Control of a Parrot Mambo drone via bluetooth

Germany, October, 2018
Edited by

Edwin Uriel Perez Loaiza
Stefan Blumauer
Technische Hochschule Ingolstadt

Abstract

In the laboratory there is wok done on Parrot Bebop drones via wi-fi communication and the programming language: C++. The work done on them is mainly using Linux operating system and ROS (Robotic operating system). There is a node that publishes command values and another node for manual control using a joy stick. The task that was entrusted to us (Edwin Perez and Mr. Stefan Blumauer) was to use a different drone, parrot mambo, that had bluetooth communication and make the programming compatible to the publishers on ROS and to make the programming similar to the logic of the C++ program already done for the parrot bebop drone.

Table of contents

Introduction

There are many drones in Dr. Frey's laboratory. The majority of them are the Parrot bebop drones. There was a new drone in the laboratory, smaller in size. It is the parrot mambo drone and it has bluetooth communication. The Bebop Drones use wi-fi and are programmed in ROS (Robotic operating system) with the C++ language. Due to the reduced space that the laboratory has, we wanted to find out if the mambo drone could work as the bebop works even if it had a different type of communication. The application-Programming-Interface (API) needs to be adapted, in order for it to run in ROS as a subscriber node, just as the bebop drones do. This question was raised because the mambo drone is smaller and because of that fact there would be more flying space and more drones could be used in that flying space.

Objective

Find an API compatible to the drone and control the Parrot Mambo drone via bluetooth. Have static results by having it take off and land and to have dynamic results by having it move backward, forward, leftward and rightward. After making sure the communication works, the next step is to run it as a node in ROS and making it behave as it did on the editor, the code will need to be adapted to it. When finished, the node will be modified in order for it to serve as a subscribing node to the topics the parrot bebop drone is subscribed to. The drone should be controlled manually and automatically, depending on the topics it is subscribed to.

Requirements

- ***Drone***:
  - Parrot Mambo
- ***Joystick:***
  - thrustmaster hotas warthog joystick
- **Python**
  - Version: 3.5
- ***Software:***
  - ROS (Robot operating system)
  - Editor: Atom
- ***Operating system:***
  - Linux
  - Distributor: Ubuntu 16.04.5 LTS
- ***Bluetooth Module***
  - ASUS USB-BT400 USB Adapter

Getting started

In general there is a lot of information out there for the Parrot Bebop drone, like: drivers and tutorials. But there isn't a lot of information for the Mambo drone and a Mambo driver for ROS; it's difficult to find. Fist, it was necessary to make sure that bluetooth communication is possible. As time went by, a bluetooth module was bought, ASUS USB-BT400 USB Adapter. The computer's version is 4.2 and the module's version is 4.0, on the other hand the module could be moved closer to the flying space due to the USB cord. Because of this, a package needed to be found that would permit someone to program an algorithm for the Mambo drone on a platform using a programming language. Fortunately, a package was found: a Python interface for Parrot Drones. Pyparrot was designed and implemented by Dr. Amy McGovern to program Parrot Mambo and Parrot Bebop 2 drones using python. This interface was developed to teach K-20 STEM concepts (programming, math, and more) by programming a drone to fly autonomously.

Installing

To start installing one must keep in mind that the mambo drone has BLE (Bluetooth Low Energy) communication. The website/tutorial where we started the installation process is this: https://pyparrot.readthedocs.io/en/latest/installation.html, this link takes us to the documentation of this package and it also serves as a tutorial for the installation process. To complete the installation on Linux it was necessary to follow the instructions on this site with the exception of installing python through anaconda because python had already been installed on our computer; it's important to mention that a python version higher than 3 is necessary for everything to run properly. One last thing we did differently was installing the software editor: Atom, instead of visual studio, which was mentioned as a requirement in the tutorial but the editor Atom will do.

