10-13-2010

On the Configuration of Sensors and Actuators on a Pioneer 3-AT Robot Controlled through the Robotic Operating System ROS

Jose-Marcio Luna

Rafael Fierro

On the Configuration of Sensors and Actuators on a Pioneer 3-AT Robot Controlled through the Robotic Operating System ROS

Jose-Marcio Luna [1]
Department of Electrical and Computer Engineering
The University of New Mexico
Albuquerque, NM 87131
e-mail: `jmarcio@ece.unm.edu`

Rafael Fierro
Department of Electrical and Computer Engineering
The University of New Mexico
Albuquerque, NM 87131
e-mail: `rfierro@eece.unm.edu`

UNM Technical Report: EECE-TR-10-0004

Report Date: October 13, 2010

[1]Supported by DOE URPR (University Research Program in Robotics) grant DE-FG52-04NA25590 and by NSF grants ECCS CAREER #0811347, IIS #0812338, CNS #0709329.

Abstract

We present a set of technical procedures to carry out experiments involving mobile robots. The hardware and software configuration to access the sensors and actuators in the robots usually turn out to be a relatively complex exercise. The intricacies of dealing with physical hardware connections and the lack of documentation related to software and hardware interfaces may significantly delay the experimental process. The Robotic Operating System (ROS), has become one of the main tools to interface a useful set of devices including several kinds of commercial robots through an IP network. Using Pioneer 3-AT robots we provide a step-by-step explanation of how to attach sensors such as GPS, IMUs, laser-range finders among others, to the robots as well as a couple of sample codes in C++ to operate the robots using ROS libraries.

Keywords

robotic operating system, ROS, pioneer robots, sensor configuration, GPS, IMU, laser range finder

1 Introduction

The current literature related to mobile robots reflects an increasing interest in the development of algorithms to control teams of robots. The robots should work cooperatively to carry out tasks that would be inefficient or even impossible to be carried out by a single robot. In reconfigurable sensor network applications each robot is equipped with a set of sensors able to measure some phenomenon or concentration in the environment. Based on the measurements, the sensors should position themselves or reconfigure the network such that a cost function or a calculated error is minimized. Examples of these kind of applications are search and rescue operations [1],[2], target detection [3],[4] and hazardous contamination [5] [6] among others.

Reconfigurable sensor networks have motivated the validation and verification of the developed theory through experiments rather than simulations. Several robotic tesbeds have been developed in the past such as the experimental testbed of the GRASP laboratory at the University of Pennsylvania [7], the team of SwarmBots presented in [8] and the COMET testbed [9], [10]. Given the importance of experimental applications for teams of mobile robots, we mean to provide a document whose main purpose is to deliver a comprehensive amount of technical information that allows the reader to easily address several issues involving the hardware and software configuration of a robotic testbed. This report specifically covers a testbed composed by a team of Pioneer 3-AT [11] robots controlled by the Robotic Operating System (ROS) [12]. It is worth mentioning that the hardware and software covered in this document are resources in the current state of the art.

The document is organized as follows, in section 2 we present a basic introduction to the main technical features of the Pioneer 3-AT (P3-AT) robots. In section 3 we provide a very brief description of ROS and specify the level of knowledge the reader should have to follow this report. We provide the web links to what we consider are the required readings and tutorials to be able to understand some basic ideas presented in the following sections. In section 4 we present the details of the hardware and software requirements in order to be able to communicate with sensors and actuators to be used with the P3-AT robots. In section 5 we present a couple of sample codes to use a GPS device and to control the speed of a P3-AT robot. Lastly, in section 6 we present the conclusions.

2 The Pioneer 3-AT robots

The Pioneer 3 - AT robots (P3-AT) [11] are programmable intelligent platforms equipped with the basic devices for navigation and sensing in the real world. They are part of an extense family of robots released in 1995 by the company Mobile Robots [13](nowadays part of the Adept team [14]) with the Pioneer 1 which continued with the Pioneer AT, Pioneer 2-DX, up to the most recent Pioneer 3-DX and 3-AT models.

The basic P3-AT robot is provided with high resolution motion encoders, reversible DC motors and motor controllers, as well as the four-wheel skid steer which carries out the balanced drive system of the robot. The power source consists of up to three 12 VDC lead-acid batteries and the control of all sensors and devices in the robot is carried out using a client/server application. The robots are equipped with a radio ethernet board which allows the wireless communication with the sensors and devices attached to the robots.

The P3-AT robot reaches speeds up to 0.7 m/s. Furthermore it can carry a payload of up to 40 kg and can climb a traversable slope of up to 40%. In Table I we show some mechanical features of the robot [11].

Some of the accesories for the P3-AT available in the MARHES lab are shown in Fig. 1. As you can notice some sensors such as the SICK laser and the Global Positioning System receiver (GPS) are aimed to carry out outdoors experiments whereas some others such as the hokuyo laser are oriented to indoors experiments.

A special aluminum plate and mounting system was created to interface the environmental sensor suite. The plate and mounting system allows for multiple sensor configurations as well as the ability to mount multiple accessories on each robotic platform. Fig. 2 shows the custom built aluminum plate with four precision light sensors and three magnetic sensors. Also shown is a Hokuyo UHG-08LX laser range finder. The addition of the

Table I: General specifications of the P3-AT robots.

Length	50.1 cm
Width	49.3 cm
Height	27.7 cm
Weight	14 kg
Payload	40 kg
Translate speed max	0.7 m/s
Rotate Speed max	140°/s

custom plate and mounting brackets allows for a quick swapping of sensors and accessories to address a variety of experimental tests.

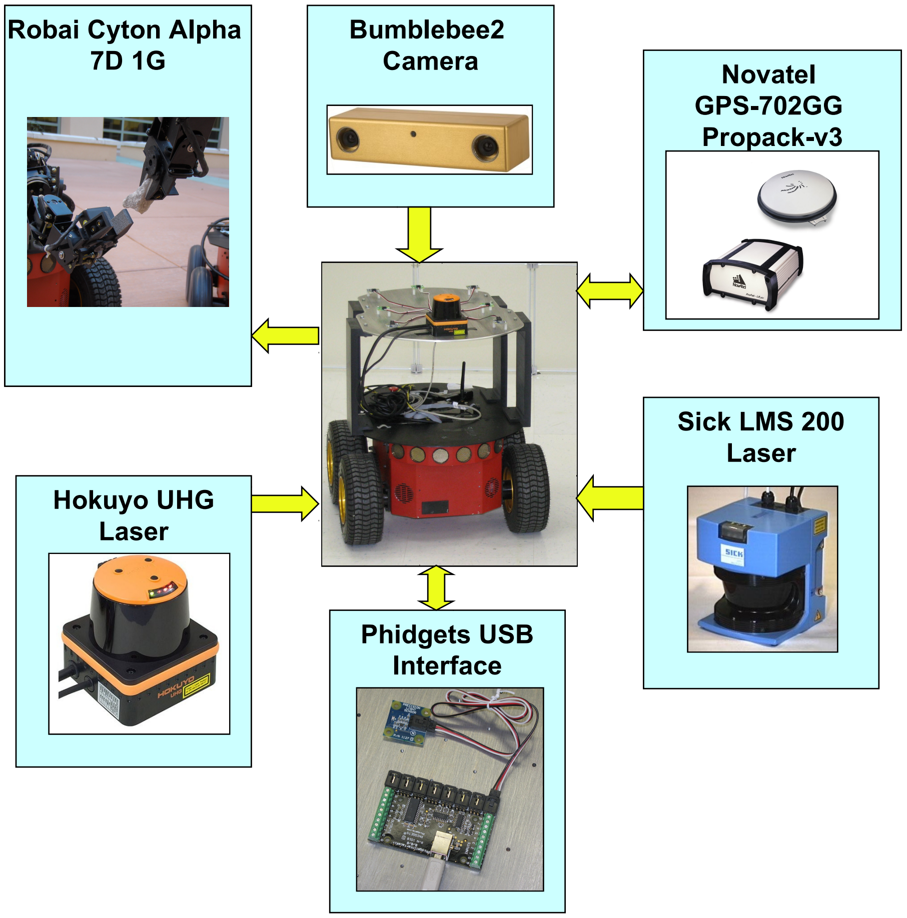


Figure 1: Accesories available for the P3-AT robots at the MARHES lab at the University of New Mexico [15].

