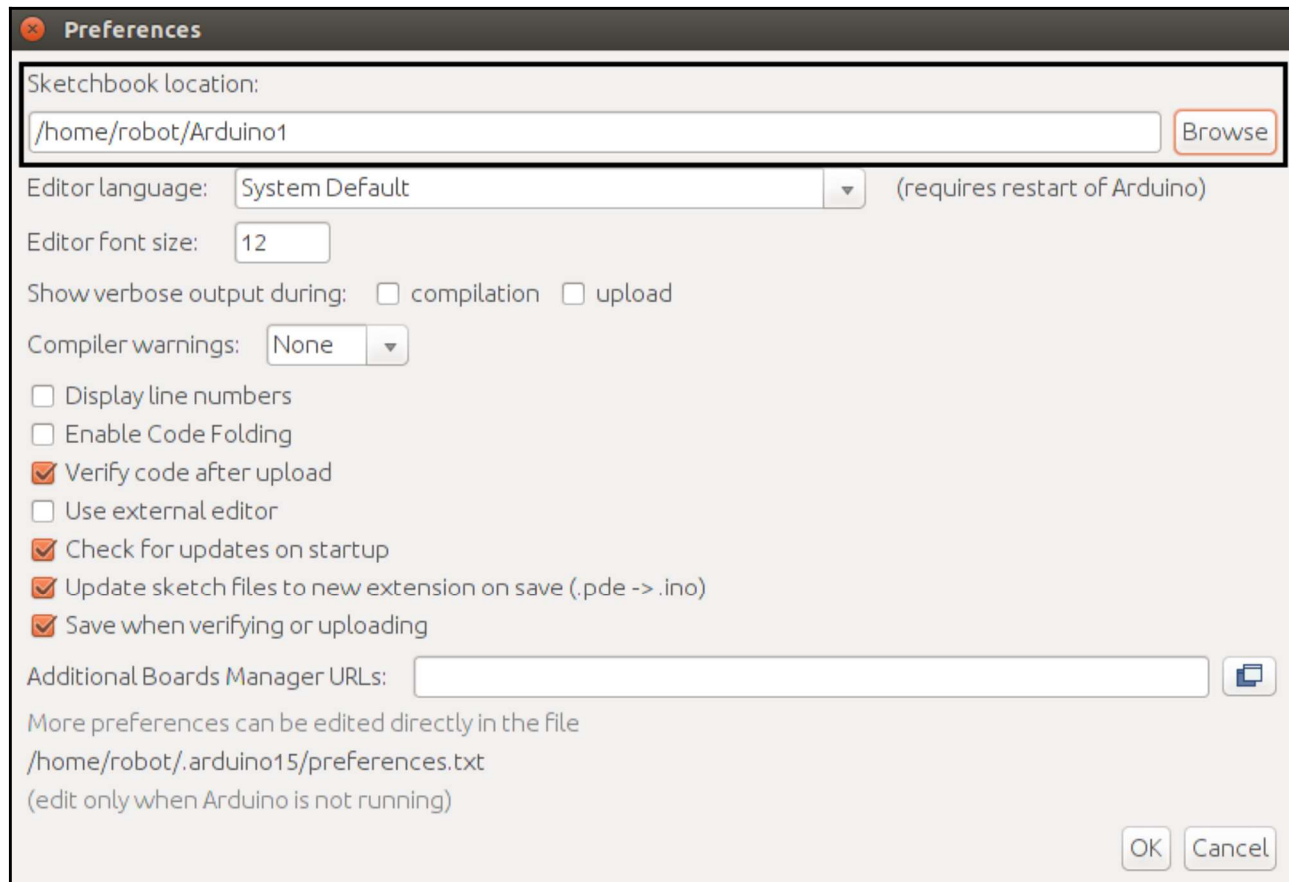


Figure 4: Preferences of Arduino IDE

We can see a folder called `libraries` inside the `Arduino1` folder. Switch to this folder, using the following command:

```
$ cd ~/Arduino1/libraries/
```

If there is no `libraries` folder, we can create a new one. After switching into this folder, we can generate `ros_lib`, using a script called `make_libraries.py`, which is present inside the `roserial_arduino` package. `ros_lib` is the `roserial_client` for Arduino, which provides the ROS client APIs inside an Arduino IDE environment:

```
$ rosrun roserial_arduino make_libraries.py
```

`roserial_arduino` is the ROS client for `arduino`, which can communicate using UART, and can publish topics, services, TF, and so on, like a ROS node. The `make_libraries.py` script will generate a wrapper of the ROS messages and services which are optimized for Arduino data types. These ROS messages and services will convert into Arduino C/C++ code equivalent, as shown next:

- Conversion of ROS messages:

```
ros_package_name/msg/Test.msg --> ros_package_name::Test
```

- Conversion of ROS services:

```
ros_package_name/srv/Foo.srv --> ros_package_name::Foo
```

For example, if we include `#include <std_msgs/UInt16.h>`, we can instantiate the `std_msgs::UInt16` number.

If the `make_libraries.py` script works fine, a folder called `ros_lib` will generate inside the `libraries` folder. Restart the Arduino IDE and we will see `ros_lib` examples as follows:

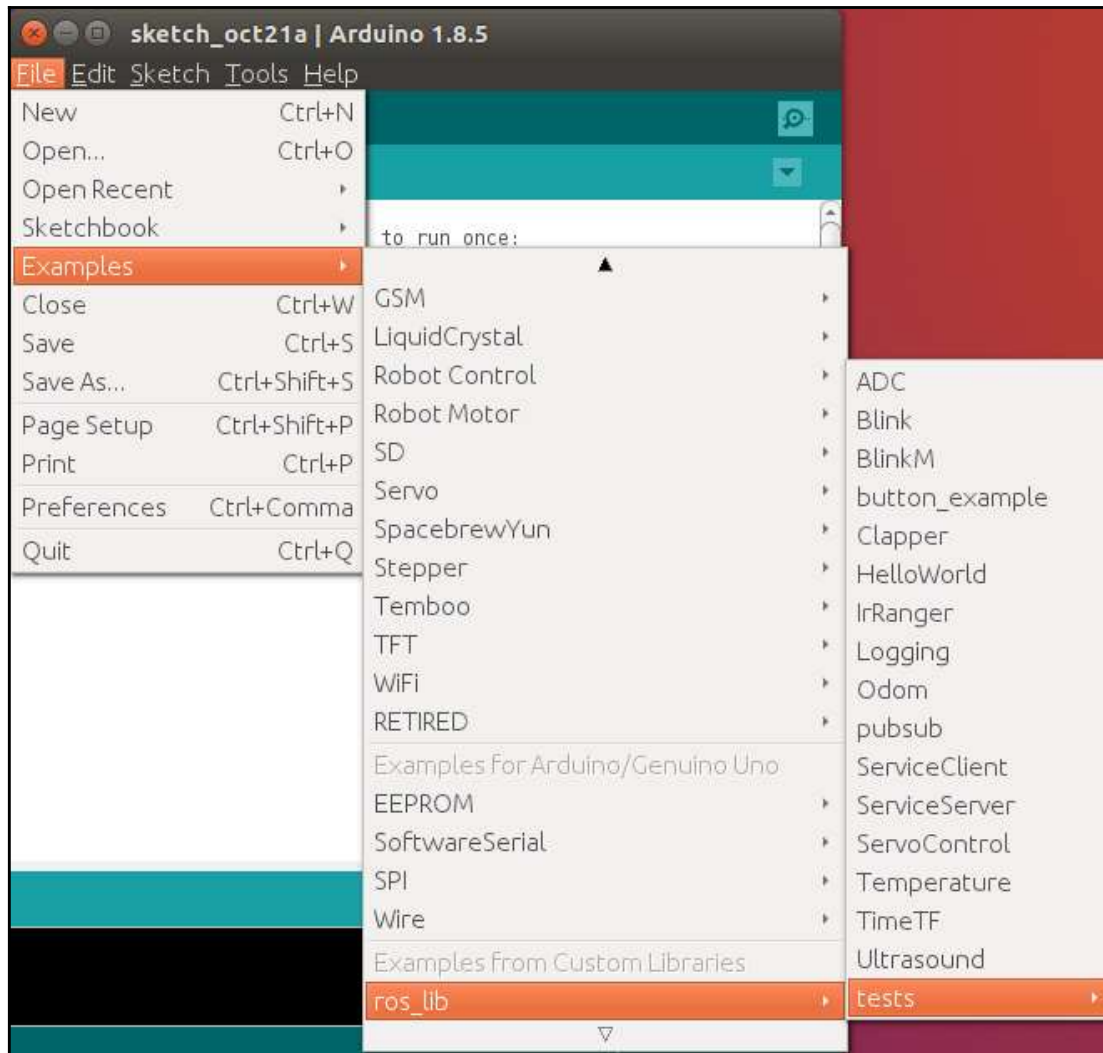


Figure 5: List of Arduino - `ros_lib` examples

We can take any example and make sure that it is building properly to ensure that the `ros_lib` APIs are working fine. The necessary APIs required for building ROS Arduino nodes are discussed next.

Understanding ROS node APIs in Arduino

The following is a basic structure of the ROS Arduino node. We can see the function of each line of code:

```
#include <ros.h>

ros::NodeHandle nh;

void setup() {
    nh.initNode();
}

void loop() {
    nh.spinOnce();
}
```

The creation of `NodeHandle` in Arduino is done using the following line of code:

```
ros::NodeHandle nh;
```

Note that `Nodehandle` should be declared before the `setup()` function, which will give a global scope to the `NodeHandle` instance called `nh`. The initialization of this node is done inside the `setup()` function:

```
nh.initNode();
```

The Arduino `setup()` function will execute only once when the device starts. Note that we can only create one node from a serial device.

Inside the `loop()` function, we have to use the following line of code to execute the ROS callback once:

```
nh.spinOnce();
```

We can create the `Subscriber` and `Publisher` objects in Arduino, like the other ROS client libraries. The following are the procedures for defining the subscriber and the publisher.

Here is how we define a subscriber object in Arduino:

```
ros::Subscriber<std_msgs::String> sub("talker", callback);
```

Here we define a subscriber which is subscribing a `String` message, where the callback is the callback function executing when a `String` message arrives on the talker topic. Given next is an example callback for handling the string data:

```
std_msgs::String str_msg;

ros::Publisher chatter("chatter", &str_msg);

void callback ( const std_msgs::String& msg){
    str_msg.data = msg.data;

    chatter.publish( &str_msg );
}
```

Note that the `callback()`, `Subscriber`, and `Publisher` definitions will be above the `setup()` function for getting the global scope. Here we are receiving `String` data, using `const std_msgs::String& msg`.