Testing demo codes

In the same link, the next step is to start with a demo code. The name of this topic in the tutorial is: Quick Start Guide with a Minidrone. Before testing anything the page tells us to find our Minidrone's specific address via bluetooth. If you have wi-fi this is also necessary. The page shows a list of steps so one can find the address with a set of commands, but this was done differently and easier in our case; bluetooth communication was activated on the computer, then the add option was selected in order to have a device scan and finally the mambo drone was found. When you find it, press next to connect to it. If this does not work, try disabling the pair option in the pin options. The address and connection status will be shown on the main bluetooth settings window. Doing this, we got the following address: *"E0:14:A3:A5:3D:FD"*.

After finding the address, it's possible to test the demo code. In the laboratory there's a reduced space that should be considered, especially when the commands and values are new. Because of what was mentioned earlier, it was decided to test the drone progressively, that is a step at a time. The first step was to have static movement. The code **(see figure 1)** has commands for various types of movements; the first step was to take off and land to be safe. Taking the demo code as a guide, a simple code in Atom was devoloped. It started by importing the Mambo library from the Pyparrot package. Then it was necessary to insert the address of said drone.

Later, the program will need to identify the mambo drone, which includes the address, and if it will be communicated to via wi-fi or not. It was set to false because it has bluetooth communication. If this code is used for a wi-fi communication then it should be changed to: True. The part of the code where it has commands to connect to the drone was left the same with the

exception of increasing the number in the command: “num_retries” to 20. The command “num_retries” is for the number of times the program will try to connect to the drone if at first it fails to do so. This number can be changed to any number but It’s recommended that it doesn’t go below 7 so it can have a fair amount of tries, this part is shown in the following image:

```
"""
Demo the trick flying for the python interface

Author: Amy McGovern
"""

from pyparrot.Minidrone import Mambo

# you will need to change this to the address of YOUR mambo
mamboAddr = "e0:14:d0:63:3d:d0"

# make my mambo object
# remember to set True/False for the wifi depending on if you are using the wifi or the BLE to connect
mambo = Mambo(mamboAddr, use_wifi=True)

print("trying to connect")
success = mambo.connect(num_retries=3)
print("connected: %s" % success)

if (success):
    # get the state information
    print("sleeping")
    mambo.smart_sleep(2)
    mambo.ask_for_state_update()
    mambo.smart_sleep(2)

    print("taking off!")
    mambo.safe_takeoff(5)

    if (mambo.sensors.flying_state != "emergency"):
        print("flying state is %s" % mambo.sensors.flying_state)
        print("Flying direct: going up")
        mambo.fly_direct(roll=0, pitch=0, yaw=0, vertical_movement=20, duration=1)

        print("flip left")
        print("flying state is %s" % mambo.sensors.flying_state)
        success = mambo.flip(direction="left")
        print("mambo flip result %s" % success)
        mambo.smart_sleep(5)

        print("flip right")
        print("flying state is %s" % mambo.sensors.flying_state)
        success = mambo.flip(direction="right")
        print("mambo flip result %s" % success)
        mambo.smart_sleep(5)

        print("flip front")
        print("flying state is %s" % mambo.sensors.flying_state)
        success = mambo.flip(direction="front")
        print("mambo flip result %s" % success)
        mambo.smart_sleep(5)

        print("flip back")
        print("flying state is %s" % mambo.sensors.flying_state)
        success = mambo.flip(direction="back")
        print("mambo flip result %s" % success)
        mambo.smart_sleep(5)

        print("landing")
        print("flying state is %s" % mambo.sensors.flying_state)
        mambo.safe_land(5)
        mambo.smart_sleep(5)

    print("disconnect")
    mambo.disconnect()
```

Figure 1. The demo code for a Mambo drone[1]

The documentation page has a section where it explains the commands. The explanation to the following commands can be found in the following link: https://pyparrot.readthedocs.io/en/latest/minidronecommands.html. The code has a condition; if it is

