Documentation for ROS and Turtlebot

CS189r

Spring 2016

1 Using rospy

A general overview of rospy is available here: `http://wiki.ros.org/rospy/Overview`.

In practice, we use rospy mostly to (a) initialize and shutdown the Turtlebot, (b) print messages to the console, and (c) establish publishers and subscribers.

1.1 Initialize

When writing a program using the ROS architecture stack, you should initialize a *node* – one node per process. Nodes collectively form a graph which facilitates internode communication via ROS streams, services, and parameters. For a full discussion of nodes, read: `http://wiki.ros.org/Nodes`. For now, you need only initialize a named node using the format as follows:

```
rospy.init_node(``my_node_name'')
```

1.2 Shutdown

Ctrl + C is the standard keyboard input for shutdown when running a Rospy script. On shutdown, the current script is executed and the robot state becomes static. Hence it is necessary to reduce the motor speeds before shutdown. This can be achieved as follows:

```
rospy.on_shutdown(self.shutdown)
```

where self.shutdown is a function (or handler) in which the robot state is handled immediately prior to the script's exit. An example shutdown functions is:

```
def shutdown(self):
        rospy.loginfo(``Stop!'')
        self.cmd_vel.publish(Twist())
        rospy.sleep(1)
```

Control flow code is generally contained within a loop of the following variety:

```
while not rospy.is_shutdown():
        #your code here
```

1.3 Print messages

Printing messages to the console is every bit as simple as we'd expect. The following code snippet prints Hello World:

```
rospy.loginfo(''Hello World!'')
```

1.4 Publishers & Subscribers

In order to understand the vital role of publishers and subscribers when coding with rospy, it is advisable to first have an understanding of ROS topics. A full overview can be found here: `http://wiki.ros.org/Topics`. Fundamentally, a topic is a named and accessible queue over which nodes can exchange messages. A topic is strongly typed to a message; only information of a particular message type can be published to a topic.

In order to publish information to a topic, you must first initialize the publisher. This can be achieved in the following way:

```
pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size=10)
```

Note that you must have imported std_msgs in order for the above publisher initialization to succeed.

Having named the above publisher as "pub", we can now publish messages to that topic as follows:

```
pub.publish(''hello world'')
```

There are alternate styles for achieving this message publishing. Namely, you can explicitly assign the argument "hello world" to be a std_msg.msg.String, you can use the implicit implementation wherein the datatype of the argument is non-explicit but rather inferred, or you can use an implicit style along with keywords. This last option is advantageous when the message type is a tuple.

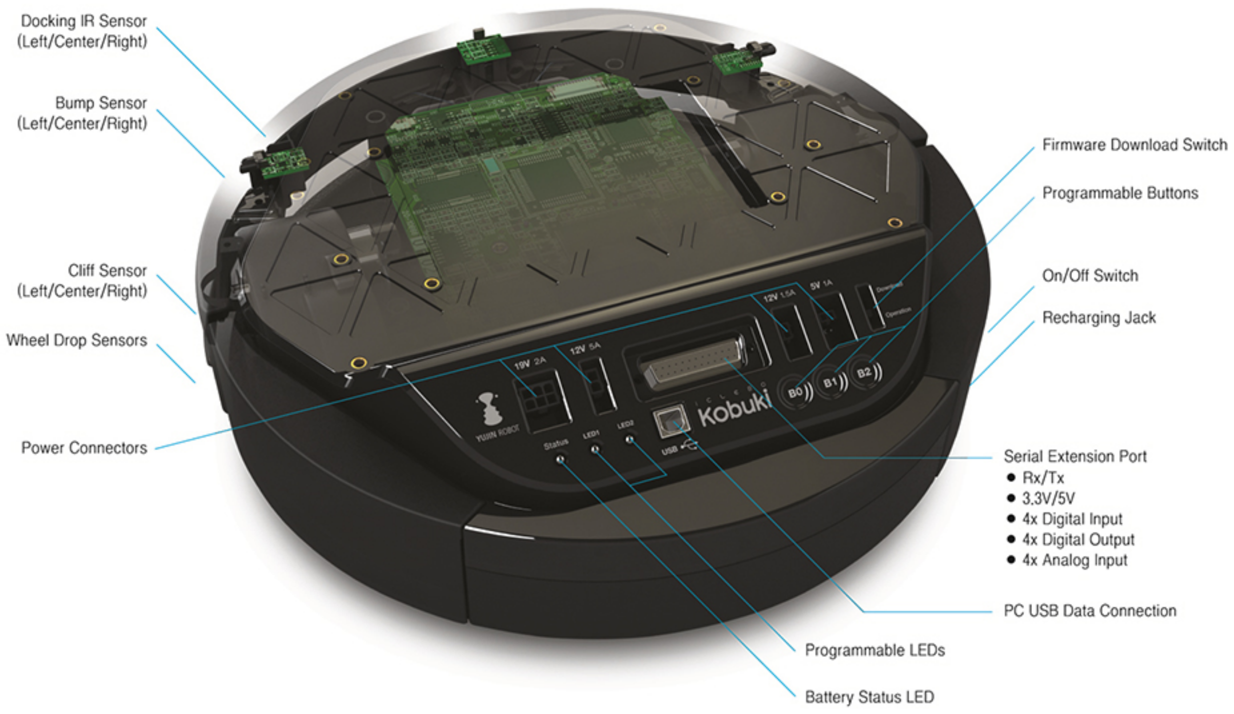
By default, a publisher is synchronous; passing the argument queue_size defaults instead to the recommended asynchronous implementation.