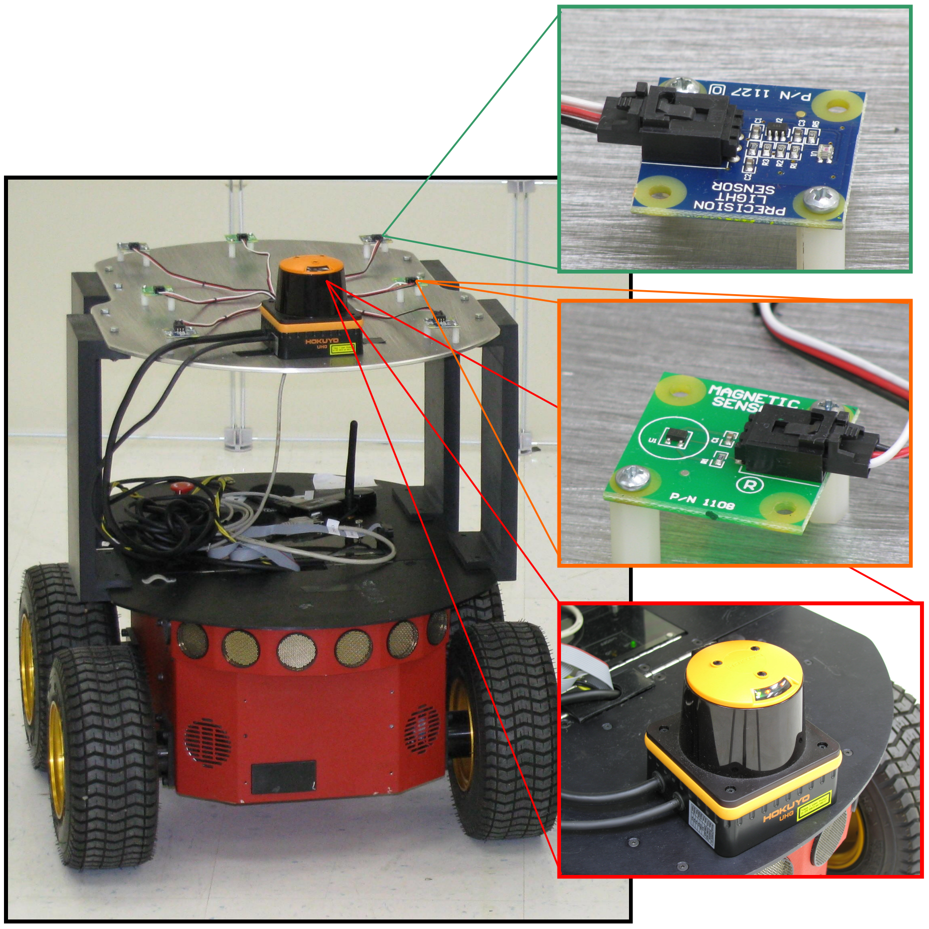


Figure 2: One of the five P3-AT robots available in the MARHES lab. Notice the detailed pictures of three different kinds of sensors, namely, Phidgets light precision sensors (top), Phidgets magnetic sensors (middle) and a Hokuyo UHG-08LX laser range finder (bottom), mounted on a custom fixture attached to the robot [10].

3 The Robotic Operating System

ROS is a meta-operating system [12] created by Willow Garage [16], which provides not only an interface to the sensors and actuators attached to a robot, but allows the implementation of commonly used functionalities such as message passing between processes and package management. The communication with the sensors and actuators is carried out through an IP network. There is a growing number of packages with drivers and applications being developed to interface several types of devices with ROS. The latest list of ROS packages is available at http://www.ros.org/browse/list.php.

ROS works under a client/server architecture, where one or more robots, with a set of attached sensors and actuators, upload a service to enable a communication channel between a remote client and the sensors. The client program can run on any computer that has a network connection to the robot or team of robots. At the best of our knowledge most of the ROS libraries are currently compatible with C, C++ and Python.

This technical report assumes that the reader has a basic knowledge of ROS, including

1. Commands for navigating the filesystem.

2. Creation and compilation of packages for ROS.

3. Creation and handling messages and services between applications.

4. Creation and configuration of nodes and topics.

5. Creation and configuration of publishers and subscribers.

6. Creation and configuration of services and clients.

Since this is a technical report aimed to ROS programmers at the intermediate level, we encourage the beginner reader to work on the tutorials for the beginner level available at http://www.ros.org/wiki/ROS/Tutorials before continuing with the reading of this document. Furthermore, we will limit the discussion to applications developed using C++. All the applications presented in this report have been implemented on the 32 bits versions of Linux Ubuntu 9.10 and 10.04.

3.1 ROS Installation

A very good documentation related to the installation and configuration of ROS is available at
http://www.ros.org/wiki/ROS/Installation
We strongly encourage the reader to carry out the installation of the *ros-[ros_version]-base*[1] package to follow the instructions given in this report without additional difficulties.

4 Installation and Configuration of Sensors with ROS

In this section we provide a detailed description of the configuration process of some of the sensors and actuators currently available for the P3-AT robots in the MARHES lab at the University of New Mexico. All the material presented in this section aims to be a quick start for setting up basic sensors and actuators commonly used in mobile sensor network applications using ROS. From now on, every subsection is dedicated to a different device. We highlight the main steps to follow in order to get all the devices properly working with ROS.

4.1 Hokuyo UHG Laser Range Finder

Working with the hokuyo laser range finder is rather straighforward using ROS. As explained in [17] and [18], *nodes* are executable files that provide the communication with different devices through connections called *topics*. We will see that for one or several sensors and actuators attached to the robots, we need at least a node running in the background to get or provide data from or to the sensors and actuators.

The Hokuyo UHG-08LX Laser [19] is an indoor range sensor which applies phase difference to get its distance measurements, minimizing the effect of the color and reflectance of the detected objects. The laser has a scan angle of 270° and a pitch of 0.36°. The maximum range of the sensor is 8 m with a divergence of 80 mm at that distance. The main features of the laser are illustrated in Table II. The laser device is shown in Fig. 3.

Hardware Connections

Depending on the Hokuyo laser model (UHG or URG) [20] we may need or not a power connector besides the data wire. In the case of the UHG laser, we need to identify the wires based on the sticker attached to the laser, indicated by the green rectangle shown Fig. 3. A Tamiya connector can be adapted to power the laser range finder

[1]Up to this point ROS has released two versions namely, *boxturle* and *cturtle* which is the latest one.

Table II: General specifications of the Hokuyo UHG-08LX laser

Power source	12 VDC ±10%
Current consumption	0.3 A
Detection Range	0.03 ∼ 11 m (Accuracy not guaranteed beyond 8 m)
Accuracy	0.1 ∼ 1 m : ±30mm 1 ∼ 8 m : 3% of the distance
Resolution	1 mm
Scan Angle	270°
Angular resolution	0.36°
Scan time	67 ms/scan
Ambient light resistance	Halogen/Mercury: 10000 lx or less Florescent: 6000 lx or less May cause measurement errors under strong light *e.g.* sunlight.
Weight	Approx 500 g
External dimensions	88 × 83 × 83 mm

as shown in Fig. 5(a). In the case of the URG model, the operation power is delivered through the USB port. Both laser models (UHG and URG) are USB compatible.