1 Quick Start Guide with a Minidrone, Quick start: Demo Code, Demo the trick flying for the python interface, Author: Amy McGovern, accessed 23 October, 2018, https://pyparrot.readthedocs.io/en/latest/quickstartminidrone.html

successful in connecting to the drone, then it will start the movement commands but not before writing the code to get the state information. The drone will start taking off with the command: *mambo.safe_takeoff()*. There are different ways to take off but this is the recommended one. The time in the parenthesis is the time out time. Then, the command *mambo.smart_sleep()* is used, this works similar to a delay function; it uses seconds and can be changed depending on how fast one wants it to work but it is *absolutely* necessary if you are using a bluetooth connection, so it doesn't disconnect. Landing has the same logic, the command for this is: mambo.safe_land(), and it also uses smart.sleep. Finally, we disconnect from the drone using: mambo.disconnect(). All of this can be seen in the following **figure (2):**

```python
from pyparrot.Minidrone import Mambo

#The address of our mambo drone according to he address shown
mamboAddr = "E0:14:A3:A5:3D:FD"

# make my mambo object
# remember to set True/False for the wifi depending on if we are using the wifi or the BLE to connect
mambo = Mambo(mamboAddr, use_wifi=False)

print("trying to connect")
success = mambo.connect(num_retries=20)
#connect(num_retries) connect to the Minidrone using BLE
#You can specify a maximum number of re-tries. Returns true if the connection suceeded or False otherwise.
print("connected: %s" % success)
if (success):
  # get the state information
    print("sleeping")
    mambo.smart_sleep(2)
    mambo.ask_for_state_update()
    mambo.smart_sleep(2)
    #smart_sleep(seconds) This sleeps the number of seconds (which can be a floating point) but wakes for all BLE or wifi notifications
    #if you are using BLE: This comamnd is VERY important. As your BLE will disconnect regularly if we use anoe command
    print("taking off!")
    mambo.safe_takeoff(5)
    mambo.smart_sleep(5)
    #we give it 5 seconds to time out

    print("landing")
    mambo.safe_land(5)
    mambo.smart_sleep(5)

    print("disconnect")
    mambo.disconnect()
```

Figure 2

To make the drone go forward, backward and sideways, it was necessary to implement a new command to the code. The command is: *mambo.fly_direct(roll=0, pitch=0, yaw=0, vertical_movement=0, duration=0.5)*. A 3D body can be rotated about three orthogonal axes, these rotations are referred to as yaw, pitch, and roll (**figure 3)**:

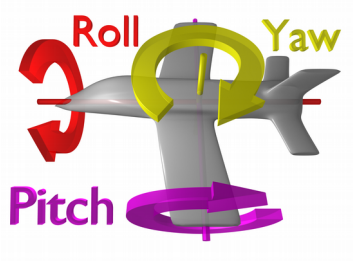

Figure 3

Wanting to have dynamics, It was necessary to take in account the axes. It's easy to see that the drone needs to tilt forward so it can fly forward. The commands are repeated for duration seconds. Each value ranges from -100 to 100 and is essentially a percentage and direction of the max_tilt (for roll/pitch) or max_vertical_speed (for vertical movement)[2]. If you want the drone to tilt on one side,