The following code shows how to define a publisher object in Arduino:

```
ros::Publisher chatter("chatter", &str_msg);
```

This next code shows how we publish the string message:

```
chatter.publish( &str_msg );
```

After defining the publisher and the subscriber, we have to initiate this inside the `setup()` function, using the following lines of code:

```
nh.advertise(chatter);
nh.subscribe(sub);
```

There are ROS APIs for logging from Arduino. The following are the different logging APIs supported:

```
nh.logdebug("Debug Statement");
nh.loginfo("Program info");
nh.logwarn("Warnings.");
nh.logerror("Errors..");
nh.logfatal("Fatalities!");
```

We can retrieve the current ROS time in Arduino using ROS built-in functions, such as time and duration.

- Current ROS time:

```
ros::Time begin = nh.now();
```

- Converting ROS time in seconds:

```
double secs = nh.now().toSec();
```

- Creating a duration in seconds:

```
ros::Duration ten_seconds(10, 0);
```

ROS - Arduino Publisher and Subscriber example

The first example using the Arduino and ROS interface is a chatter and talker interface. Users can send a `String` message to the `talker` topic and Arduino will publish the same message in a `chatter` topic. The following ROS node is implemented for Arduino, and we will discuss this example in detail:

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

//Creating Nodehandle
ros::NodeHandle nh;

//Declaring String variable
std_msgs::String str_msg;

//Defining Publisher
ros::Publisher chatter("chatter", &str_msg);
//Defining callback
void callback ( const std_msgs::String& msg){

    str_msg.data = msg.data;
    chatter.publish( &str_msg );
}

//Defining Subscriber
ros::Subscriber<std_msgs::String> sub("talker", callback);

void setup()
```

```
{
  //Initializing node
  nh.initNode();
  //Start advertising and subscribing
  nh.advertise(chatter);
  nh.subscribe(sub);
}

void loop()
{
  nh.spinOnce();
  delay(3);
}
```

We can compile the preceding code and upload to the Arduino board. After uploading the code, select the desired Arduino board that we are using for this example and the device serial port of the Arduino IDE.

Go to **Tools | Boards** to select the board and **Tools | Port** to select the device port name of the board. We are using Arduino Mega for these examples.

After compiling and uploading the code, we can start the ROS bridge nodes in the PC that connects Arduino and the PC, using the following command. Ensure that Arduino is already connected to the PC before executing this command:

```
$ rosrun roserial_python serial_node.py /dev/ttyACM0
```

In this case, we are running the `serial_node.py` on the port `/dev/ttyACM0`. We can search for the port name listing the contents of the `/dev` directory. Note that, to use this port, root permissions are needed. In this case, we could change the permissions using the following command in order to read and write data on the desired port:

```
$ sudo chmod 666 /dev/ttyACM0
```

We are using the `roserial_python` node here as the ROS bridging node. We have to mention the device name and baud-rate as arguments. The default baud-rate of this communication is 57600. We can change the baud-rate according to our application and the usage of `serial_node.py` inside the `roserial_python` package is given at http://wiki.ros.org/roserial_python. If the communication between the ROS node and the Arduino node is correct, we will get the following message:

```
[INFO] [WallTime: 1438880620.972231] ROS Serial Python Node
[INFO] [WallTime: 1438880620.982245] Connecting to /dev/ttyACM0 at 57600 baud
[INFO] [WallTime: 1438880623.117417] Note: publish buffer size is 512 bytes
[INFO] [WallTime: 1438880623.118587] Setup publisher on chatter [std_msgs/String]
[INFO] [WallTime: 1438880623.132048] Note: subscribe buffer size is 512 bytes
[INFO] [WallTime: 1438880623.132745] Setup subscriber on talker [std_msgs/String]
```

Figure 6: Running the `rosserial_python` node

When `serial_node.py` starts running from the PC, it will send some serial data packets called query packets to get the number of topics, the topic names, and the types of topics which are received from the Arduino node. We have already seen the structure of serial packets that are being used for Arduino ROS communication. Given next is the structure of a query packet which is sent from `serial_node.py` to Arduino:

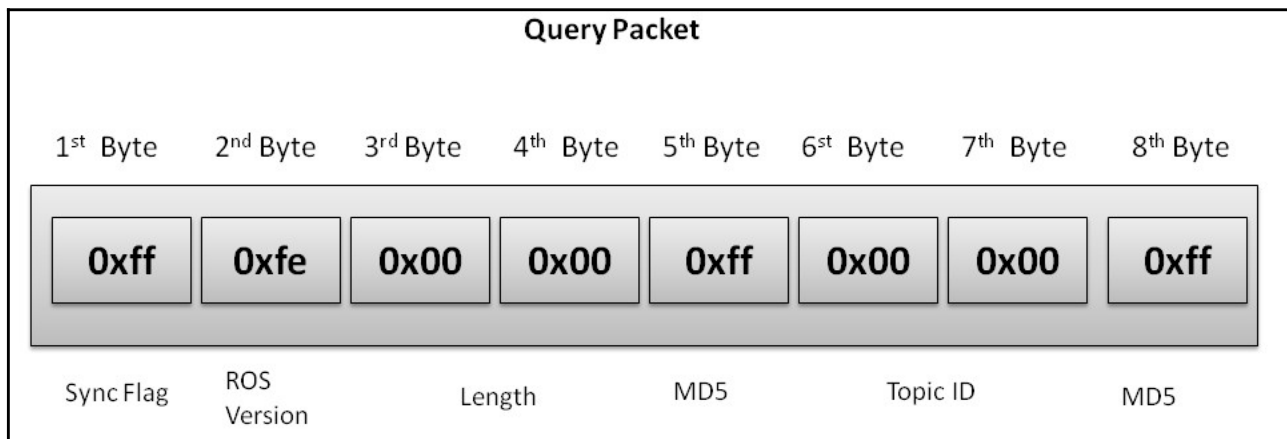


Figure 7: Structure of the query packet

The query topic contains fields such as Sync Flag, ROS version, length of the message, MD5 sum, Topic ID, and so on. When the query packet is received on the Arduino, it will reply with a topic info message that contains the topic name, type, length, topic data, and so on. The following is a typical response packet from Arduino:

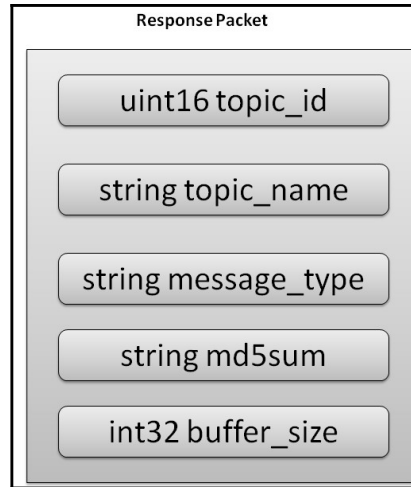


Figure 8: Structure of response packet

If there is no response for the query packet, it will send it again. The synchronization in communication is based on ROS time.

From Figure 6, we can see that when we run `serial_node.py`, the buffer size allocated for publish and subscribe is 512 bytes. The buffer allocation is dependent on the amount of RAM available on each microcontroller that we are working with. The following is a table showing the buffer allocation of each Arduino controller. We can override these settings by changing the `BUFFER_SIZE` macro inside `ros.h`.

AVR model	Buffer size	Publishers/Subscribers
ATMEGA 168	150 bytes	6/6
ATMEGA 328P	280 bytes	25/25
All others	512 bytes	25/25

There are also some limitations in the *float64* data type of ROS in Arduino. It will truncate to 32-bit. Also, when we use string data types, use the unsigned char pointer for saving memory.

After running `serial_node.py`, we will get the list of topics, using the following command:

```
$ rostopic list
```

We can see that topics such as `chatter` and `talker` are being generated. We can simply publish a message to the `talker` topic, using the following command:

```
$ rostopic pub -r 5 talker std_msgs/String "Hello World"
```

It will publish the "Hello World" message with a rate of 5.

We can echo the `chatter` topic, and we will get the same message as we published:

```
$ rostopic echo /chatter
```

Arduino-ROS, example - blink LED and push button

In this example, we can interface the LED and push button to Arduino and control using ROS. When the push button is pressed, the Arduino node sends a `True` value to a topic called `pushed`, and at the same time, it switches on the LED which is on the Arduino board.

The following shows the circuit for doing this example:

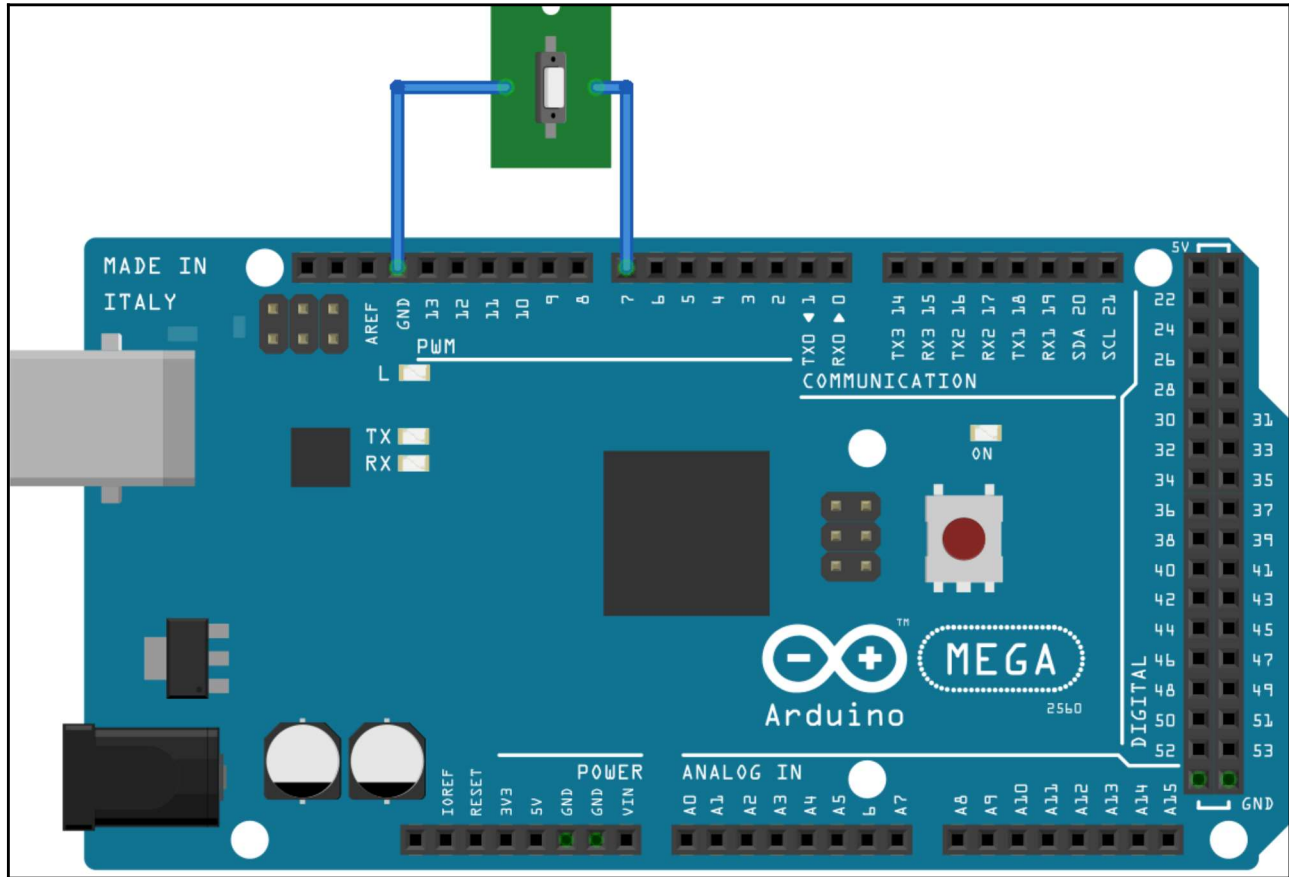


Figure 9: Interfacing the push button to Arduino