Once we make sure that the laser is properly connected to the power source and the USB port, we should verify that the laser has been properly identified by the system. Let us type the following command on a Linux terminal after connecting the laser,

```
$ dmesg | tail
```

The information about the USB devices recently connected to the embedded computer should appear on the terminal as follows,

```
[16282.420179] usb 5-1: new full speed USB device using uhci_hcd and address 3
[16282.751265] usb 5-1: configuration #1 chosen from 1 choice
[16282.756271] cdc_acm 5-1:1.0: ttyACM0: USB ACM device
```

if this information does not show up on the terminal, it is because the system is not recognizing the device. In that case, check the physical connections one more time and reconnect the device if necessary.

Software Configuration

The *hokuyo_node* package [21] comes already in the *ros-[ros_version]-base* installation of ROS, but it may require to be compiled. If that is the case, proceed to compile it by executing the following commands on the Linux terminal,

```
$ rosdep install hokuyo_node
$ rosmake hokuyo_node
```

Now, verify the USB port the laser is currently connected to, by using *dmesg — tail*, and specify the current USB port by executing the following command on the Linux terminal,

```
$ rosparam set hokuyo_node/port /dev/ttyACM1
```

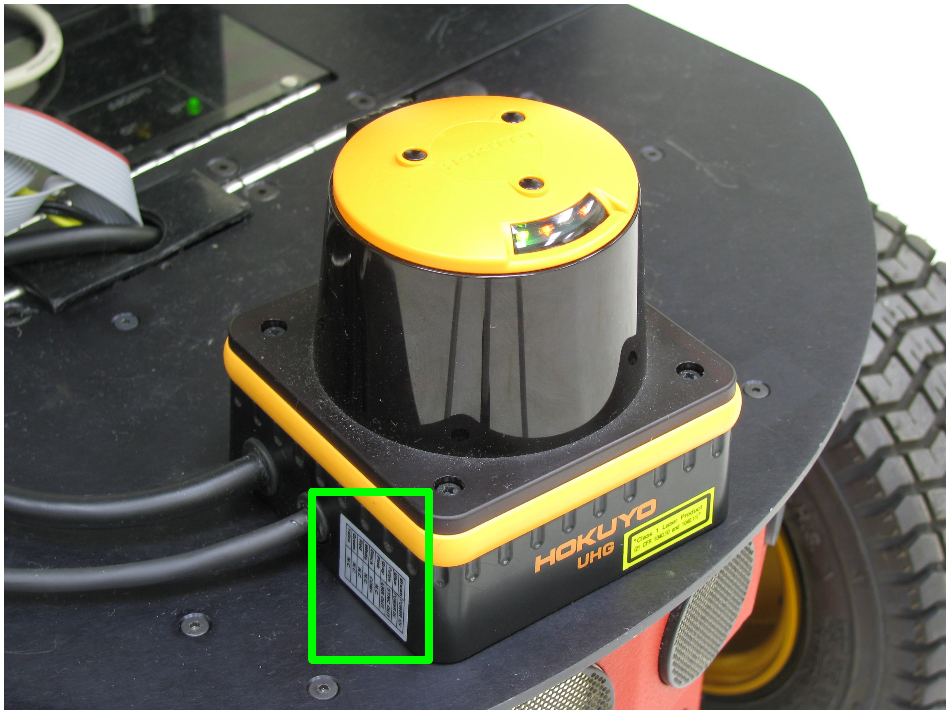


Figure 3: Hokuyo UHG-08LX laser range finder. The green frame encloses the sticker with the color of the wires and their functionality.

In this example the laser is plugged to the port */dev/ttyACM1*.

Now, we are ready to run the MASTER node,

```
$ roscore
```

and run the hokuyo_node in a new terminal. This node allows us to get data from the laser,

```
$ rosrun hokuyo_node hokuyo_node
```

Now, type the following command to visualize the data sent by the laser through the topic */scan* on the terminal,

```
$ rostopic echo /scan
```

then, we should be able to see the distance measurements from the laser on the screen.

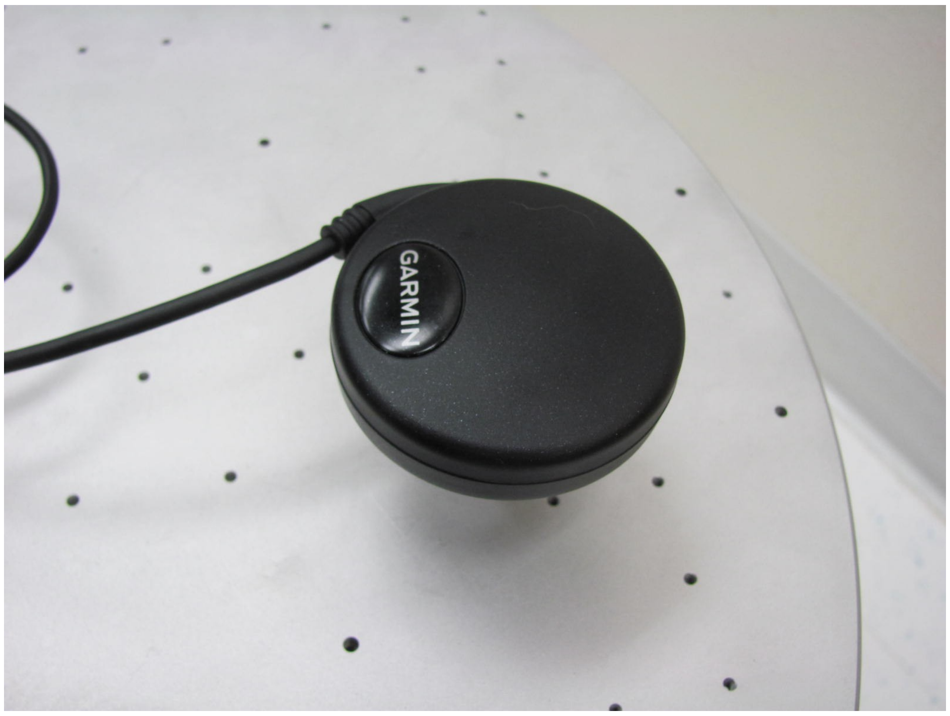


Figure 4: Garmin 18 5 hz GPS.

4.2 Garmin 18-18x 5 hz GPS

A Garmin Global Positioning system (GPS) is available to be mounted in our testbed for outdoor applications. We have the Garmin 18 and 18x models [22], [23] available which work at a frequency of 5 hz. This rather economical solution is part of the mainstream tools used for outdoors location of autonomous vehicles. The Garmin 18 is shown in Fig. 4 . The possibility of connecting up to 12 satellites at once, the compatibility with real-time WAAS corrections with position errors of less than 3 meters, and its lightness and compactness make these devices a suitable solution for robotic platforms. The main features of the Garmin 18-18x 5 hz GPS are illustrated in Table III.

Hardware Connection

The Garmin GPS 18-18x 5 hz comes with either USB or RS-232 compatibility. The USB version is powered by the USB port while the RS-232 comes with a set of wires to be connected to a power source of 5 V and another set to send and receive the data through a RS-232 connector. In order to adapt the laser to a P3-AT robot, we can adapt a Tamiya connector for the power source and a female DE-9 serial connector to get the GPS data as shown in Fig 5.