Subscribing is a more intuitive concept. Assuming a topic exists, you can subscribe in the following way:

```
rospy.Subscriber('topic_name', std_msgs.msg.String, process_topic)
```

Where process_topic is a function which processes any information received from the topic topic_name. For example, if topic_name consisted of any information relating to the state of the bump sensors on the Kobuki base, then process_topic should be called when an event, e.g. the bumper is pressed or released, occurs.

2 Kobuki Base:



Pictured above is the Kobuki base. Familiarize yourself with the sensors and I/O options that Kobuki provides. Here, we'll focus on the bump sensors, the cliff sensors, the wheel drop sensors, and the programmable buttons.

2.1 Bump Sensor

All events relating to the Bump sensor are sent over a publisher when you run the launch command:

```
roslaunch turtlebot_bringup minimal.launch
```

Hence, in order to react to bump events, we subscribe to the Bump puplisher. An example follows:

```
rospy.Subscriber('mobile_base/events/bumper', BumperEvent, processBump)
```

In which mobile_base/events/bumper is the name of the Node, BumperEvent is the type of information published to the Node, and processBump is a function we provide which processes each event. Effectively, we are passing information from the bump sensor to processBump as a parameter; we run our event handling in the function processBump.

Assuming 'bump' is a global variable, one possible implementation of processBump is as follows:

```
# If bump data is received, process the data
# data.bumper: LEFT (0), CENTER (1), RIGHT (2)
# data.state: RELEASED (0), PRESSED (1)
```

```
    def processBump(data):
        global bump
        if (data.state == BumperEvent.PRESSED):
            bump = True
        else:
            bump = False
       rospy.loginfo(''Bumper Event'')
       rospy.loginfo(data.bumper)
```

2.2 Cliff Sensors, Wheel Drop Sensors, Programmable Buttons, Etc.

Event handling for any sensing event is similar to how we handle a bump event. Hence, we will not go into detail about the remaining sensors on the Kobuki base. However, we will distribute code which provides an overview of these sensors, and additional information can be found on the following github: `https://github.com/yujinrobot/kobuki_msgs/tree/indigo/msg`.

2.3 LEDs

In order to publish data to the LEDs on the Kobuki base, we establish two publishers:

```
# Publish led1
self.led1 =
        rospy.Publisher('/mobile_base/commands/led1', Led, queue_size=10)

# Publish led2
self.led2 =
        rospy.Publisher('/mobile_base/commands/led2', Led, queue_size=10)
```

When we want to use the LEDs to output information, we publish LED states to the publishers above, as follows:

```
    self.led1.publish(Led.RED)
    self.led2.publish(Led.GREEN)
```

3 Motor Control

To control the motors on the Kobuki base, we set up a publisher which passes messages to a ROS Node. In particular, we use a line as follows to access the cmd_vel_mux/input/navi node:

```
self.cmd_vel =
      rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
```

This publisher send a queue of up to 10 Twist-type messages to the node which controls the Kobuki motors.

3.1 Twist Datatype

We import the Twist datatype from geometry_msgs as in

```
    from geometry_msgs.msg import Twist
```

A Twist message is composed of two 3-vectors of the form (x, y, z), one of which expresses the desired linear velocity, another of which expresses the desired angular velocity. For example, if we want the robot to move in a straight line at 0.2 meters per second, we could publish the following Twist message to the publisher:

```
move_cmd = Twist()
move_cmd.linear.x = 0.2
self.cmd_vel.publish(move_cmd)
```

Or, if we want the robot to turn 45 degrees, we may publish instead:

```
turn_cmd = Twist()
turn_cmd.linear.x = 0
turn_cmd.angular.z = radians(45)
self.cmd_vel.publish(turn_cmd)
```

4 Kinect Data

5 Quick Documentation Links

ROS:

- Introduction: `http://wiki.ros.org/ROS/Introduction`
- Overview: `http://wiki.ros.org/ROS/Concepts`
- Nodes: `http://wiki.ros.org/Nodes`
- Topics: `http://wiki.ros.org/Topics`

rospy:

- Overview: `http://wiki.ros.org/rospy/Overview`
- Logging messages: `http://wiki.ros.org/rospy_tutorials/Tutorials/Logging`
- Initialize, Shutdown: `http://wiki.ros.org/rospy/Overview/Initialization%20and%20Shutdown`
- Publishers, Subscribers: `http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers`

Kobuki (sensors):

- Overview: `http://wiki.ros.org/kobuki_msgs`
- Examining the ROS Nodes: `http://wiki.ros.org/kobuki/Tutorials/Examine%20Kobuki`
- Code examples: `https://github.com/yujinrobot/kobuki_msgs/tree/indigo/msg`
- Data types for sensors: `http://docs.ros.org/indigo/api/kobuki_msgs/html/index-msg.html`
- Sensors: `http://wiki.ros.org/kobuki/Tutorials/Examine%20Kobuki`

Motors:

- Twist geometry message: `http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html`