```

/*
 * Button Example for Rosserial
 */

#include <ros.h>
#include <std_msgs/Bool.h>

//Nodehandle
ros::NodeHandle nh;

//Boolean message for Push button
std_msgs::Bool pushed_msg;

//Defining Publisher in a topic called pushed
ros::Publisher pub_button("pushed", &pushed_msg);

```

```
//LED and Push button pin definitions
const int button_pin = 7;
const int led_pin = 13;

//Variables to handle debouncing
//https://www.arduino.cc/en/Tutorial/Debounce

bool last_reading;
long last_debounce_time=0;
long debounce_delay=50;
bool published = true;

void setup()
{
  nh.initNode();
  nh.advertise(pub_button);
  //initialize an LED output pin
  //and a input pin for our push button
  pinMode(led_pin, OUTPUT);
  pinMode(button_pin, INPUT);
  //Enable the pullup resistor on the button
  digitalWrite(button_pin, HIGH);
  //The button is a normally button
  last_reading = ! digitalRead(button_pin);
}

void loop()
{
  bool reading = ! digitalRead(button_pin);
  if (last_reading!= reading){
    last_debounce_time = millis();
    published = false;
  }
  //if the button value has not changed for the debounce delay, we know its
  stable

  if ( !published &&& (millis() - last_debounce_time) >
debounce_delay) {
    digitalWrite(led_pin, reading);
    pushed_msg.data = reading;
    pub_button.publish(&pushed_msg);
    published = true;
  }

  last_reading = reading;
  nh.spinOnce();
}
```

The preceding code handles the key debouncing and changes the button state only after the button release. The preceding code can upload to Arduino and can interface to ROS, using the following commands:

- Start `roscore`:

```
$ roscore
```

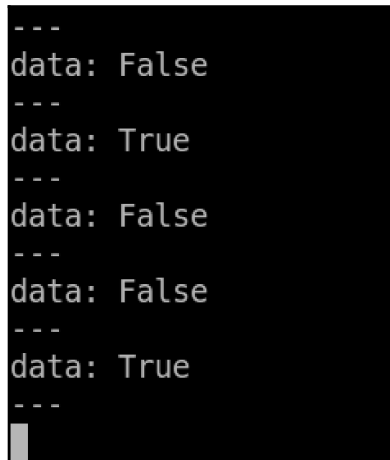
- Start `serial_node.py`:

```
$ rosrund roserial_python serial_node.py /dev/ttyACM0
```

We can see the button press event by echoing the topic pushed:

```
$ rostopic echo pushed
```

We will get following values when a button is pressed:



```
---  
data: False  
---  
data: True  
---  
data: False  
---  
data: False  
---  
data: True  
---  
data: 
```

Figure 10: Output of Arduino pushing button

Arduino-ROS, example - Accelerometer ADXL 335

In this example, we are interfacing Accelerometer ADXL 335 to Arduino Mega through ADC pins and plotting the values using the ROS tool called `rqt_plot`.

The following image shows the circuit of the connection between ADLX 335 and Arduino:

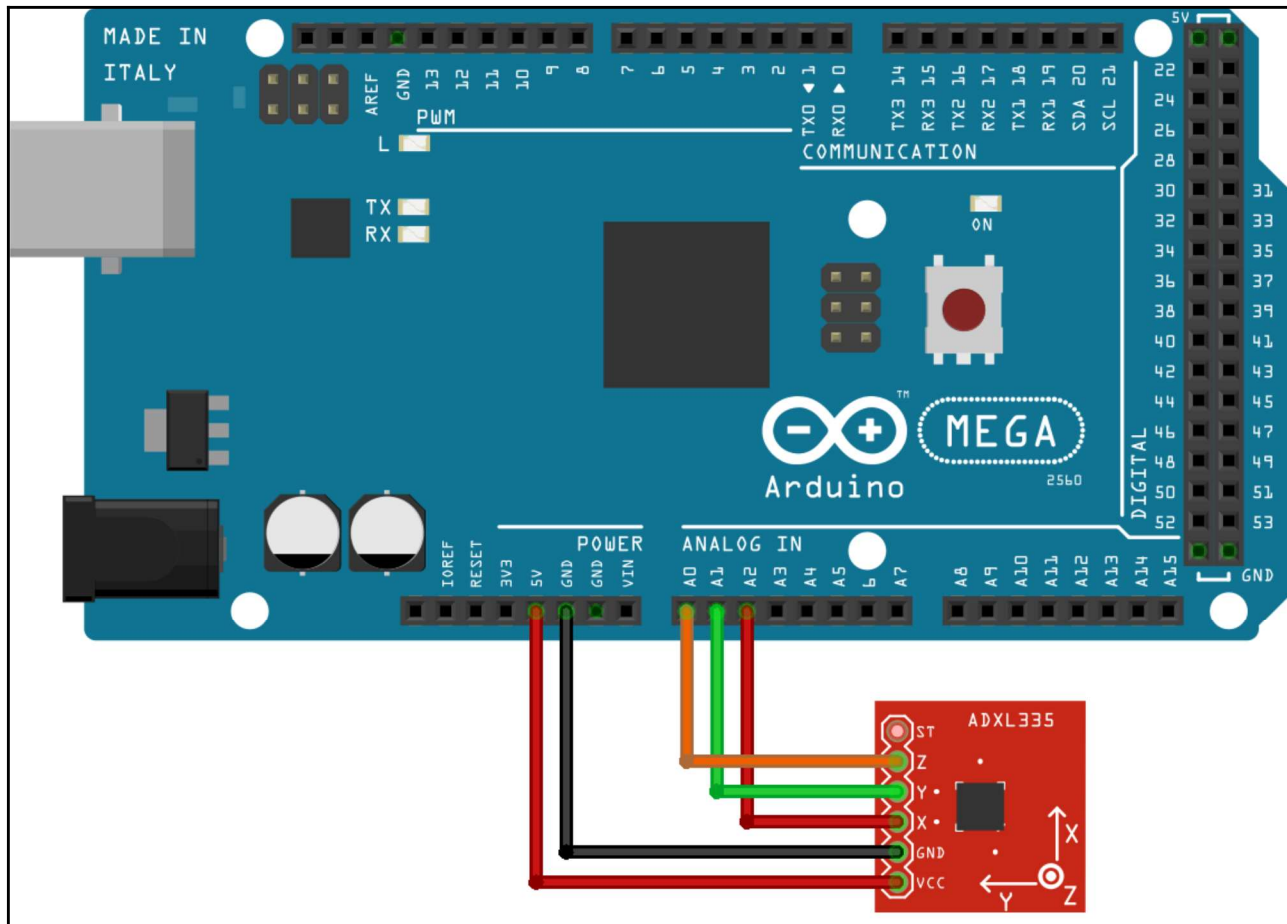


Figure 11: Interfacing Arduino - ADXL 335

ADXL 335 is an analog accelerometer. We can simply connect to the ADC port and read the digital value. The following is the embedded code to interface ADXL 335 via Arduino ADC:

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

const int xpin = A2;           // x-axis of the accelerometer
const int ypin = A1;           // y-axis
const int zpin = A0;           // z-axis (only on 3-axis models)
```

```
ros::NodeHandle nh;

//Creating an adc message
rosterial_arduino::Adc adc_msg;

ros::Publisher pub("adc", &adc_msg);

void setup()
{
    nh.initNode();

    nh.advertise(pub);
}

//We average the analog reading to eliminate some of the noise
int averageAnalog(int pin){
    int v=0;
    for(int i=0; i<4; i++) v+= analogRead(pin);
    return v/4;
}

void loop()
{
    //Inserting ADC values to ADC message
    adc_msg.adc0 = averageAnalog(xpin);
    adc_msg.adc1 = averageAnalog(ypin);
    adc_msg.adc2 = averageAnalog(zpin);

    pub.publish(&adc_msg);

    nh.spinOnce();

    delay(10);
}
```

The preceding code will publish the ADC values of X, Y, and Z axes in a topic called /adc. The code uses the `rosterial_arduino::Adc` message to handle the ADC value. We can plot the values using the `rqt_plot` tool.