Table III: General specifications of the Garmin 18-18x 5 hz GPS

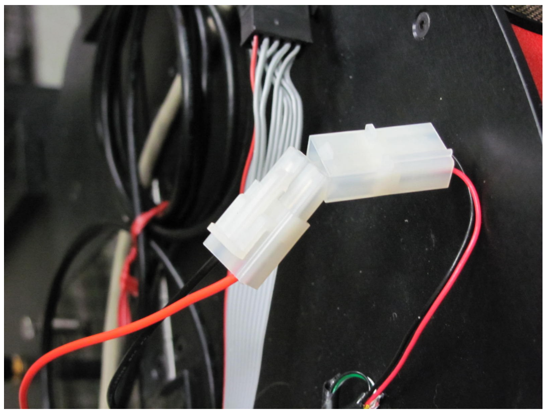
Weight	5.7 oz
Case Material	Polycarbonate thermoplastic
Cable length	5 m
Input Voltage	4.0–5.5 V
Input Current	65 mA @ 5.0 V
CMOS Serial Output Levels	0 – 5.0 V
Operating temperature	$-30°C + 90°C$
Update Rate	1 record per second (GPS 18–18x) 5 record per second (GPS 18–18x 5 hz)
Accuracy	Position: $<$ 15 m, 95 % typ (Standard Positioning Service) Position: $<$ 3 m, 95 % typ (WAAS) Velocity: 0.1 knot RMS steady state
Size	61 mm in diameter and 19.5 mm in height
Interface	GPS 18–18x: TIA-232-F (RS-232) async receiver. Def 4800 baud GPS 18–18x: 5hz TIA-232-F (RS-232) async receiver. Def 19200 baud

To check the actual data transmitted by the GPS you can use the command *gpscat* which is part of the applications included in the package *python-gps*. To install the *python-gps* package execute the following command in Linux Ubuntu,

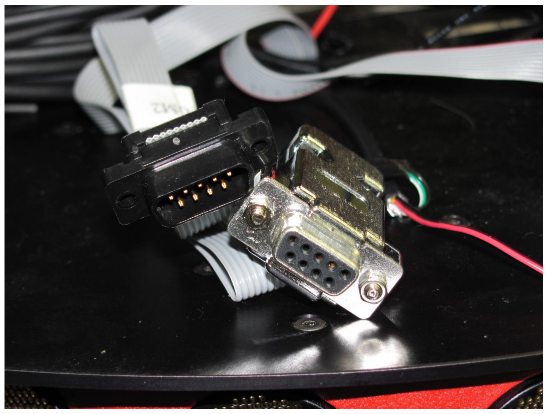
```
$ sudo apt-get install python-gps
```

Now, we are ready to check the NMEA messages delivered by the GPS by typing the following command on the terminal,

```
$ gpscat -s [baudrate] [serial_port]
```



(a) Tamiya connector



(b) DE-9 connector

Figure 5: The Tamiya connector and a DE-9 connector can be adapted in several devices to plug the power wires and data wires respectively.

The baudrate changes depending on the GPS model we use, and we suggest to try different values (e.g. 1200, 4800, 9600, 19200) if necessary. Based on Table III for a Garmin 18 (5 hz) GPS, the baudrate is 19200 baud/s, then to show the NMEA data on the terminal execute the following command assuming that the GPS is connected to the port */dev/ttyS1*.

```
$ gpscat -s 19200 /dev/ttyS1
```

The NMEA messages should appear on the terminal as follows,

```
$GPRMC,002454,A,3553.5295,N,13938.6570,E,0.0,43.1,180700,7.1,W,A*3F
$GPRMB,A,,,,,,,,,,,,A,A*0B
$GPGGA,002454,3553.5295,N,13938.6570,E,1,05,2.2,18.3,M,39.0,M,,*7F
$GPGSA,A,3,01,04,07,16,20,,,,,,,,3.6,2.2,2.7*35
$GPGSV,3,1,09,01,38,103,37,02,23,215,00,04,38,297,37,05,00,328,00*70
$GPGSV,3,2,09,07,77,299,47,11,07,087,00,16,74,041,47,20,38,044,43*73
$GPGSV,3,3,09,24,12,282,00*4D
$GPGLL,3553.5295,N,13938.6570,E,002454,A,A*4F
$GPBOD,,T,,M,,*47
$PGRME,8.6,M,9.6,M,12.9,M*15
$PGRMZ,51,f*30
$HCHDG,101.1,,,7.1,W*3C
$GPRTE,1,1,c,*37
$GPRMC,002456,A,3553.5295,N,13938.6570,E,0.0,43.1,180700,7.1,W,A*3D
```

and now, we are ready to install the ROS libraries to access the GPS data.

Software Configuration

Let us start by installing the *gpsd* package [24],

```
$ sudo apt-get install gpsd
```

Now, in order to install the *gpsd_client* for ROS you should install the revision control tool for linux *git* [25],

```
$ sudo apt-get install git-core
```

To install the stack containing the *gpsd_client* package we suggest the path given by /opt/ros/boxturtle/stacks, but this is left to the preferences of the user. Once we are located in your chosen folder type,

```
$ sudo git clone git://ram.umd.edu/umd-ros-pkg.git
```

and it will download all the stack to the designated folder. Once this is done move to the directory where the *gpsd_client* package is located,

```
$ roscd gpsd_client
```

If for some reason this does not work, move to the *gpsd_client* folder through the following path, /[your_chosen folder]/umd-ros-pkg/gps_umd/gpsd_client, and proceed to compile the package,

```
$ sudo rosmake --rosdep-install
```

Now, the *gpsd_client* node should be ready to be used. If you get some error messages while compiling the package, make sure that the dependencies were compiled before, especially the *gps_common* package.

As mentioned at the *gpsd_client* web site [26], the *gpsd* daemon should be running before the *gpsd_client* node is running. To run the *gpsd* daemon type,

```
$ gpsd -S [TCP_port] [serial_port]
```

The default TCP port is 2947 but in this particular example we will use the TCP port 4000 and the serial port /dev/ttyS1 as before, then,

```
$ gpsd -S 4000 /dev/ttyS1
```

We can check that the data from the GPS is now going through the TCP port 4000 by running the following command,

```
$ telnet localhost 4000
```

Now, press any key and you should start visualizing NMEA data on the screen. At this point we are ready to run the MASTER node,,

```
$ roscore
```

Then run the *gpsd_client* node,

```
$ rosrun gpsd_client gpsd_client _host:=localhost _port:=4000
```

Now, we are ready to test the node showing the actual data on the terminal by typing the following command,

```
$ rostopic echo /fix
```

and all the NMEA messages from the GPS should start appearing on the terminal right away.

4.3 Microstrain 3DM-GX2

The Microstrain 3DM-GX2 is described as an enhanced Inertial Measurements Unit (IMU) which combines a triaxial accelerometer, gyroscope, magnetometer and temperature sensors in order to compensate and correct typical sensor errors such as hysteresis and supply voltage variations [27]. Its small size and light weight make it suitable for authonomous vehicle implementations. The main features of the Microstrain 3DM-GX2 are shown in Table IV and the sensor is shown in Fig. 6.

Hardware Connection

The Microstrain 3DM-GX2 IMUs are USB devices whose power is obtained through the USB port. As before, type the following command on the terminal after the IMU has been connected.

```
$ dmesg | tail
```

Table IV: General specifications of the Garmin 18-18x 5 hz GPS

Orientation range	360° about all axes
Accelerometer range	$\pm$ 5 g (standard)
Accelerometer bias stability	± 0.005 g
Gyro range	$\pm 300^\circ/\mathrm{s}$ (standard)
Gyro bias stability	$\pm 0.2^\circ/\mathrm{s}$
Magnetometer range	± 1.2 Gauss
Magnetometer bias stability	0.01 Gauss
A/D resolution	16 bits
Orientation Accuracy	$\pm 0.5^\circ$ typ static test conditions $\pm 2.0^\circ$ typ dynamic test conditions
Orientation Resolution	$< 0.1^\circ$ minimum
Interface	USB 2.0
Digital output rate	1–250 hz
Serial data rate	$115,200$ bps
Supply voltage	5.2–9 V
Power Compsumption	90 mA
Operating Temperature	$-40\mathrm{to}70^\circ\mathrm{C}$
Dimensions	$41 \times 63 \times 32\mathrm{mm}$
Weight	39 g