2 Minidrone commands and sensors, Flying, https://pyparrot.readthedocs.io/en/latest/minidronecommands.html

two of the rotors will begin spinning faster on one side, creating more lift. In doing so, some of the upward lift becomes more of a sideways force, causing the drone to move forward or backwards. In the same manner, different prop speeds will spin the drone[3]. Pitch was set to 50 to go forward and negative 50 for it to go backward. The duration was set to 0.5 seconds because this indicates how long the commands will be repeated and in a reduced space this needs to be controlled. After this, it's necessary to type mambo.smart_sleep() so it doesn't disconnect. Finally, it's disconnected from the drone. This code can be seen in the following **figure (4)**:

```
print("trying to connect")
success = mambo.connect(num_retries=20)
#The command "num_retries" is for the number of times the program will try to connect to the drone if at first it fails to do so.
#This number can be changed to any number but It's recommended that it doesn't go below 7 so it can have a fair amount of tries
print("connected: %s" % success)
if (success): #If success
  # get the state information
    print("sleeping")
    mambo.smart_sleep(2)
    mambo.ask_for_state_update()
    mambo.smart_sleep(2)
    #smart_sleep(seconds) This sleeps the number of seconds (which can be a floating point) but wakes for all BLE or wifi notifications
    #if you are using BLE: This comamnd is VERY important. As your BLE will disconnect regularly if we use another command
    print("taking off!")
    mambo.safe_takeoff(5) #we give it 5 seconds to time out
    mambo.smart_sleep(5) #5 seconds of delay


    print("Flying direct: going forward (positive pitch)")
    mambo.fly_direct(roll=0, pitch=50, yaw=0, vertical_movement=0, duration=0.5)
    mambo.smart_sleep(2)
    #Each value ranges from -100 to 100 and is essentially a percentage and direction of the max_tilt (for roll/pitch) or max_vertical_speed (for vertical movement)
    #We are using the pitch value to tilt it to go foward
    #duration of the movement is in seconds

    print("Flying direct: going backwards (negative pitch)")
    mambo.fly_direct(roll=0, pitch=-50, yaw=0, vertical_movement=0, duration=0.5)
    mambo.smart_sleep(2)
    #We are using the pitch value, negatively, to tilt it to go backwards

    print("Flying direct: roll")
    mambo.fly_direct(roll=50, pitch=0, yaw=0, vertical_movement=0, duration=0.5)
    mambo.smart_sleep(2)
    #We are using the roll value to tilt it to go left

    print("Flying direct: roll")
    mambo.fly_direct(roll=-50, pitch=0, yaw=0, vertical_movement=0, duration=0.5)
    mambo.smart_sleep(2)
    #We are using the roll value to tilt it to go right

    print("landing")
    mambo.safe_land(5)
    mambo.smart_sleep(5)

    print("disconnect")
    mambo.disconnect()
```

Figure 4

Writing the Subscriber Node

After doing all of the above, we were ready to start writing a subscriber node in ROS. To start writing node, it was necessary to create a catkin package. This implies more things, like having a catkin workspace; it's highly recommended to create a workspace for catkin. A catkin workspace is a folder where you modify, build, and install catkin packages[4]. After creating the workspace, we created the package for our project. Information and guidance on how to do this can be found on Ros.org/tutorials/Creating a ROS Package.

3 How drones work, Nathan Chandler, https://science.howstuffworks.com/transport/flight/modern/drones5.htm

4 Writing a Simple Publisher and Subscriber (Python), http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

Writing a publisher/subscriber node can also be found in ROS tutorials but it only serves as a guide. It has an example and it explains the code. In the subscriber example code **(see figure 5)**, it starts with: #!/usr/bin/env python. Every Python ROS Node will have this declaration at the top. The first line makes sure your script is executed as a Python script[5].

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Figure 5

When first used this and compiled, it showed an error and it was because the code was being compiled with python 2.7 instead of 3.5. To correct this you can set your python3 or higher as the default or you can add 3.5 to the end of the directory (the version may differ depending on your version but it must be higher than 3). Something like this: #!/usr/bin/env python3.5. Later, the importation of the package for the mambo was added, the address was added, functions were added and then all the functions were called into one where it subscribed to the corresponding ROS topics.

Breaking this up into parts **(see figure 6)**, first the “rospy” was imported because you need to import it if you are writing a ROS node. Importing libraries, the same syntax that was used in the last code (demo code) is used for this as well in order to use the mambo commands. From the import of string messages we also added: Empty, because we don’t use string messages and the publisher node indicates that it publishes empty messages to its static movements. This part was ended with the declaration of the mambo’s address.

5 Writing a Simple Publisher and Subscriber (Python), the code explained, http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

Next, the necessary functions were added. Just as the last one we worked with, we decided to have static movement; taking off and landing. To do this, 3 functions were implemented; one for taking off, another for landing and one to call them both while subscribing. The first function is taking off. The corresponding code to take off is typed. Something we weren't familiar with as unexperienced programmers in python was the use of *self.* To access/create instance variables for any object, we always need self as a reference to the object under use, especially in functions it requires self (can be any user defined name) as the first argument to refer to the object who called the function[6]. And we use pass at the end to pass the argument. We do the same thing for the landing function but with it's corresponding command.