The following is the command to plot the three axes values in a single plot:

```
$ rqt_plot adc/adc0 adc/adc1 adc/adc2
```

Next is a screenshot of the plot of the three channels of ADC:

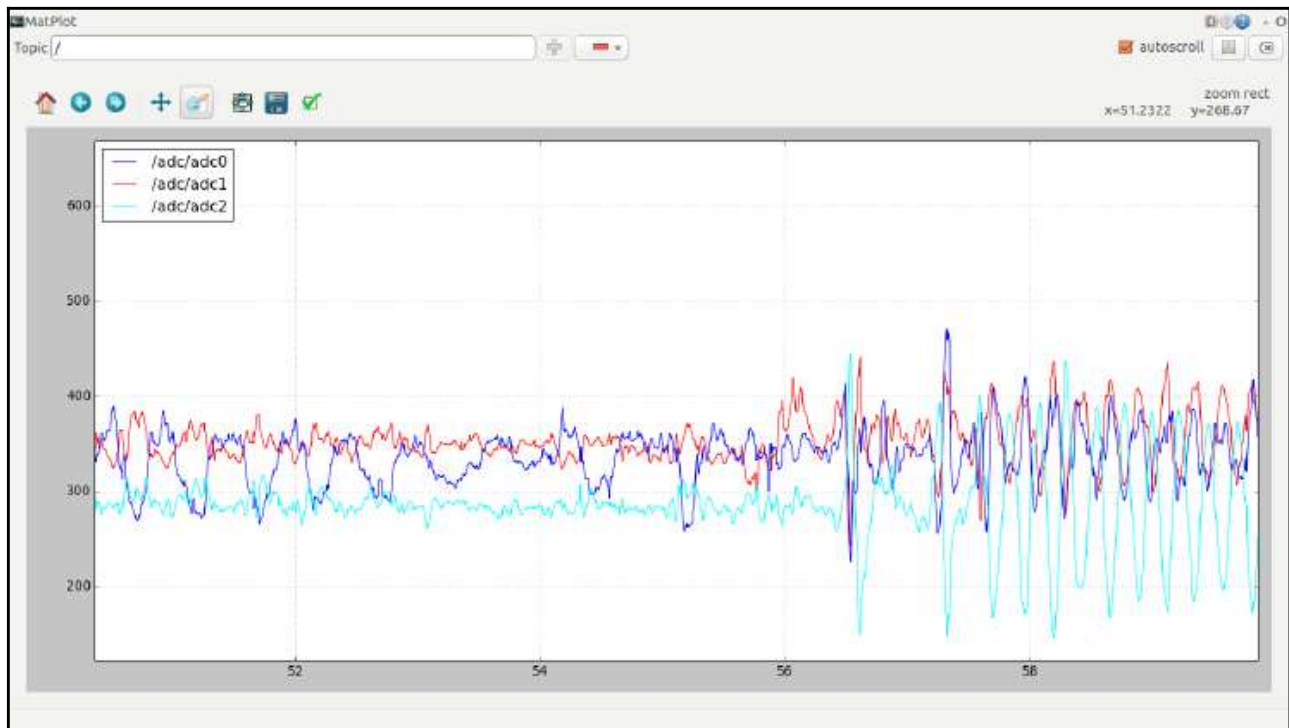


Figure 12: Plotting ADXL 335 values using rqt_plot

Arduino-ROS, example - ultrasonic distance sensor

One of the useful sensors in robots are the range sensors. One of the cheapest range sensors is the ultrasonic distance sensor. The ultrasonic sensor has two pins for handling input and output, called `Echo` and `Trigger`. We are using the HC-SR04 ultrasonic distance sensor, which is shown in the following image:

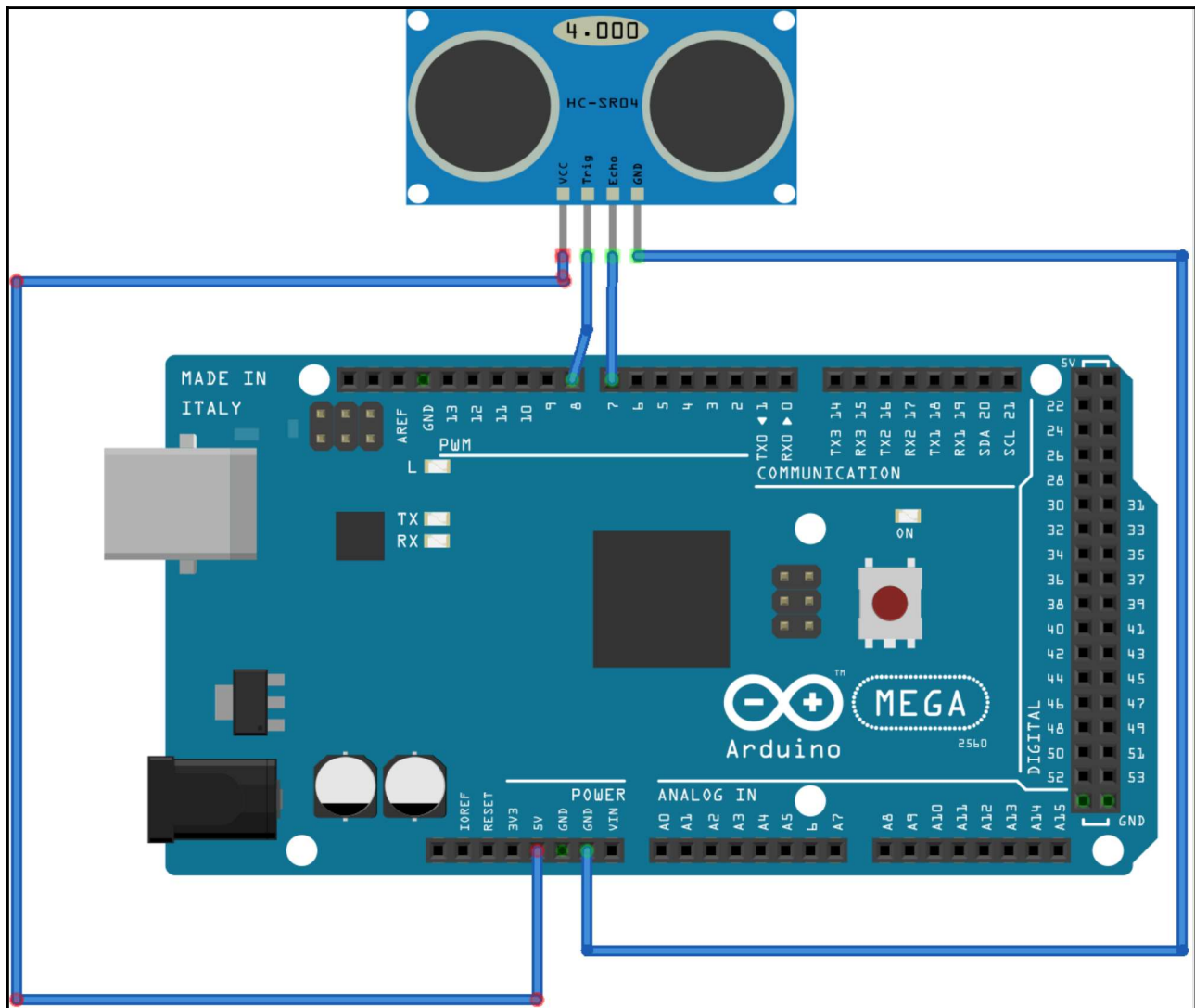


Figure 13: Plotting ADXL 335 values using `rqt_plot`

The ultrasonic sound sensor contains two sections: one is the transmitter and the other is the receiver. The ultrasonic distance sensor works like this: when a trigger pulse of a short duration is applied to the trigger pin of the ultrasonic sensors, the ultrasonic transmitter sends the sound signals to the robot environment. The sound signal sent from the transmitter hits on some obstacles and is reflected to the sensor. The reflected sound waves are collected by the ultrasonic receiver, generating an output signal which has a relation to the time required to receive the reflected sound signals.

Equations to find distance using the ultrasonic range sensor

The following are the equations used to compute the distance from an ultrasonic range sensor to an obstacle:

$$\text{Distance} = \text{Speed} * \text{Time}/2$$

$$\text{Speed of sound at sea level} = 343 \text{ m/s or } 34,300 \text{ cm/s}$$

$$\text{Thus, Distance} = 17,150 * \text{Time (unit cm)}$$

We can compute the distance to the obstacle using the pulse duration of the output. The following is the code to work with the ultrasonic sound sensor and send a value through the ultrasound topic using the range message definition in ROS:

```
#include <ros.h>
#include <ros/time.h>
#include <sensor_msgs/Range.h>

ros::NodeHandle nh;

#define echoPin 7 // Echo Pin
#define trigPin 8 // Trigger Pin

int maximumRange = 200; // Maximum range needed
int minimumRange = 0; // Minimum range needed
long duration, distance; // Duration used to calculate distance

sensor_msgs::Range range_msg;
ros::Publisher pub_range( "/ultrasound", &range_msg);

char frameid[] = "/ultrasound";
```

```
void setup() {
  nh.initNode();
  nh.advertise(pub_range);
  range_msg.radiation_type = sensor_msgs::Range::ULTRASOUND;
  range_msg.header.frame_id = frameid;
  range_msg.field_of_view = 0.1; // fake
  range_msg.min_range = 0.0;
  range_msg.max_range = 60;
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

float getRange_Ultrasound(){

  int val = 0;

  for(int i=0; i<4; i++) {
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);

    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);
    duration = pulseIn(echoPin, HIGH);
    //Calculate the distance (in cm) based on the speed of sound.
    val += duration;
  }
  return val / 232.8 ;
}
long range_time;

void loop() {
  /* The following trigPin/echoPin cycle is used to determine the
  distance of the nearest object by bouncing soundwaves off of it. */

  if ( millis() >= range_time ){
    int r =0;

    range_msg.range = getRange_Ultrasound();
    range_msg.header.stamp = nh.now();
    pub_range.publish(&range_msg);
    range_time = millis() + 50;
  }
  nh.spinOnce();
  delay(50);
}
```

We can plot the distance value, using the following commands:

- Start `roscore`:

```
$ roscore
```

- Start `serial_node.py`:

```
$ rosrun roserial_python serial_node.py /dev/ttyACM0
```

- Plot values using `rqt_plot`:

```
$ rqt_plot /ultrasound
```

As seen in the screenshot below, the center line indicates the current distance (`range`) from the sensor. The upper line is the `max_range` and line below is the `min_range`.