It will show the information about the USB port which has been assigned to the IMU as shown below. If this information does not show up check the physical connections one more time and reconnect the device if necessary.

```
[14008.314512] usb 2-1: new full speed USB device using uhci_hcd and address 2
[14008.478574] usb 2-1: configuration #1 chosen from 1 choice
[14008.484438] cp210x 2-1:1.0: cp210x converter detected
[14008.594470] usb 2-1: reset full speed USB device using uhci_hcd and address 2
[14008.742441] usb 2-1: cp210x converter now attached to ttyUSB0
```

Software Configuration

Usually when the *ros-[ros_version]-base* package is installed, it already includes the *imu_drivers* stack which includes the package *microstrain_3dmgx2_imu*. You can check if the package is installed by executing the command,

```
$ rospack find microstrain_3dmgx2_imu
```

It should show the path of the package in the system. If it does not return any output go then to the folder of your preference and make sure you have administrative permissions to install the package; then execute the following *subversion* [28] command on the terminal,

```
$ svn co `roslocate svn microstrain_3dmgx2_imu`
```

This will download the whole *microstrain_3dmgx2_imu* package to the actual folder. Then install the dependencies and compile the package by executing the following commands on the terminal,

Figure 6: The Microstrain 3DM-GX2 IMU.

```
$ rosdep install microstrain_3dmgx2_imu
$ rosmake microstrain_3dmgx2_imu
```

After you make sure that the *microstrain_3dmgx2_imu* package is installed on your system, please connect the IMU to the USB port and check the name and path of the current port the IMU is using by using the *dmesg — tail* command. Then run the MASTER node,

```
$ roscore
```

Now, run the *imu_node* node to get the data from the IMU,

```
$ rosrun microstrain_3dmgx2_imu imu_node
```

This node assumes that the IMU is plugged to the port /dev/ttyUSB0 by default. In case you have the IMU connected to a different port *e.g.* /dev/ttyUSB1 then we should specify it by typing,

```
$ rosrun microstrain_3dmgx2_imu imu_node _port:=/dev/ttyUSB1
```

Some messages will start showing up on the terminal indicating that the IMU is calibrating. Then, there should be an indication that the IMU is working. At this point we are ready to show the IMU messages on the terminal by checking the information of the */imu/data* topic,

```
$ rostopic echo /imu/data
```

and we should be able to see the IMU data as shown below,

```
header:
  seq: 1951
  stamp: 1281648062918698527
  frame_id: imu
orientation:
  x: 0.745911226365
  y: -0.659430505519
  z: -0.0608342600388
  w: 0.0711829027672
orientation_covariance: [0.001, 0.0, 0.0, 0.0, 0.001, 0.0, 0.0, 0.0, 0.001]
angular_velocity:
  x: -0.00512957340106
  y: -0.000760086695664
  z: 0.000855371588841
angular_velocity_covariance: [0.000, 0.0, 0.0, 0.0, 0.000, 0.0, 0.0, 0.0, 0.000]
linear_acceleration:
  x: 0.0427974506538
  y: 1.85255225912
  z: -9.62984849498
linear_acceleration_covariance: [0.009, 0.0, 0.0, 0.0, 0.009, 0.0, 0.0, 0.0, 0.009]
```

Notice that the IMU data is given in quaternions by the ROS node. The programmer should carry out the respective conversion to degrees or radians if required.

4.4 Phidgets Environmental Sensor Suite

The environmental sensor suite consists of a Phidgets USB interface I/O board [29] capable of measuring eight digital and analog inputs, and capable of driving eight digital outputs. The Phidgets I/O board can accommodate pressure, temperature, humidity, light intensity, and magnetic field sensors among others. Furthermore, it is equipped with a digital input hardware filter to eliminate false triggering at the digital inputs. A detailed view of a Phidgets USB interface with an attached light precision sensor is shown in Fig 7.

Light Precision Sensors

The 1127 precision light sensor [30] is a 5 VDC device able to measure from 1 lx to 1000 lx with a typical accuracy of 95%. The formula to convert the sensor data to luminosity units is

$$\text{Luminosity (lx)} = \text{SensorValue}$$

The main features of the device are given in Table V, and a picture of the sensor is shown in Fig. 7.

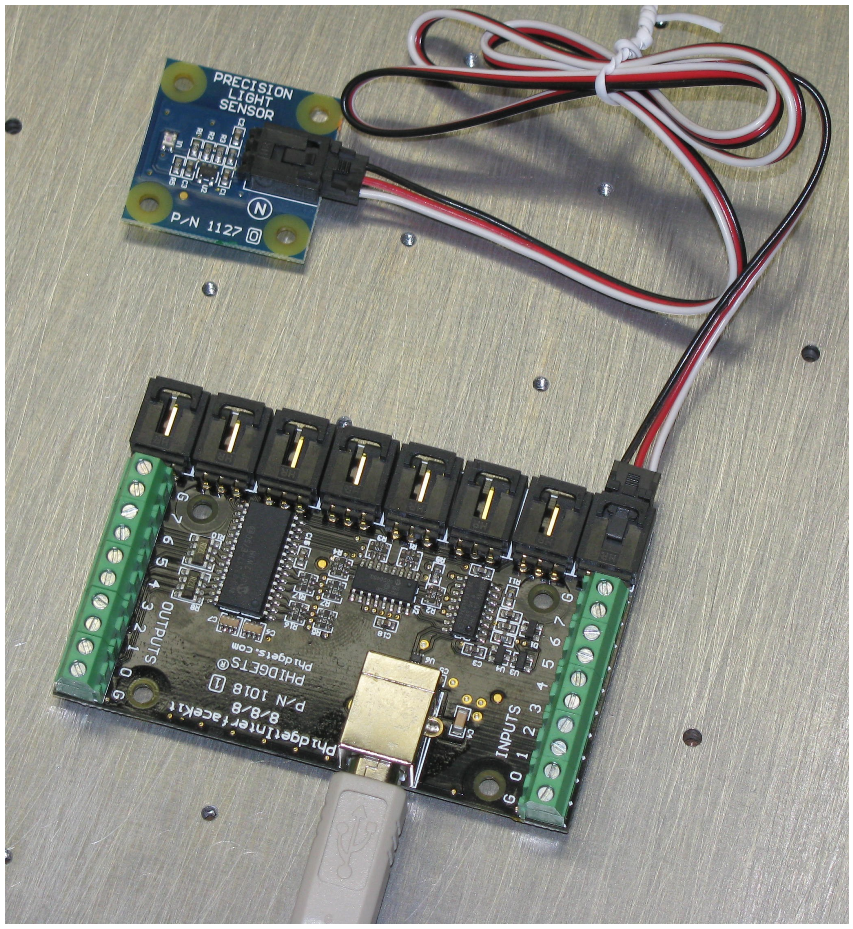


Figure 7: A Phidgets USB interface I/O board with an attached precision light sensor. Notice the USB port at the bottom of the image and the digital input outputs at the right and left edges respectively. The analog inputs are located at the top edge of the board where the light sensor is connected.

Magnetic Sensors

The 1108 magnetic sensor [31] is a temperature stable sensor with a sensivity of 1 Gauss/SensorValue. The formula to convert the sensor value to Gauss units is

$$\Phi\ (\text{Gauss}) = 500 - \text{SensorValue}.$$

Table V: General specifications of the Phidgets 1127 light precision sensor

Current consumption	2 mA
Bandwidth	50 Hz
Minimum Voltage	3.3 VDC
Maximum Voltage	5 VDC
Minimum light level	1 lx
Maximum light level	1000 lx
Error	5%

Table VI: General specifications of the Phidgets 1108 magnetic sensor

Current consumption	2 mA
Voltage output range	0.2 - 4.7 VDC
Device supply operating range	4.5 - 5.5 VDC
Typical error (@25°C)	±0.5%
Sensivity	1 G/SensorValue

We illustrate the main features of this sensor in Table VI. The sensor is shown in Fig. 2.