Before explaining the last function, it's necessary to mention that the code for the joystick, manual control, only published for bebop topics. To subscribe to the topics being published we needed to add it to the code. This wasn't difficult because we didn't have to create a new code or modify the one there. What was done was add to it. The code is meant to publish to 5 bebop drones. All that was done was add mambo as another drone to control setting it as a variable, copying publishing code for the bebops but writing them under the mambo name. One more thing that was added was to have a type of *geometry_msgs::Twist* for the mambo.

Finally, we add one last function that will subscribe to the joystick node. We start by rospy.init_node(NAME, ...), this is very important as it tells rospy the name of your node -- until rospy has this information, it cannot start communicating with the ROS Master[7]. In this case, our node will take on the name mambo. We then subscribe to the topics we want and call the functions to enable their actuation. We continue with: rospy.Subscriber("/mambo/takeoff", Empty, takeoff). This declares that our node subscribes to the /mambo/takeoff topic which is of type Empty, takeoff is invoked. This is then repeated for landing. Finally, we end with rospy.spin(), this keeps your node from exiting until the node has been shutdown.

6 Poorna Pragna Poorna Pragna, Machine Learning and Deep Learning Practitioner, https://www.quora.com/What-does-self-mean-in-python-class-Why-do-we-need-it

7 Writing a Simple Publisher and Subscriber (Python), the code explained, http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

```python
#!/usr/bin/env python3.5
import rospy
from std_msgs.msg import String, Empty
from pyparrot.Minidrone import Mambo
import sys, signal

#The address of our mambo drone according to he address shown
mamboAddr = "E0:14:A3:A5:3D:FD"

def takeoff(self): #Function to take off
    print("taking off!")
    mambo.safe_takeoff(1)
    pass

def land(self):
    print("landing")
    mambo.safe_land(5)
    mambo.smart_sleep(5)
    mambo.disconnect()
    pass

def mambo_functions():

    rospy.init_node('mambo')
    rospy.Subscriber("/mambo/takeoff", Empty, takeoff)
    rospy.Subscriber("/mambo/land", Empty, land)
    rospy.spin()


if __name__ == '__main__':
    print("trying to connect")
    success = mambo.connect(num_retries=2)
    #connect(num_retries) connect to the Minidrone using BLE
    #You can specify a maximum number of re-tries. Returns true if the connection suceeded or False otherwise.
    print("connected: %s" % success)
    if (success):
        mambo_functions()
```

Figure 6

Making adjustments to takeoff and land

Landing and taking off needed to be more restricted so it wouldn't interfere with each other. For this reason a flag was created. This flag holds the state we want to start with and helps us with the steps we want to follow. It's called "in_the_air". It can be inferred by its name that it's meant to represent when the drone is in the air. Initially it is set to false due to the fact that the drone is off and then turned on and put on the ground. Next, a condition is put in the *takeoff* function, if the *in_the_air* is false, then the *mambo.safe_takeoff()* can be executed and finally the state *in_the_air* is set to true. The state "in_the_air" has to be declared globally in the function because if it's not then it won't be recognized and there will be a syntax error. Similar logic is used for the landing command. First, there is a condition to know if the drone is_the_air because that's the only reason it would have to land. Then, the state *in_the_air* is set to false and the commands to land it are executed (**see figure 7**). Don't forget to declare "in_the_air". There are three landing prints because we wanted to know if the code was executed up to that line of the code, they can be deleted.

```
mamboAddr = "E0:14:A3:A5:3D:FD"

# make my mambo object
# remember to set True/False for the wifi depending on if we are using the wifi or the BLE to connect
mambo = Mambo(mamboAddr, use_wifi=False)
#Flags that will help us know the state of our drone
in_the_air = False # this is set to false because when we start the drone, it is waiting for a command on the ground
wait = 0

def takeoff(self): #Function to take off
    global in_the_air #we have to declare it once more because the function won't recognize it if it's out of it.
#    rospy.loginfo(rospy.get_caller_id())
    if in_the_air == False:
        print("taking off!")
        mambo.safe_takeoff(3)
        in_the_air = True #the state changes to true because it has taken off
#        mambo.smart_sleep(3)
    pass

def land(self):
    global in_the_air
    global rospy
    if in_the_air == True:
        in_the_air = False
        print("landing")
        mambo.safe_land(3)
        mambo.smart_sleep(3)
        print("landing2")
        rospy.signal_shutdown("mambo landed")
        print("landing3")
    pass
```