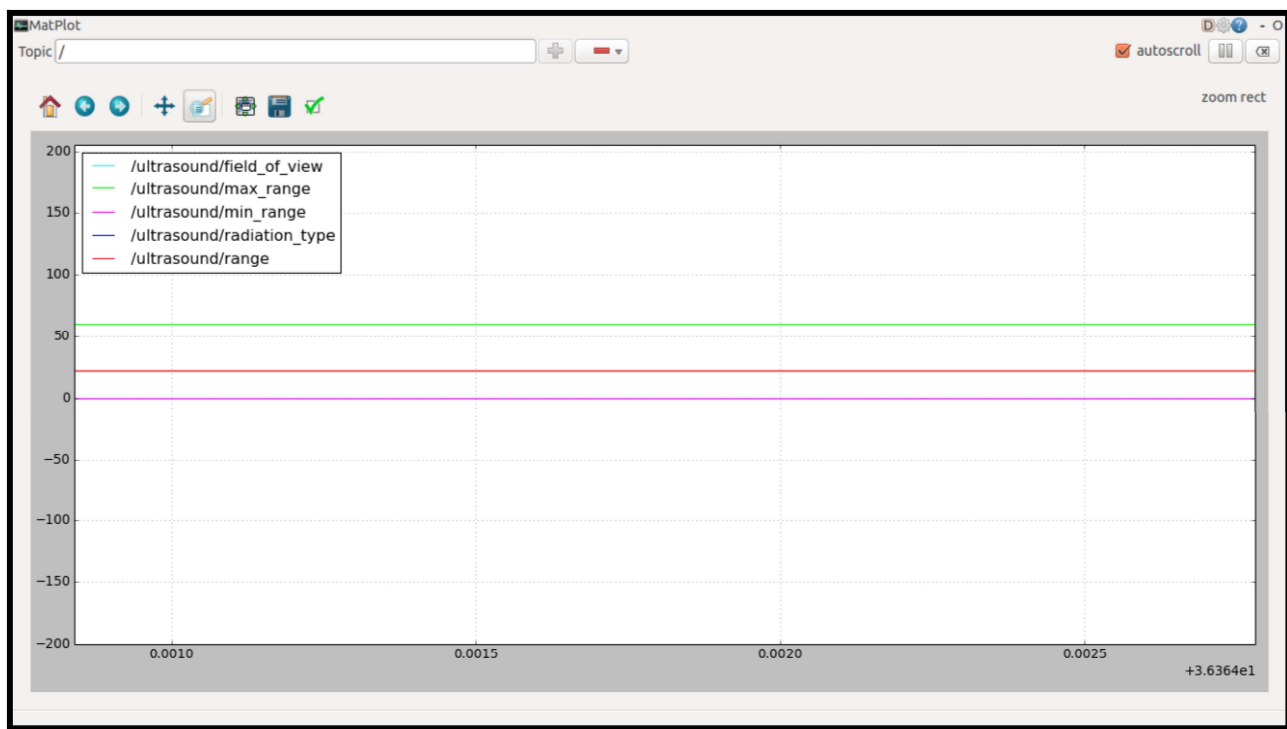


Figure 14: Plotting ultrasonic sound sensor distance value

Arduino-ROS example - Odometry Publisher

In this example, we will see how to send an `odom` message from an Arduino node to a PC.

This example can be used in a robot for computing `odom` and sending to the ROS

Navigation stack as the input. The motor encoders can be used for computing `odom` and can transmit to a PC. In this example, we will see how to send `odom` for a robot which is moving in a circle, without taking the motor encoder values:

```
/*
 * roserial Planar Odometry Example
 */

#include <ros.h>
#include <ros/time.h>
#include <tf/tf.h>
#include <tf/transform_broadcaster.h>

ros::NodeHandle nh;
//Transform broadcaster object
geometry_msgs::TransformStamped t;
tf::TransformBroadcaster broadcaster;

double x = 1.0;
double y = 0.0;
double theta = 1.57;

char base_link[] = "/base_link";
char odom[] = "/odom";

void setup()
{
  nh.initNode();
  broadcaster.init(nh);
}

void loop()
{
  // drive in a circle
  double dx = 0.2;
  double dtheta = 0.18;

  x += cos(theta)*dx*0.1;
  y += sin(theta)*dx*0.1;
  theta += dtheta*0.1;

  if(theta > 3.14)
```

```

    theta=-3.14;
    // tf odom->base_link
    t.header.frame_id = odom;
    t.child_frame_id = base_link;
    t.transform.translation.x = x;
    t.transform.translation.y = y;
    t.transform.rotation = tf::createQuaternionFromYaw(theta);
    t.header.stamp = nh.now();
    broadcaster.sendTransform(t);
    nh.spinOnce();
    delay(10);
}

```

After uploading the code, run `roscore` and `rosserial_node.py`. We can view `tf` and `odom` in RViz. Open RViz and view `tf`, as shown next. We will see the `odom` pointer moving in a circle on RViz, as follows:

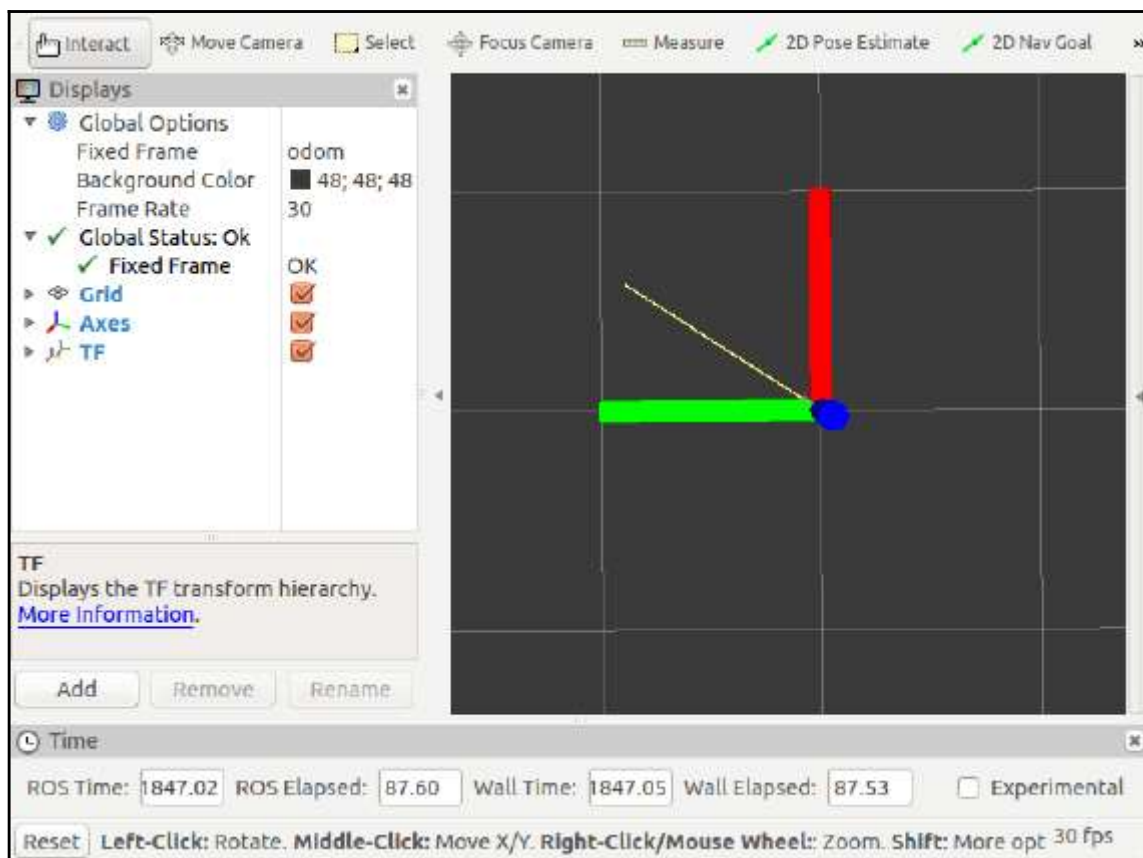


Figure 15: Visualizing odom data from Arduino

Interfacing non-Arduino boards to ROS

Arduino boards are commonly used boards in robots, but what happens if we want a board that is more powerful than Arduino? In such a case, we may want to write our own driver for the board, which can convert the serial messages into topics.

We will look at the interfacing of a non-Arduino board called *Tiva C Launchpad* to ROS, using a Python driver node, in Chapter 11: *Building and Interfacing Differential Drive Mobile Robot Hardware in ROS*. This chapter is about interfacing a real mobile robot to ROS, and the robot using the *Tiva C Launchpad* board for its operation.

Setting ROS on Odroid-XU4 and Raspberry Pi 2

Odroid-XU4 and Raspberry Pi2 are single board computers which have a low form factor the size of a credit card. These single board computers can be installed in robots and we can install ROS on them.

The main specifications comparison of Odroid-XU4 and Raspberry Pi2 is shown next:

Device	Odroid-XU4	Raspberry Pi 2
CPU	2.0 GHz Quad core ARM Cortex-A15 CPU from Samsung	900 MHz quad core ARM Cortex A7 CPU from Broadcom
GPU	Mali-T628 MP6 GPU	VideoCore IV
Memory	2 GB	1 GB
Storage	SD card slot or eMMC module	SD card slot
Connectivity	2 x USB 3.0, 1 x USB 2.0, micro HDMI, Gigabit Ethernet	4 x USB, HDMI, Ethernet, 3.5 mm audio jack
OS	Android, Ubuntu/Linux	Raspbian, Ubuntu/Linux, Windows 10
Connectors	GPIO, SPI, I2C, RTC (Real Time Clock) backup battery connector	Camera interface (CSI), GPIO, SPI, I2C, JTAG
Price	\$59	\$35

The following is an image of the Odroid-XU4 board:



Figure 16: Odroid-XU4 board

The Odroid board is manufactured by a company called **Hard Kernel**. The official web page of the Odroid-XU4 board is at http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825.

Odroid-XU4 is the most powerful board of the Odroid family. There are cheaper and lower performance boards as well, such as Odroid-C1+ and C2. All these boards support ROS. One of the popular single board computers is Raspberry Pi. The Raspberry Pi boards are manufactured by the Raspberry Pi Foundation, which is based in the UK (visit <https://www.raspberrypi.org>).

The following is an image of the Raspberry Pi 2 board:



Figure 17: The Raspberry Pi 2 board

We can install Ubuntu and Android on Odroid. There are also unofficial distributions of Linux, such as Debian mini, Kali Linux, Arch Linux, and Fedora, and support libraries, such as ROS, OpenCV, PCL, and so on. For getting ROS on Odroid, we can either install a fresh Ubuntu and install ROS manually like a standard desktop PC, or directly download the unofficial Ubuntu distribution for Odroid with ROS already installed.

The image for Ubuntu 16.04 for Odroid boards can be downloaded from http://de.eu.odroid.in/ubuntu_16.04lts/. You can download the desired kernel version for the Odroid-XU4 board (for example, ubuntu-16.04-mate-odroid-xu4-20170731.img.xz). This file contains pre-installed images of Ubuntu.

The list of the other operating systems supported on Odroid-XU4 is given on the wiki page at <http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4>, while the Raspberry Pi 2 official OS images are given at <https://www.raspberrypi.org/downloads/>.

The official OSes supported by the Raspberry Pi Foundation are Raspbian and Ubuntu. There are unofficial images based on these OSes which have ROS pre-installed on them. In this book, we are using the Raspbian Jessie images (<https://www.raspberrypi.org/downloads/>) with ROS installed, following the ROS wik page for the installation: <http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Kinetic%20on%20the%20Raspberry%20Pi>.

How to install an OS image to Odroid-XU4 and Raspberry Pi 2

We can download the Ubuntu image for Odroid and Raspbian Jessie image for Raspberry Pi 2 and can install to a micro SD card, preferably 16 GB. Format the micro SD card in the FAT32 filesystem, or we can use the SD card adapter or the USB-memory card reader for connecting to a PC.

We can either install the OS in Windows or in Linux. The procedure for installing the OS on these boards follows.

Installation in Windows

In Windows, there is a tool called **Win32diskimage** which is designed specifically for Odroid. You can download the tool from

http://dn.odroid.com/DiskImager_ODROID/Win32DiskImager-odroid-v1.3.zip.