4.5 Hardware Connection

As mentioned before, the Phidgets interface kit comes with a USB type A port that powers the device as well. In order to get data from the device it is necessary to install the Phidgets drivers available at http://www.phidgets.com/drivers.php. In this report we limit the description of the installation of the Phidgets drivers in Linux Ubuntu only.

4.6 Software Configuration

First, download the Phidgets drivers for Linux available at http://www.phidgets.com/downloads/libraries/libphidget_2.1.7.20100621.tar.gz.

Save the compressed file *libphidget_2.1.7.20100621.tar.gz* in the folder of your preference. Please be sure you have administratives rights over the folder and go to the chosen folder,

```
$ cd /[path to your chosen folder]
```

Now, uncompress the file *libphidget_2.1.7.20100621.tar.gz* by typing,

```
$ tar -xzvf libphidget_2.1.7.20100621.tar.gz
```

and go to the uncompressed folder,

```
$ cd libphidget-2.1.7.20100621/
```

Then configure the compilation of the drivers by typing,

```
$ ./configure --enable-shared
```

where the parameter *enable-shared* is just a precaution to avoid future errors related to shared libraries. Afterwards, compile and install the drivers,

```
$ make
$ sudo make install
```

Once we are done with the installation of the drivers we proceed to install the necessary ROS packages to interface the Phidgets interface kit. If you installed the *ros-[ros_version]-base* package, as suggested at the beginning, you may only need the additional packages *phidgets_py_api* and *phidgets_ros*. To install them, just go to the folder of your preference and be sure you have administrative rights over the folder. Then go to the chosen folder and type on the terminal,

```
$ svn co 'roslocate svn phidgets_py_api'
```

It is worth mentioning that the package *phidgets_py_api* should be installed at first since it is a dependency of the *phidgets_ros package*. Then move inside the *phidgets_py_api* folder and compile it,

```
$ sudo rosmake --rosdep-install
```

Now, go to the folder of your preference and proceed to download the *phidgets_ros* package by typing,

```
$ svn co 'roslocate svn phidgets_ros'
```

In a similar way, go inside the folder *phidgets_ros* and type,

```
$ sudo rosmake --rosdep-install
```

Now, you should be ready to use your phidgets sensors. Please connect your 1018-PhidgetsInterfaceKit to any USB available on your computer and execute the following command on the terminal,

```
 $ dmesg | tail
```

In the last two lines of the output in the terminal, we should read someting similar to,

```
[17596.120134] usb 5-1: new low speed USB device using uhci_hcd and address 4
[17596.305435] usb 5-1: configuration #1 chosen from 1 choice
```

which indicates that the device has been detected. Then run the MASTER node,

```
$ roscore
```

and the interface_kit.py node,

```
$ rosrun phidgets_ros interface_kit.py
```

We should be able to check the data given by any sensor attached to the interface board. As an example let us suppose that we have a magnetic sensor connected to the analog port whose index is 0, then the command,

```
$ rostopic echo /interface_kit/95135/sensor/0
```

will show the data delivered by that sensor. Notice that the digit at the end of the stream indicates the index of the analog port, then it can be any integer from 0 to 7.

4.7 Odometry and Sonar Ring

In the MARHES lab the available P3-AT robots came with wheel encoders and a front sonar ring already set up. However there are some additional sensors that can be attached to the P3-AT robots such as, grippers and pant-tilt units that can be controlled or accessed using ROS. Unfortunately, since we have not implemented anything using those resources in the MARHES lab we limit the discussion to the odometry and sonar ring in this section.

Table VII: General specifications of the motion encoders of the P3-AT robots

Counts/rev	34,000
Counts/mm	49
Counts/rotation	22,500

Sonar Front Array

This low cost sensor is a classroom oriented tool to give the students experience with obstacle detection sensors [11]. As shown in Fig. 2, the circular sensors at the front of the robot forms an array of sonars, with 2 sonars on each side and 6 more forward covering interval angles of 20°.

Motion Enconders

The P3-AT comes with 4 motion encoders [11] ready to use with the features shown in Table VII.

4.8 Hardware Connection

As we mentioned before, all these accessories already came with the P3-AT robots so we do not cover their physical installation in this document.

4.9 Software Configuration

In order to access the odometry and the sonar ring we need the *p2os_driver* package [32]. To download the package, please locate a folder of your preference where you have administrative rights. Then go to the chosen folder and type,

```
$ svn co `roslocate svn p2os_driver`
```

then the latest version of the driver will be downloaded to the chosen folder. Then, navigate to the folder *src* inside the *p2os_driver* folder that you just downloaded,

```
$ cd /[your chosen path]/p2os_driver/src
```

and using your favorite text editor go to the line 37 which contains the text,

```
n_private.param("use_sonar", use_sonar_, false);
```

and replace it by,

```
n_private.param("use_sonar", use_sonar_, true);
```

this will allow ROS to access the data given by the sonar ring.

Now, we should be ready to test the delivered data from the sonars and from the odometry by checking the topics called */pose* and */sonar* by running the MASTER node,

```
$ roscore
```

then running the p2os node,

```
$ rosrun p2os_driver p2os
```

Now, we can check the topic */pose*

```
$ rostopic echo /pose
```

and the following information should start appearing on the terminal,

```
header:
  seq: 2844
  stamp: 1281647684650540000
  frame_id: odom
child_frame_id: base_link
pose:
  pose:
    position:
      x: 0.051
      y: -0.001
      z: 0.0
    orientation:
      x: -0.0
      y: 0.0
      z: 0.0174524064373
      w: -0.999847695156
  covariance: [0.0, 0.0, ... ,0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
twist:
  twist:
    linear:
      x: 0.0
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 0.0
  covariance: [0.0, 0.0, ... ,0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Do not be alarmed by the beeping of the robot. This node not only allows the reception of the data from the sensors but it allows us to control the angular and linear speed of the robot as well as to enable or disable the motors of the robots. Then, the beep is an indicator that the robot is not receiving any angular nor linear speed values at the node inputs.

5 Sample Codes for ROS in C++

Up to this point, we have been able to communicate with several devices using ROS commands. In this section we illustrate the utilization of the ROS libraries in C++ to receive data from a Garmin 18-18x 5 hz GPS and to send speed commands to a P3-AT robot.

5.1 Sample Code for GPS Data Acquisition

This code captures data from the GPS and shows it on the terminal. Let us start creating a new package for our application called *gps_test_pkg* which should depend on the package *gpsd_client*,

```
$ sudo roscreate-pkg gps_test_pkg gpsd_client
```

Now, go to the main folder of the *gps_test_pkg* package,

```
$ roscd gps_test_pkg
```

and create the *src* folder,

```
$ mkdir src
```

Use your favorite text editor to create a file named *gpsTest.cpp* and copy and paste the following C++ code on it,

```
#include <iostream>
#include <ros/ros.h>
#include <gps_common/GPSFix.h>

class GpsTest
{
public:
  // Type for GPS messages
  gps_common::GPSFix gpsMsg;

  // Constructor
  GpsTest(ros::NodeHandle nh_) : n(nh_)
  {
    // Subscribing to the topic /fix
    gps_sub = n.subscribe("/fix", 100, &GpsTest::gpsCallback, this);
  }

  // Callback Function for the GPS
  void gpsCallback(const gps_common::GPSFixConstPtr &msg)
  {
    gpsMsg = *msg;
  }

private:
  // Nodehandle
  ros::NodeHandle n;

  // Subscriber
  ros::Subscriber gps_sub;
};

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

  // Variables to store the Latitude and Longitude from the GPS respectively
  double gpsLat = 0;
  double gpsLong = 0;

  // Initializing the node for the GPS
  ros::init(argc, argv, "gps_Subscriber");
  ros::NodeHandle nh_;

  GpsTest *p = new GpsTest(nh_);