Figure 7

Adding movements

In the figure above (**figure 7**) *wait = 0* can be shown below *in_the_air.* What is it for? It's a counter that we first set to 0. Why do we need it? Because we want to control the rate at which something is executed. Before we continue explaining the purpose of *wait,* it's necessary to remember that we can't execute any movements if we're not in the air. That being said, another condition is put in a new function that will be called "movement" with the same syntax as the other's, self and pass. In that function there will be a condition which will permit the rest of the code in that condition to be executed only if the state is satisfied, *in_the_air == True.* After, the counter *wait* comes into place. The counter is initiated with the code *wait=wait+1*, this counts the number of seconds starting at zero (explained before). Another condition is put so when ever the counter reaches 10 then it will permit the code in that condition be executed. The number of seconds has been experimented on and as a result we came to the conclusion that 10 hertz was the best frequency for the drone to execute movement. When the frequency was set higher or lower than this, the drone reacted with a considerable delay to the joystick movements. The publisher to the subscriber communication is set to 100 hertz but the drone cannot process commands at this frequency and this is the reason that the execution of the joystick commands was forced to work at 10 hertz. In the condition, *fly_direct(roll, pitch, yaw, vertical_movement, duration)* was used for the movement. Taking into consideration the axes from the figure below (**figure 7**) and the three orthogonal axes of an aircraft (see figure 3), the movement direction was set.

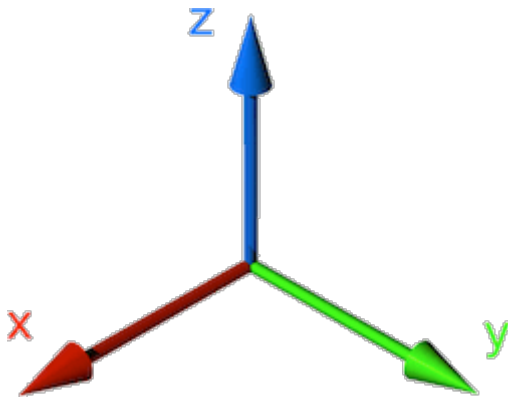

Figure 7

*roll=(int(data.linear.y * 100)*(-1)), pitch=int(data.linear.x * 100), yaw=(int(data.angular.z * 100)*(-1)), vertical_movement=int(data.linear.z * 100)*

The *data.linear.x*, *y* and *z* were taken from the manual control code from the joystick. It was turned into integer values with “int” and multiplied by 100 to have values ranging from 0 to 100. The *duration* was set to 0.1 which is in the tenth place. We didn’t notice a a real difference when making the number smaller, for instance: 0.01, 0.001, etc. The code looked like this: *mambo.fly_direct(roll=(int(data.linear.y * 100)*(-1)), pitch=int(data.linear.x * 100), yaw=(int(data.angular.z * 100)*(-1)), vertical_movement=int(data.linear.z * 100), duration=0.01).* Finally the counter is set to 0 once more so it can go through the process once more, code in **figure 8**.

```
def movement(data):
    global in_the_air
    global wait
    global mambo

    if in_the_air == True: #this can only happen when the mambo is in the air
        wait=wait+1 #counter. This counts in seconds starting at 0
        if wait == 10: # when the counter gets to 0 then this condition is met and executed
            mambo.fly_direct(roll=(int(data.linear.y * 100)*(-1)), pitch=int(data.linear.x * 100), yaw=(int(data.angular.z * 100)*(-1)), vertical_movement=int(data.linear.z * 100), duratio
#the values from above, e.g data.linear.y, are taken from the "manual control code". It's multiplied by 100 to get values from 0 to 100 and it is turned to integer values
            wait = 0; #it's set to 0 again so the process can repeat
#        wait=wait+1
pass
```