Run **Win32 Disk Imager** with the Administrator privilege. Select the downloaded image, select the memory card drive, and write the image to the drive.

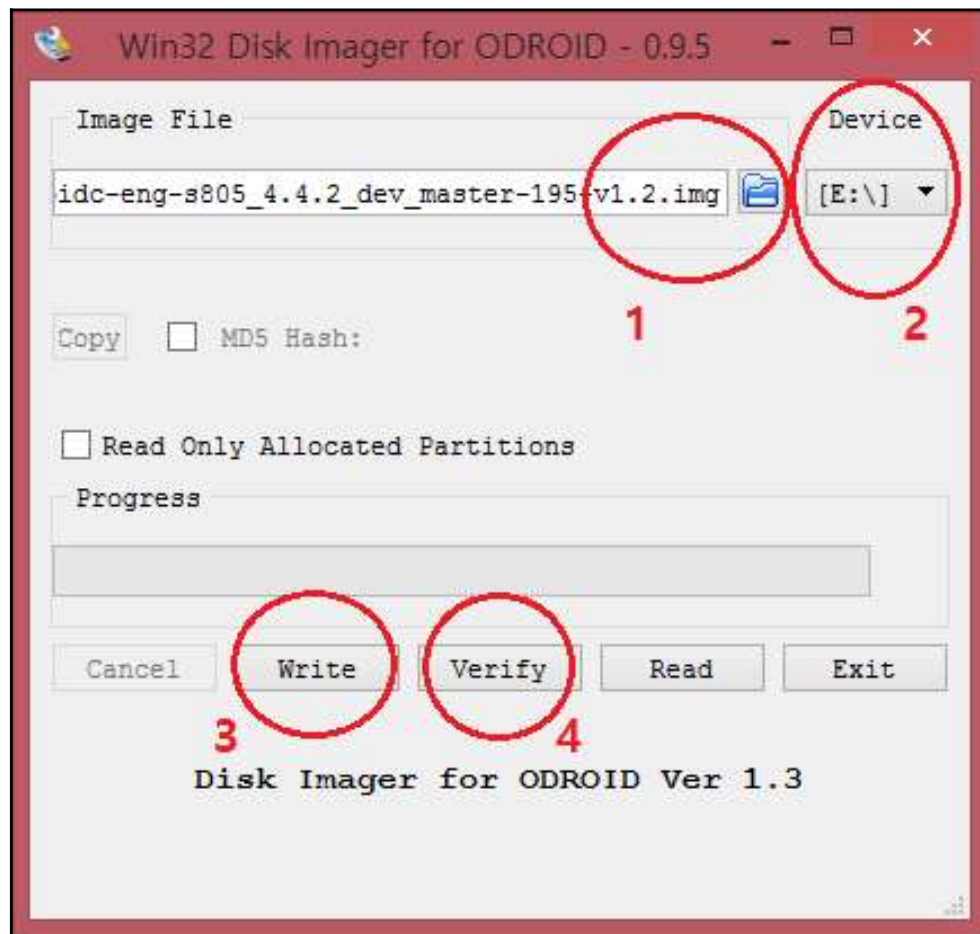


Figure 18: Win32 Disk Imager for Odroid

After completing this wizard, we can put the micro SD card in Odroid and boot up the OS with ROS support. The same tool can be used for Raspbian installation in Raspberry Pi 2. We can use the actual version of Win32 Disk Imager for writing Raspbian to a micro SD card from the following link:

<http://sourceforge.net/projects/win32diskimager/>.

Installation in Linux

In Linux, there is a tool called **disk dump (dd)**. This tool helps to copy the content of the image to the SD card. *dd* is a command line tool which is available in all the Ubuntu/Linux-based OSes. Insert the micro SD card, format to the FAT32 filesystem, and use the command mentioned later to write an image to the micro SD card.

In the *dd* tool, there is no progress bar to indicate the copy progress. To get the progress bar, we can install a pipe viewer tool called *pv*:

```
$ sudo apt-get install pv
```

After installing *pv*, we can use the following command to install the image file to the micro SD card. Note that you should have the OS image in the same path of the Terminal, and note the micro SD card device name; for example, *mmcblk0*, *sdb*, *sdd*, and so on. You will get the device name using the *dmesg* command:

```
$ dd bs=4M if=image_name.img | pv | sudo dd of=/dev/mmcblk0
```

image_name.img is the image name and the device name is */dev/mmcblk0*. *bs=4M* indicates the block size. If the block size is 4M, *dd* will read 4 megabytes from the image and write 4 megabytes to the device. After completing the operation, we can send it to Odroid and Raspberry Pi and boot the OS.

Connecting to Odroid-XU4 and Raspberry Pi 2 from a PC

We can work with Odroid-XU4 and Raspberry Pi 2 by connecting to the HDMI display port and connecting the keyboard and mouse to the USB like a normal PC. This is the simplest way of working with Odroid and Raspberry Pi.

In most of the projects, the boards will be placed on the robot, so we can't connect the display and the keyboards to it. There are several methods for connecting these boards to the PC. It will be good if we can connect the internet to these boards too. The following methods can connect the internet to these boards, and, at the same time, we can remotely connect via the SSH protocol:

- **Remote connection using Wi-Fi router and Wi-Fi dongle through SSH:** In this method, we need a Wi-Fi router with internet connectivity and a Wi-Fi dongle in the board for getting the Wi-Fi support. Both the PC and board will connect to the same network, so each will have an IP address and can communicate using that address.

- **Direct connection using an Ethernet hotspot:** We can share the internet connection and communicate using SSH via `Dnsmasq`, a free software DNS forwarder and DHCP server using low system resources. Using this tool, we can tether the Wi-Fi internet connection of the laptop to the Ethernet and we can connect the board to the Ethernet port of the PC. This kind of communication can be used for robots which are static in operation.

The first method is very easy to configure; it's like connecting two PCs on the same network. The second method is a direct connection of the board to the laptop through the Ethernet. This method can be used when the robot is not moving. In this method, the board and the laptop can communicate via SSH at the same time and can share Internet access too. We are using this method in this chapter for working with ROS.

Configuring an Ethernet hotspot for Odroid-XU4 and Raspberry Pi 2

The procedure for creating an Ethernet hotspot in Ubuntu and sharing Wi-Fi internet through this connection follows.

Go to **Edit Connections...** from the network settings and click on **Add** to add a new connection, as shown next:

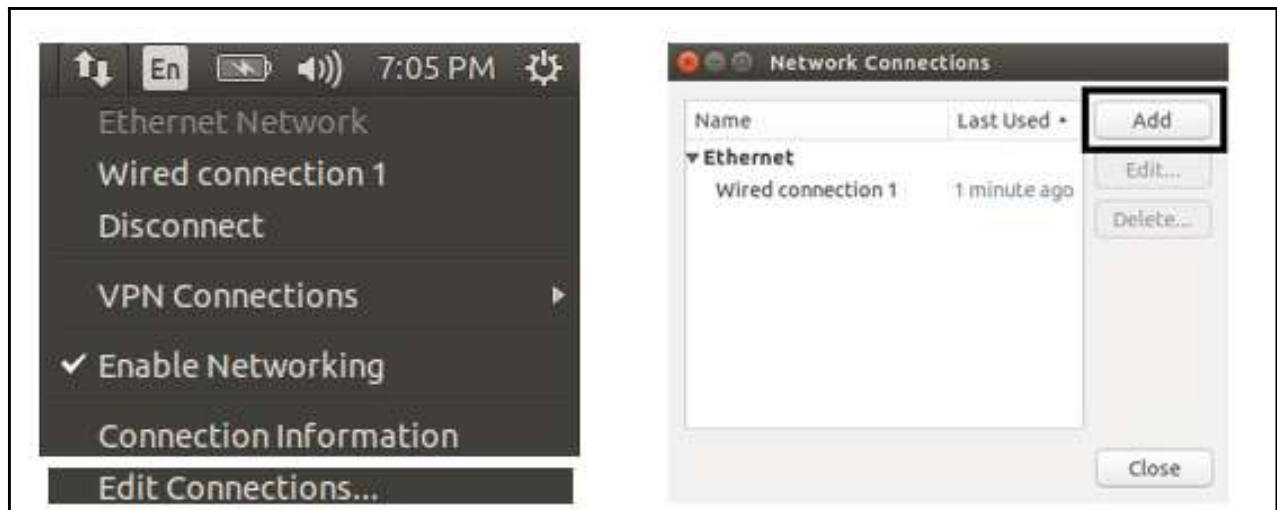


Figure 19: Configuring a network connection in Ubuntu

Create an **Ethernet** connection and in the **IPv4** setting, change the method to **Shared to Other computers**, and give the connection name as **Share**, as shown next:

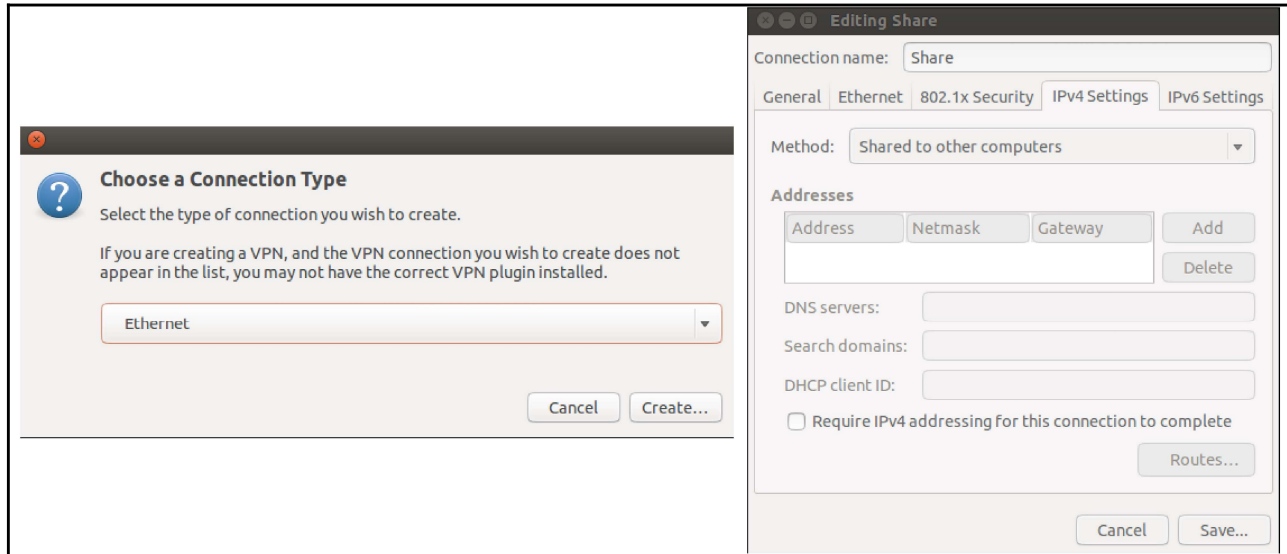


Figure 20: Creating a new connection for sharing through the Ethernet

Plug in the micro SD card, power up the Odroid or Raspberry Pi, and connect the Ethernet port from the board to the PC. When the board boots up, we will see that the shared network is automatically connected to the board.