  // Getting the data from the GPS
  gpsLong = p->gpsMsg.longitude;
  gpsLat = p->gpsMsg.latitude;

  std::cout << "Current Latitude: " << gpsLat << std::endl;
  std::cout << "Current Longitude " << gpsLong << std::endl;

  ros::spin();
  return 0;
}
```

5.2 Compiling and Running the Node

Now, go to the main folder of the package and use your favorite text editor to add the following line to the *CMakeList.txt* file,

```
rosbuild_add_executable(gpsTest src/gpsTest.cpp)
```

Now, proceed to compile the package

```
$ sudo rosmake --rosdep-install
```

and you should be able to run the node,

```
$ rosrun gps_test_pkg gpsTest
```

and visualize the latitude and longitude coordinates given by the GPS on the terminal.

5.3 The Code Explained

Let us put the code in pieces to make it clear. The following section of the code is for including the necessary C++ libraries and packages:

```
001 #include <iostream>
002 #include <ros/ros.h>
003 #include <gps_common/GPSFix.h>
```

Then, we declare a class containing all the functions and variables we need to subscribe to the topic containing the GPS data,

```
005 class GpsTest
006 {
```

Afterwards, we declare the public attributes of the class. The variable gpsMsg stores the GPS data so we can access it from the main function,

```
public:
  // Type for GPS messages
  gps_common::GPSFix gpsMsg;
```

Now, we create the constructor of the class. In this case the constructor is the function which allows us to encapsulate the subscriber *gps_sub*. We subscribe the node to the */fix* topic using a callback of type class method (check section 2.3.2 at Publishers and Subscribers). Since we are encapsulating the nodehandle *n* we pass the nodehandle *nh_* from the main function via member initializers to the constructor. The colon (:) in the header separates the input parameter from the member initializer.

```
  // Constructor
  GpsTest(ros::NodeHandle nh_) : n(nh_)
  {
    // Subscribing to the topic /fix
    gps_sub = n.subscribe("/fix", 100, &GpsTest::gpsCallback, this);
  }
```

Then, we write the callback signature (check section 2.2 at Publishers and Subscribers) for the callback associated to the GPS. Notice that this callback puts the information data from the GPS in the public variable *gpsMsg*.

```
  // Callback Function for the GPS
  void gpsCallback(const gps_common::GPSFixConstPtr &msg)
  {
    gpsMsg = *msg;
  }
```

Now, we encapsulate the nodehandle *n* and the subscriber *gps_sub* by declaring them private variables in the class,

```
private:
  // Nodehandle
  ros::NodeHandle n;

  // Subscriber
  ros::Subscriber gps_sub;
};
```

Then, the main function starts,

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

The following section initializes the node with the label *gps_Subscriber* and declares the nodehandle *nh_*.

```
  // Initializing the node for the GPS
  ros::init(argc, argv, "gps_Subscriber");
  ros::NodeHandle nh_;
```

In the following lines, the *new* operator assigns a storage of the proper size for the object *p* of type *GpsTest*. We call the constructor to initialize the object and to return a pointer to the type *GpsTest*.

```
  GpsTest *p = new GpsTest(nh_);
```

Then, We use the arrow ($->$) operator to access the member *gpsMsg* of the class *GpsTest* to get the latitude and longitude values from the GPS. We proceed to show the data on the terminal by using the conventional *std::cout* commands. Finally the *ros::spin* command waits for new messages from the GPS to run the callback.

```
  // Getting the data from the GPS
  gpsLong = p->gpsMsg.longitude;
  gpsLat = p->gpsMsg.latitude;

  std::cout << "Current Latitude: " << gpsLat << std::endl;
  std::cout << "Current Longitude " << gpsLong << std::endl;

  ros::spin();
  return 0;
}
```

For a list of the available data given by the *gps_common/GPSFix* messages besides latitude and longitude, check gps_common/GPSFix.

5.4 Sample Code for Speed Control

This very simple code moves the robot at a constant linear and angular speed. Let us create a new package called *robot_driver_pkg* which depends on the package *p2os_driver*,

```
$ sudo roscreate-pkg robot_driver_pkg p2os_driver
```

Now, go to the main folder of the *robot_driver_pkg* package,

```
$ roscd robot_driver_pkg
```

and create the *src* folder,

```
$ mkdir src
```

Use your favorite text editor to create a file named *robotDriver.cpp* and copy and paste the following C++ code on it,

```
#include <iostream>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <p2os_driver/p2os.h>

class RobotDriver
{
public:
  // Constructor
  RobotDriver(ros::NodeHandle &nh)
  {
    nh_ = nh;
    // publishing to the topic cmd_vel
    cmd_vel_pub_ = nh_.advertise<geometry_msgs::Twist>("cmd_vel", 1);
    // publishing to the topic cmd_motor_state
    cmd_motor_pub_ = nh_.advertise<p2os_driver::MotorState>("cmd_motor_state",1000);
  }

  bool drive()
  {
    // Commands of type "twist" for the linear and angular velocity
    geometry_msgs::Twist base_cmd;
    // Commands of type "p2os_driver" for the motor state
    p2os_driver::MotorState motor_cmd;
    motor_cmd.state = 1;// enabling motors

    base_cmd.linear.x = base_cmd.linear.y = base_cmd.angular.z = 0; // Speed = 0 by default

    ros::Rate loop_rate(10); // message requency 10 hz
    while(ros::ok()){

        base_cmd.linear.x = 0.1;

        // publishing the speed command on topic cmd_vel
        cmd_vel_pub_.publish(base_cmd);
        // publishing the motor command on topic cmd_motor_state
  cmd_motor_pub_.publish(motor_cmd);

  ros::spinOnce();
  loop_rate.sleep();
    }
    return true;
  }
private:
  // Nodehandle
  ros::NodeHandle nh_;
  // Publishers
  ros::Publisher cmd_vel_pub_;
  ros::Publisher cmd_motor_pub_;
};

int main(int argc, char** argv)
{
  //init the ROS node
  ros::init(argc, argv, "robot_driver");
  ros::NodeHandle nh;

  RobotDriver driver(nh);
  driver.drive();
}
```

5.5 Compiling and Running the Node

First of all, be sure that the robot is at the floor level in an obstacle-free area. Now, go to the main folder of the package and use your favorite text editor to add the following line to the *CMakeList.txt* file,

```
rosbuild_add_executable(robotDriver src/robotDriver.cpp)
```

Now, proceed to compile the package

```
$ sudo rosmake --rosdep-install
```

and you should be able to run the node,

```
$ rosrun robot_driver_pkg robotDriver
```

and the robot should start moving straighforward at 0.1 m/s.

5.6 The Code Explained

In order to make the code clear we put it in pieces. The following section includes the required libraries,

```
#include <iostream>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <p2os_driver/p2os.h>
```

Now, we declare the class which contains all the necessary functions and variables,

```
class RobotDriver
{
```

Then, we declare the public attributes. The constructor receives the nodehandle *nh* via member initializers. We publish a message of type *geometry_msgs::Twist* containing the linear and angular velocities to the topic *cmd_vel* (see geometry_msgs/Twist Message). Furthermore we publish a message of type *cmd_motor_state* which contains the enabled/disabled state of the motors of the robot.

```
public:
  // Constructor
  RobotDriver(ros::NodeHandle &nh)
  {
    nh_ = nh;
    // publishing to the topic cmd_vel
    cmd_vel_pub_ = nh_.advertise<geometry_msgs::Twist>("cmd_vel", 1);
    // publishing to the topic cmd_motor_state
    cmd_motor_pub_ = nh_.advertise<p2os_driver::MotorState>("cmd_motor_state",1000);
  }
```

In the following section we declare the variables *base_cmd* and *motor_cmd* which store the messages of the velocities and the motor state respectively. We enable the motors by setting the value *motor_cmd.state* to any integer value different than zero. Lastly, we set all the initial velocities (angular and linear) to zero.