Figure 8

Callback

The *mambo_functions()* function is where the program subscribes to the topics, declares what message type and calls the functions it will invoke, as is done here and in figure 9: *rospy.Subscriber("/mambo/takeoff", Empty, takeoff).* Then *rospy.spin()*, It is mainly used to prevent your Python Main thread from exiting.

As the code was tested, we noticed that the drone wasn’t disconnecting when it landed as we intended it to. We had to terminate the process with Ctrl-C at times. Due to this, we investigated a different way and we came across a register handler that is called when rospy process begins shutdown. By default, exited on Ctrl-C. You can request a callback using rospy.on_shutdown() when your node is about to begin shutdown. This will be invoked before actual shutdown occurs, so

you can perform service and parameter server calls safely. Messages are not guaranteed to be published[8] (**see figure 9**).

```
def shutdown_hook():
    global in_the_air

    if in_the_air == True:
        print("exit_controlled: landing mambo")
        mambo.safe_land(3)
        in_the_air = False
    print ('rospy is going down')




def mambo_functions(): #the subscribing node
    global landed
    global rospy

    rospy.init_node('mambo') #initiate the code
    rospy.Subscriber("/mambo/takeoff", Empty, takeoff) #to what topic, type of message and function to call
    rospy.Subscriber("/mambo/land", Empty, land)
    rospy.Subscriber("/mambo/cmd_vel", Twist, movement)
    rospy.on_shutdown(shutdown_hook) #Register handler to be called when rospy process begins shutdown. Request a callback
    rospy.spin() #It is mainly used to prevent your Python Main thread from exiting
```

Figure 9

Main

Finally, we come to the main function. This is where the basic commands for the mambo to connect and execute If true is used again as discussed in earlier code. Here we use the code that was written for the demo code but we left some things out, like asking for the state information and the individual commands for the different movements and replaced them with the invocation of the function: *mambo_functions()*, which includes every movement. When it comes to an end, landing, it will disconnect using *mambo.disconnect()* and *sys.exit()*. They disconnect from the mambo and exit from the system respectively.

Results

When the programs are compiled and run, the drone does what it was intended to do, referring to static positions and dynamic movements. Using the ASUS USB_BT400 USB Adapter helped to connect faster to the drone. Sometimes when trying to connect to the drone, it disconnects and tries to reconnect again. Also, when landing the drone, instead of disconnecting right away, it takes too much time doing this because it tries to reconnect so it can disconnect properly.

Discussion

Disconnecting and trying to reconnect again is not frequent but we assume that when it happens it’s due to the versions of the firmware being used in the drone and the adapter we use. The

8 Registering shutdown hooks, rospy.on_shutdown(h), http://wiki.ros.org/rospy/Overview/Initialization%20and%20Shutdown

landing situation is not completely understood why it happens. There must be something in the code we are missing or not putting in its right place.

Conclusion

The mambo drone could be used as the bebop drone by the use of bluetooth instead of wi-fi. C++ wasn't used to program it, python3.5 was. The API could be adapted to ROS in order for it to work as well as it did on the editor. It could also be modified and used as a subscriber node to the joystick's publishing node. Due to the fact that this drone is smaller than a bebop drone and that it can be used with ROS makes it a good option to do work on. All of this was done using python but in the future it will be tried with C++ because the bebops use this language.

Bibliography

http://wiki.ros.org/rospy/Overview/Initialization%20and%20Shutdown
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29
https://www.quora.com/What-does-self-mean-in-python-class-Why-do-we-need-it
https://science.howstuffworks.com/transport/flight/modern/drones5.htm
https://pyparrot.readthedocs.io/en/latest/minidronecommands.html
https://pyparrot.readthedocs.io/en/latest/quickstartminidrone.html