We can communicate with the board using the following commands:

- In Odroid:

```
$ ssh odroid@ip_address
password is odroid
```

- In Raspberry Pi 2:

```
$ ssh pi@ip_address
password is raspberry
```

After doing SSH into the board, we can launch `roscore` and most of the ROS commands on the board like our PC. We will look at two examples using these boards. One is for blinking an LED, and the other is for handling a push button. The library we are using for handling the GPIO pins of Odroid and Raspberry is called **Wiring Pi**.

Odroid and Raspberry Pi have the same pin layout and most of the Raspberry Pi GPIO libraries are ported to Odroid, which will make the programming easier. One of the libraries we are using in this chapter for GPIO programming is Wiring Pi. Wiring Pi is based on C++ APIs, which can access the board GPIO using C++ APIs.

In the following sections, we will look at the instructions for installing Wiring Pi on Odroid and Raspberry 2.

Installing Wiring Pi on Odroid-XU4

The following procedure can be used to install Wiring Pi on Odroid-XU4. This is a customized version of Wiring Pi, which is used in Raspberry Pi 2:

```
$ git clone https://github.com/hardkernel/wiringPi.git
$ cd wiringPi
$ sudo ./build
```

Odroid-XU4 has 42 pins placed on two different connectors, CON10 and CON11 respectively, as show in the following image:

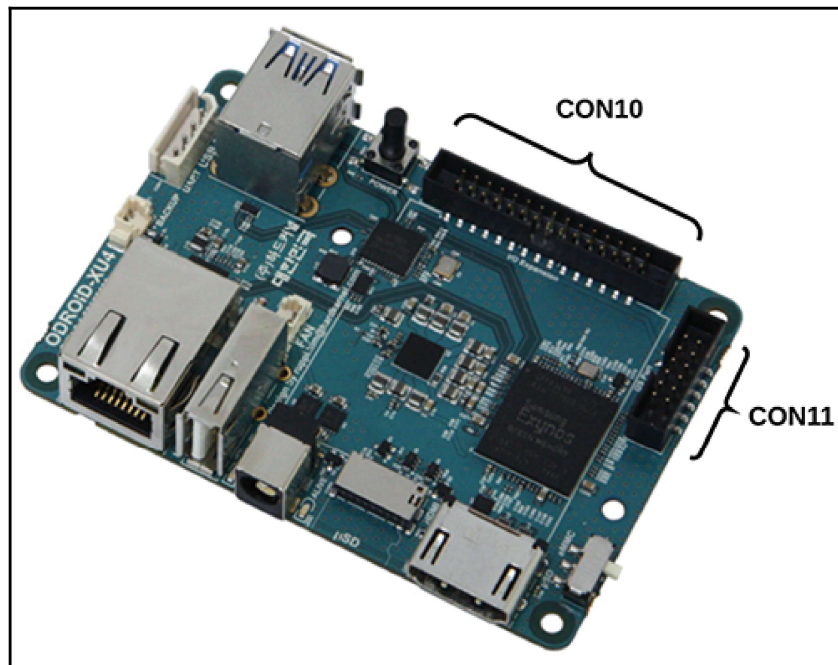


Figure 21: The CON10 and CON11 headers on Odroid-XU4

The Wiring Pi pin out of Odroid-XU4 (CON10) for bot connectors is given next:

ODROID XU4 Pin Layout (CON10)							
WiringPi GPIO#	Name(GPIO#)	Label	HEADER		Label	Name(GPIO#)	WiringPi GPIO#
		5V0	1	2	GND		
	ADC_0.AIN0	AIN0	3	4	#173	UART0_RTS	1
0	UART_CTS	#174	5	6	#171	UART0_RxD	16
12	MOSI_SPI1	#192	7	8	#172	UART0_TxD	15
13	MISO_SPI1	#191	9	10	#189	CLK_SPI1	14
10	CSN_SPI1	#190	11	12	PRWON		
2	GPIO	#21	13	14	#210	SCL.i2c	9
7	GPIO	#18	15	16	#209	SDA.i2c	8
3	GPIO	#22	17	18	#19	GPIO	4
22	GPIO	#30	19	20	#28	GPIO	21
26	GPIO	#29	21	22	#31	GPIO	23
	ADC_0.AIN3	AIN3	23	24	#25	GPIO	11
5	SCL_i2c	#23	25	26	#24	GPIO	6
27	SDA_i2c	#33	27	28	GND	GND	
		1V8	29	30	GND	GND	

Figure 22: Pin out of Odroid-XU4, CON10

The Wiring Pi pin out of Odroid-XU4 (CON11) for bot connectors is given next:

ODROID XU4 Pin Layout (CON11)							
WiringPi GPIO#	Name(GPIO#)	Label	HEADER		Label	Name(GPIO#)	WiringPi GPIO#
		5V0	1	2	GND		
		1V8	3	4	#173	SDA_i2c_5	30
	GPIO	#34	5	6	#171	SCL_i2c_5	31
	SCLK_i2s_0	#225	7	8	#172	GND	
	CDCLK_i2s_0	#226	9	10	#189	SDO_i2s_0	
	LRCK_i2s_0	#227	11	12	PRWON	SDI_i2s_0	

Figure 23: Pin out of Odroid-XU4, CON11

Installing Wiring Pi on Raspberry Pi 2

The following procedure can be used to install Wiring Pi on Raspberry Pi 2:

```
$ git clone git clone git://git.drogon.net/wiringPi
$ cd wiringPi
$ sudo ./build
```

The pin out of Raspberry Pi 2 and Wiring Pi is shown next:

P1: The Main GPIO connector							
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin
		3.3v	1	2	5v		
8	Rv1:0 - Rv2:2	SDA	3	4	5v		
9	Rv1:1 - Rv2:3	SCL	5	6	0v		
7	4	GPIO7	7	8	Tx0	14	15
		0v	9	10	RxD	15	16
0	17	GPIO0	11	12	GPIO1	18	1
2	Rv1:21 - Rv2:27	GPIO2	13	14	0v		
3	22	GPIO3	15	16	GPIO4	23	4
		3.3v	17	18	GPIO5	24	5
12	10	MOSI	19	20	0v		
13	9	MISO	21	22	GPIO6	25	6
14	11	SCLK	23	24	CE0	8	10
		0v	25	26	CE1	7	11
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin

P5: Secondary GPIO connector (Rev. 2 Pi only)							
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin
		5v	1	2	3.3v		
17	28	GPIO8	3	4	GPIO9	29	18
19	30	GPIO10	5	6	GPIO11	31	20
		0v	7	8	0v		
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin

Figure 24: Pin out of Raspberry Pi 2

The following are the ROS examples for Raspberry Pi 2.

Blinking LED using ROS on Raspberry Pi 2

This is a basic LED example which can blink the LED connected to the first pin of Wiring Pi, that is the 12th pin on the board. The LED cathode is connected to the GND pin and 12th pin as an anode. The following image shows the circuit of Raspberry Pi with an LED:

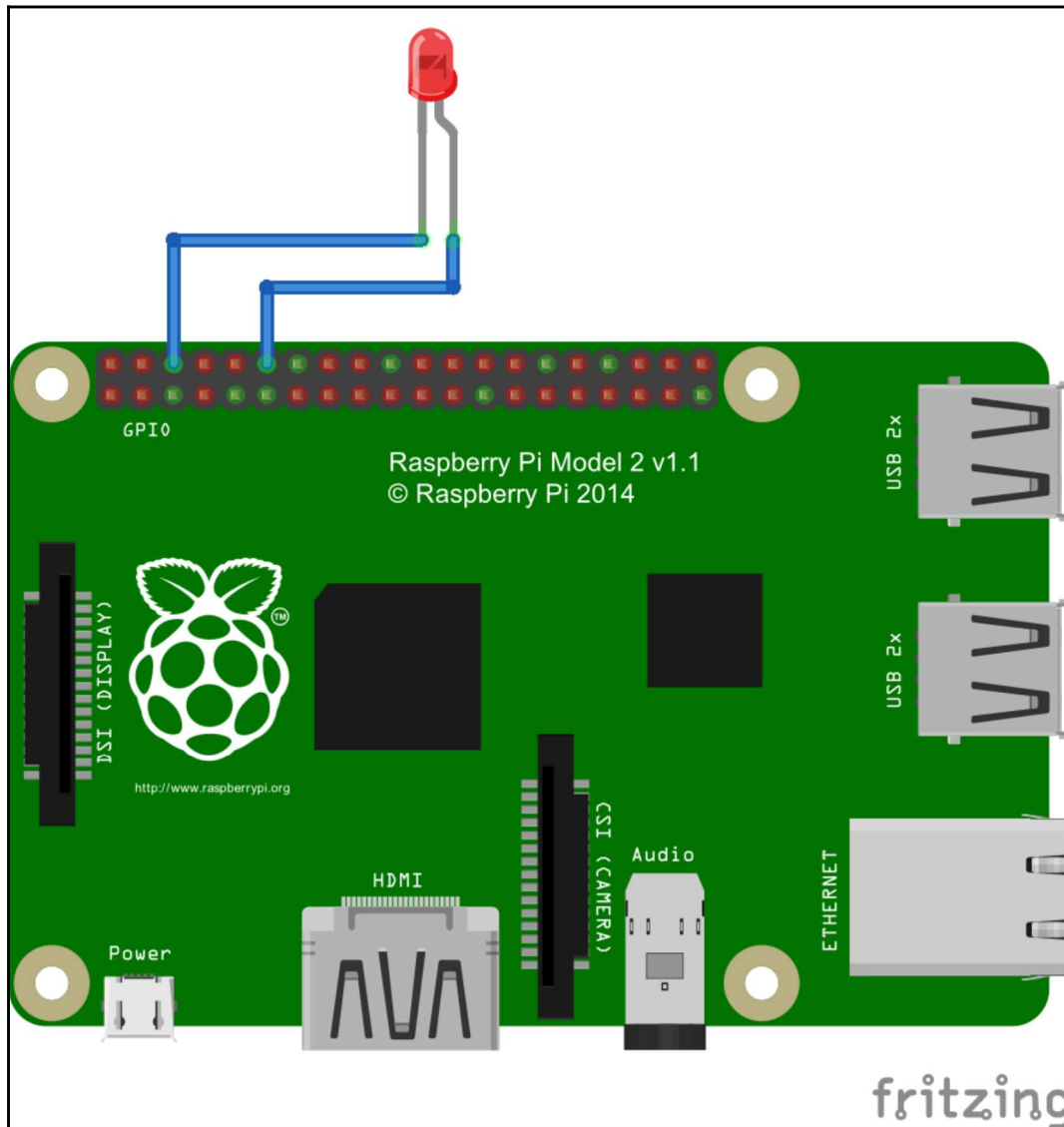


Figure 25: Blinking an LED using Raspberry Pi 2

We can create the example ROS package, using the following command:

```
$ catkin_create_pkg ros_wiring_example roscpp std_msgs
```

You will get the existing package from the `ros_wiring_examples` folder.