```
  bool drive()
  {
    // Commands of type "twist" for the linear and angular velocity
    geometry_msgs::Twist base_cmd;
    // Commands of type "p2os_driver" for the motor state
    p2os_driver::MotorState motor_cmd;

    motor_cmd.state = 1;// enabling motors

    base_cmd.linear.x = base_cmd.linear.y = base_cmd.angular.z = 0; // Speed = 0 by default
```

Then, we set the frequency of the algorithm to 10 hz with the *ros::Rate loop_rate(10)* command. At every cycle we set the value of the linear speed at 0.1 m/s and proceed to publish the messages in the variables *base_cmd* and *motor_cmd* to the topics *cmd_vel* and *cmd_motor_state* respectively.

```
    ros::Rate loop_rate(10); // message requency 10 hz
    while(ros::ok()){

        base_cmd.linear.x = 0.1;

        // publishing the speed command on topic cmd_vel
        cmd_vel_pub_.publish(base_cmd);
        // publishing the motor command on topic cmd_motor_state
  cmd_motor_pub_.publish(motor_cmd);
```

The *ros:spinOnce()* command in the next section is not necessary in this algorithm since we are not receiving callbacks from any topic. Recall that in section 5.1 every time the GPS sends a message a callback is executed to store the message in a variable *GPSMsg* but this is not the case here. However, we encourage the reader to get used adding this line as a good programming routine. The command *loop_rate.sleep()* will guarantee that the frequency of the while loop will be about 10 hz.

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

Lastly, we find the *main* function. The first two lines initialize the *robot_driver* node and the nodehandle *nh*. The third line creates an object called driver of type RobotDriver and the fourth line executes the function drive from the created object.

```
int main(int argc, char** argv)
{
  //init the ROS node
  ros::init(argc, argv, "robot_driver");
  ros::NodeHandle nh;

  RobotDriver driver(nh);
  driver.drive();
}
```

6 Conclusions

We have provided the basics of the technical procedure to configure a set of sensors and actuators attached to a P3-AT robots using ROS. Given the intrinsic complexity involved in hardware configuration procedures and the lack of documentation in that sense, we aimed to provide a better understanding of the experimental design related to mobile sensor networks. Even though there are a lot of devices not covered by the document, we hope the reader would feel encouraged enough to try out new devices and configurations.

References

[1] B. Lavis, Y. Yokokohji, and T. Furukawa, “Estimation and control for cooperative autonomous searching in crowded human emergencies,” in *IEEE International Conference on Robotics and Automation*, 2008, pp. 2831–2836.

[2] E. M. Craparo, J. P. How, and E. Modiano, “Simultaneous placemente and assignment for exploration in mobile backbone networks,” in *IEEE Conference on Decision and Control*, December 2008, pp. 1696–1701.

[3] E. W. Frew, , and J. Elston, “Target assignment for integrated search and tracking by active robots networks,” in *IEEE International Conference on Robotics and Automation*, May 2008, pp. 2345–2359.

[4] S. Ferrari, R. Fierro, B. Perteet, C. Cai, and K. Baumgartner, “A geometric optimization approach to detecting and intercepting dynamic targets using a mobile sensor network,” *SIAM Journal on Control Optimzation*, vol. 48, no. 1, pp. 292–320, 2009.

[5] J.-M. Luna, R. Fierro, C. Abdallah, and J. Wood, “An adaptive coverage control algorithm for deployment of nonholonomic mobile sensors,” in *IEEE International Conference on Decision and Control*, December 2010.

[6] R. Cortez, X. Papageorgiou, H. G. Tanner, A. V. Klimenko, K. N. Borozdin, , R. Lumia, and W. Priedhorsky, "Smart radiation sensor management: Nuclear search and mapping using mobile robots," *IEEE Robotics and Automation Magazine*, vol. 15, no. 3, pp. 85–93, September 2008.

[7] N. Michael, J. Fink, and V. Kumar, "Experimental testbed for large multirobot teams: Verification and validation," *IEEE Robotics and Automation Magazine*, vol. 15, no. 1, pp. 53–61, March 2008.

[8] M. Schwager, J. McLurkin, J. Slotine, and D. Rus, "From theory to practice: Distributed coverage control experiments with groups of robots," in *Proc. of the International Symposium on Experimental Robotics*, July 2008.

[9] D. Cruz, J. McClintock, B. Perteet, O. Orqueda, and Y.Cao, "Decentralized cooperative control: A multi-vehicle platform for research in networkrd embedded systems," *IEEE Control Systems Magazine*, vol. 27, no. 3, pp. 58–78, June 2007.

[10] R. A. Cortez, J. Luna, R.Fierro, and J. Wood, "Multi-vehicle testbed for decentralized environmental sensing," in *IEEE International Conference on Robotics and Automation*, 2010, pp. 1052–1058.

[11] "Pioneer 3 operations manual," [online] Available: http://robots.mobilerobots.com/docs/all_docs/P3OpMan6.pdf, 2010.

[12] "Ros/ introduction," [online] Available: http://www.ros.org/wiki/ROS/Introduction, 2010.

[13] "Mobile robots, autonomous mobile robots cores, bases and accesories," [online] Available: http://www.mobilerobots.com/Mobile_Robots.aspx, 2010.

[14] "Adept, your intelligent robotics partner," [online] Available: http://www.adept.com/, 2010.

[15] J.-M. Luna, "Distributed, adaptive deployment for nonholonomic mobile sensor networks: Theory and experiments," Master's thesis, The University of New Mexico, November 2009.

[16] "Willow garage," [online] Available: http://www.willowgarage.com/, 2010.

[17] "Understanding ros nodes," [online] Available: http://www.ros.org/wiki/ROS/Tutorials/UnderstandingNodes, 2010.

[18] "Understanding ros topics," [online] Available: http://www.ros.org/wiki/ROS/Tutorials/UnderstandingTopics, 2010.

[19] "Scanning laser range finder uhg-08lx specifications," [online] Available: http://www.acroname.com/robotics/parts/R311-HOKUYO-LASER2s.pdf, 2007.

[20] "Range-finder type laser scanner urg-04lx specifications," [online] Available: http://www.acroname.com/robotics/parts/R283-HOKUYO-LASER1s.pdf, 2005.

[21] "hokuyo_node package summary," [online] Available: http://www.ros.org/wiki/hokuyo_node, 2010.

[22] "Gps 18 technical specifications," [online] Available: http://static.garmincdn.com/pumac/425_TechnicalSpecification.pdf, 2005.

[23] "Gps 18 xtechnical specifications," [online] Available: http://static.garmincdn.com/pumac/GPS18x_TechnicalSpecifications.pdf, 2008.

[24] "gpsd a gps service daemon," [online] Available: http://gpsd.berlios.de/, 2010.

[25] "git the fast version control system," [online] Available: http://git-scm.com/, 2010.

[26] "gpsd_client," [online] Available: http://www.ros.org/wiki/gpsd_client, 2010.

[27] “3dm-gx2 gyro enhanced orientation sensor,” [online] Available: http://www.microstrain.com/pdf/3dm-gx2_datasheet_v1.pdf, 2007.

[28] “Subversion,” [online] Available: http://subversion.apache.org/, 2010.

[29] “Product manual. 1018 - phidgetinterfacekit 8/8/8,” [online] Available: http://www.phidgets.com/documentation/Phidgets/1018.pdf, 2007.

[30] “Product manual. 1127 - precision light sensor,” [online] Available: http://www.phidgets.com/documentation/Phidgets/1127.pdf, 2007.

[31] “Product manual. 1108 - magnetic sensor,” [online] Available: http://www.phidgets.com/documentation/Phidgets/1108.pdf, 2007.

[32] “Getting started with p2os,” [online] Available: http://www.ros.org/wiki/p2os/Tutorials/Getting%20Started%20with%20p2os, 2010.