Create a `src` folder and create the following code called `blink.cpp` inside the `src` folder:

```
#include "ros/ros.h"
#include "std_msgs/Bool.h"
#include <iostream>

//Wiring Pi header
#include "wiringPi.h"

//Wiring PI first pin

#define LED 1

//Callback to blink the LED according to the topic value
void blink_callback(const std_msgs::Bool::ConstPtr& msg)
{

    if(msg->data == 1){
        digitalWrite (LED, HIGH) ;
        ROS_INFO("LED ON");
    }
    if(msg->data == 0){
        digitalWrite (LED, LOW) ;
        ROS_INFO("LED OFF");
    }
}

int main(int argc, char** argv)
{
    ros::init(argc, argv, "blink_led");
    ROS_INFO("Started Raspberry Blink Node");
    //Setting WiringPi
    wiringPiSetup (); //Setting LED pin as output
    pinMode(LED, OUTPUT);
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("led_blink",10,blink_callback);
    ros::spin();
}
```

This code will subscribe a topic called `led_blink`, which is a Boolean type. If we publish 1 to this topic, it will switch on the LED. If we publish 0, the LED will turn off.

Push button + blink LED using ROS on Raspberry Pi 2

The next example is handling input from a button. When we press the button, the code will publish to the `led_blink` topic and blink the LED. When the switch is off, the LED will also be OFF. The LED is connected to the 12th pin and GND, and the button is connected to the 11th pin and GND. The following image shows the circuit of this example. The circuit is also the same for Odroid:

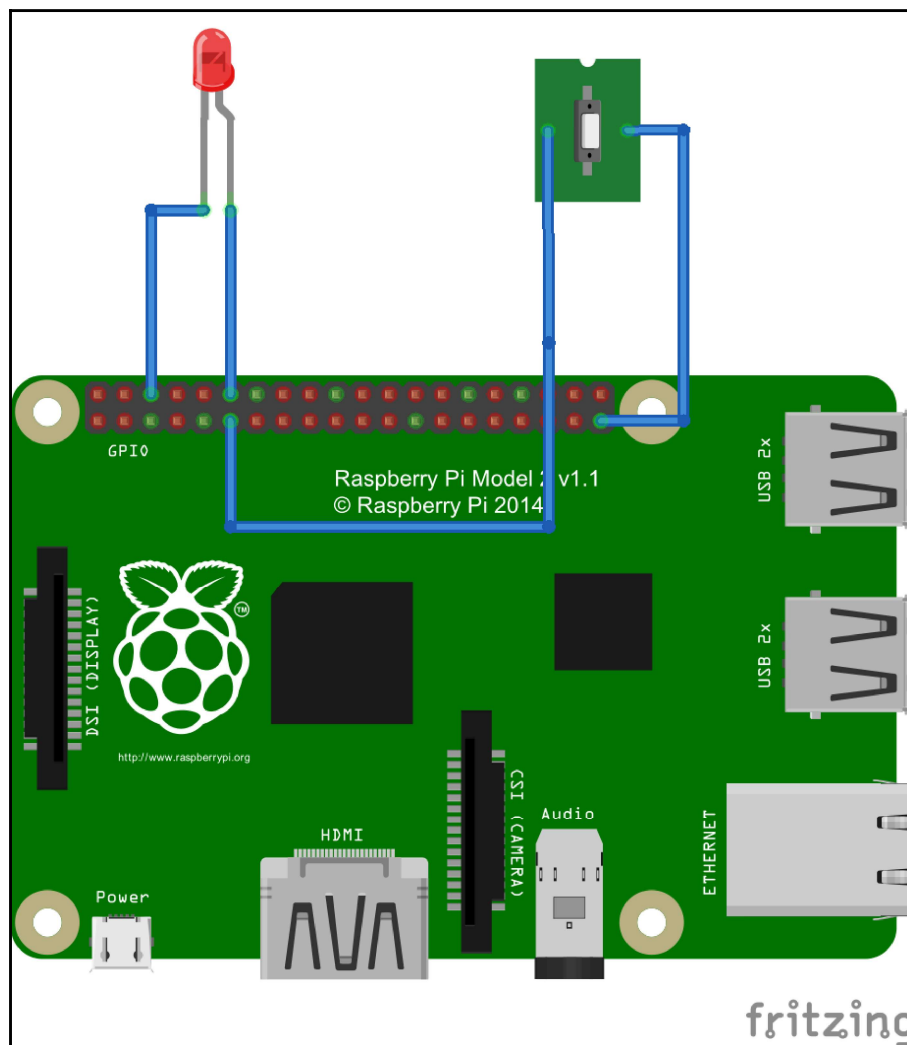


Figure 26: LED + button in Raspberry Pi 2

The code for interfacing the LED and button is given next. The code can be saved with the name `button.cpp` inside the `src` folder:

```
#include "ros/ros.h"
#include "std_msgs/Bool.h"

#include <iostream>
#include "wiringPi.h"

//Wiring PI 1
#define BUTTON 0
#define LED 1

void blink_callback(const std_msgs::Bool::ConstPtr& msg)
{
    if(msg->data == 1){

        digitalWrite (LED, HIGH) ;
        ROS_INFO("LED ON");
    }

    if(msg->data == 0){
        digitalWrite (LED, LOW) ;
        ROS_INFO("LED OFF");
    }

}

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

    ros::init(argc, argv, "button_led");
    ROS_INFO("Started Raspberry Button Blink Node");

    wiringPiSetup ();

    pinMode(LED, OUTPUT);
    pinMode(BUTTON, INPUT);
    pullUpDnControl(BUTTON, PUD_UP); // Enable pull-up resistor on button

    ros::NodeHandle n;
    ros::Rate loop_rate(10);

    ros::Subscriber sub = n.subscribe("led_blink",10,blink_callback);
```

```
    ros::Publisher chatter_pub = n.advertise<std_msgs::Bool>("led_blink",
10);

    std_msgs::Bool button_press;
    button_press.data = 1;

    std_msgs::Bool button_release;
    button_release.data = 0;

    while (ros::ok())
    {

        if (!digitalRead(BUTTON)) // Return True if button pressed
        {
            ROS_INFO("Button Pressed");
            chatter_pub.publish(button_press);
        }
        else
        {

            ROS_INFO("Button Released");
            chatter_pub.publish(button_release);
        }

        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

CMakeLists.txt for building these two examples is given next. The Wiring Pi code needs to link with the Wiring Pi library. We have added this in the CMakeLists.txt file:

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_wiring_examples)

find_package(catkin REQUIRED COMPONENTS
    roscpp
    std_msgs
)

find_package(Boost REQUIRED COMPONENTS system)

//Include directory of wiring Pi
```

```
set(wiringPi_include "/usr/local/include")

include_directories(
    ${catkin_INCLUDE_DIRS}
    ${wiringPi_include}
)

//Link directory of wiring Pi
LINK_DIRECTORIES("/usr/local/lib")

add_executable(blink_led src/blink.cpp)

add_executable(button_led src/button.cpp)

target_link_libraries(blink_led
    ${catkin_LIBRARIES} wiringPi
)

target_link_libraries(button_led
    ${catkin_LIBRARIES} wiringPi
)
```

Build the project using `catkin_make` and we can run each example. For executing the Wiring Pi based code, we need a root permission.

Running examples in Raspberry Pi 2

Now that we have built the project, before running the examples, we should do the following setup in Raspberry Pi. You can do this setup by logging in to Raspberry Pi through SSH.

We need to add the following lines to the `.bashrc` file of the root user. Take the `.bashrc` file of the root user:

```
$ sudo -i
$ nano .bashrc
```

Add the following lines to the end of this file:

```
source /opt/ros/indigo/setup.sh
source /home/pi/catkin_ws/devel/setup.bash
export ROS_MASTER_URI=http://localhost:11311
```

We can now log in with a different Terminal in our Raspberry Pi 2, and run the following commands to execute the `blink_demo` program:

Start `roscore` in one Terminal:

```
$ roscore
```

Run the executable as the root in another Terminal:

```
$ sudo -s
# cd /home/odroid/catkin_ws/build/ros_wiring_examples
# ./blink_led
```

After starting the `blink_led` node, publish 1 to the `led_blink` topic in another Terminal:

- For setting the LED to the ON state:

```
$ rostopic pub /led_blink std_msgs/Bool 1
```

- For setting the LED to the OFF state:

```
$ rostopic pub /led_blink std_msgs/Bool 0
```

- Run the button LED node in another Terminal:

```
$ sudo -s
# cd /home/odroid/catkin_ws/build/ros_wiring_examples
# ./button_led
```

Press the button and we can see the LED blinking. We can also check the button state by echoing the topic `led_blink`:

```
$ rostopic echo /led_blink
```

Interfacing DYNAMIXEL actuators to ROS

One of the latest smart actuators available on the market is DYNAMIXEL, which is manufactured by a company called Robotis. The DYNAMIXEL servos are available in various versions, some of which are shown in the following image:



Figure 27: Different types of DYNAMIXEL servos

These smart actuators have complete support in ROS, and clear documentation is also available for them.

The official ROS wiki page of DYNAMIXEL is
at http://wiki.ros.org/dynamixel_controllers/Tutorials.

Questions

- What are the different roserial packages?
- What is the main function of `roserial_arduino`?
- How does roserial protocol work?
- What are the main differences between Odroid and Raspberry Pi boards?

Summary

This chapter was about interfacing I/O boards to ROS and adding sensors on it. We have discussed the interfacing of the popular I/O board called Arduino to ROS, and interface's basic components, such as LEDs, buttons, accelerometers, ultrasonic sound sensors, and so on. After looking at the interfacing of Arduino, we discussed how to set up ROS on Raspberry Pi 2 and Odroid-XU4. We also presented a few basic examples in Odroid and Raspberry Pi based on ROS and Wiring Pi. Finally, we looked at the interfacing of smart actuators called DYNAMIXEL in ROS.