Raspberry Pi <-> APM 2.6
Communication and Control

Varanon Austin Pukasamsombut 留学生
Field Robotics Group, Tohoku University
Prof. Yoshida Kazuya, Assoc. Prof. Keiji Nagatani

March 3, 2015

Outline

1. Introduction

This tutorial aims to teach the following:

- Connection of the APM 2.6 to the Raspberry Pi through UART0.
- Using ROS (Robot Operating System) to communicate with the APM.
- Creating and running programs with ROS to control the APM.

Emphasis will be put on sections where other tutorials are not available.

Forewarning

- The methods explained in this tutorial are only one of many ways to achieve the same results.
- Programming will always lead to new errors and problems, so time and patience will be required.
- Achieving the final result of this tutorial will most likely take longer than a week, or possibly a month, if you are new to programming.
- Most things on this tutorial was found through Google.

My Methods

My methods were done in the following conditions:

- Raspberry Pi Model B+ with a 32 GB microSD card running Raspbian
- APM 2.6 with ArduCopter v3.2.1
- ROS Indigo installed from source
- Most communication with the Raspberry Pi was done through SSH by an Ethernet Cable from a Laptop running Windows 7 using PuTTy.

2. Basic Knowledge

Programming Tip:

Don’t skip the tutorials.

Understanding the concepts will help you solve errors, so properly go through tutorials without relying on “copy-and-paste.”

Here are things you should know for this tutorial:

Raspberry Pi

- Learn the basics of the Raspberry Pi and get it set-up with Raspbian.

  http://www.raspberrypi.org/help/quick-start-guide/

- Understand how to use the command line (Unix Shell) on the Raspberry Pi to manipulate and navigate through the system. *Don’t rely on a Desktop*, since going through it with the Shell will help you understand the code more and make you a better programmer.

  http://linuxcommand.org/lc3_learning_the_shell.php

Programming

- Be able to understand both basic Python and C++. Most programs written for Linux based systems (like the Raspberry Pi) will run on either Python or C++. Python will be needed to read those programs, and C++ will be used for the programs we write.

  http://www.learnpython.org/

  http://www.cplusplus.com/doc/tutorial/

- You will also need to learn how to use a text editor in the command window on the Raspberry Pi. There are many choices. (I use Vim.)

  Type “vimtutor” into the Linux Command Terminal.

APM 2.6

- You should already be able to manually fly your UAV with the APM. I used Mission Planner for all of my APM tuning. All information related to the APM can be found through the wiki.

  http://copter.ardupilot.com/

- You should also understand a little of how the APM 2.6 code works and how it communicates with the ground station using MAVLink, which we will be using as the main means of communicating between the Raspberry Pi and APM.

  http://qgroundcontrol.org/mavlink/start

3. Establishing Connection

This tutorial was used in order to connect the Raspberry Pi with the APM.

http://dev.ardupilot.com/wiki/companion-computers/raspberry-pi-via-mavlink/

The tutorial shows a Pixhawk, but the same can be done with the APM 2.6 by using the open Telemetry Port shown on its top. (Next slide)

The Raspberry Pi pins can be found in more detail here:

http://elinux.org/RPi_Low-level_peripherals

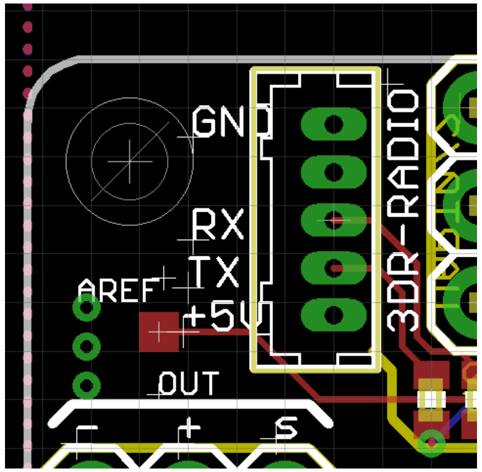
The Top Telemetry Port pins should be:

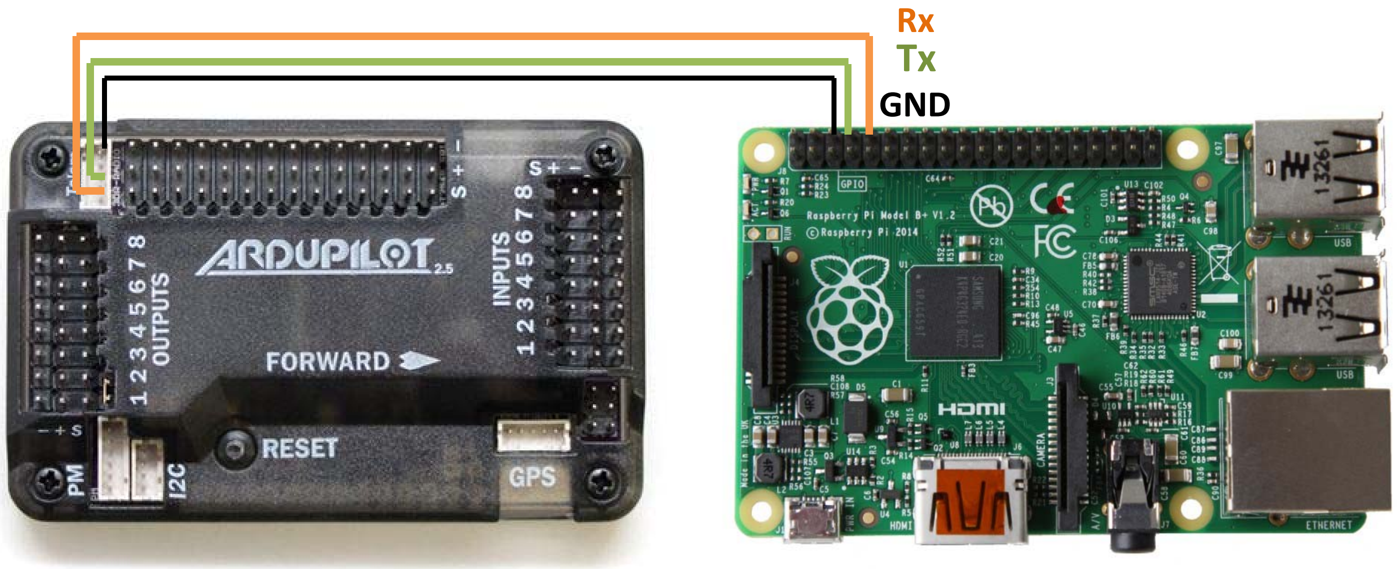
+5V, TX, RX, EMPTY, and GND. (Bottom to Top) Like this ----->

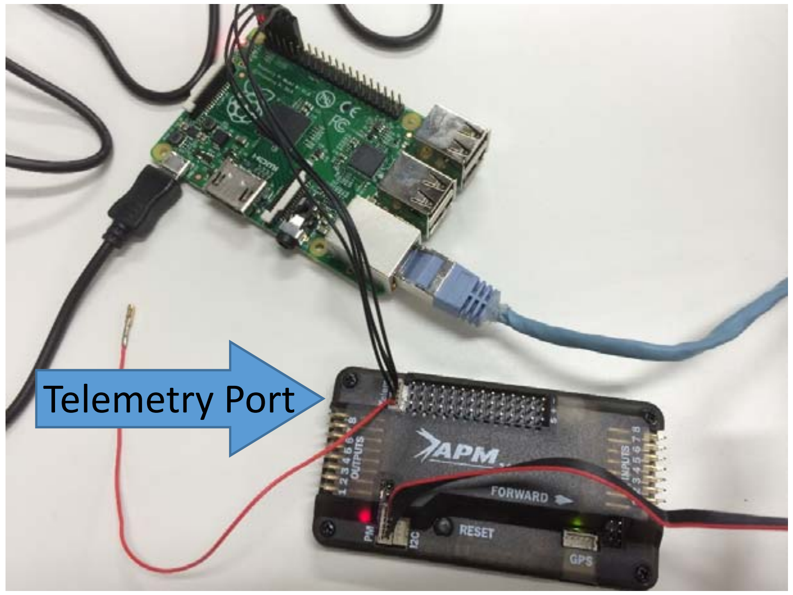
The +5V and EMPTY pin should stay disconnected.

TX = Transmitter, RX = Receiver. Connect TX <-> RX.

PCB Diagram of APM







3. Establishing Connection

In a simpler explanation, connect it this way:

APM 2.6		Raspberry Pi
GND	---------------------	GND
Empty		Open
RX	-----------------------	GPIO 14 (UART0_TXD)
TX	-----------------------	GPIO 15 (UART0_RXD)
+5V		Open

3. Establishing Connection

Important Note:

The Telemetry Port (UART0) on the APM and the USB Port uses the same serial port for connection, so there is a MUX (Multiplexer) that disables the Telemetry Port if the USB is connected. So,

you need a battery/power supply to power the APM without using a USB cable in order to use the UART0 Port and connect to the Raspberry Pi.

Once you are connected, follow through the rest of the tutorial in order to check the connection using MAVProxy. You should see a connection.

[Extra note: We will replace this program with *mavros* later on.]

Other Alternatives

Using the Telemetry Port (UART0) makes the APM lose the ability to send signals back to the Ground Station during flight with a transmitter. However, it is still able to receive signals from a RC transmitter.

If you would like to keep the Telemetry Port to connect to a Ground Station during flight, then you might want to try connecting using the UART2 Port on the APM (which may require soldering). I have not done this myself and more information can be found by looking on the DIYDrones Forum: http://diydrones.com/forum/topics/connecting-apm-2-0-and-raspberry-pi

4. Setting up ROS on RPi

Install ROS Indigo on the Raspberry Pi by following this tutorial:

http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Indigo%20on%20Raspberry%20Pi

Programming Tip: Try to Read and Understand some of the Code in the Tutorial instead of only copy-and-pasting. You will have to use these functions again later.

Important Note: Building your Workspace can take a LONG time (1 – 10 hours) depending on how many packages you are installing. If you are just going through the tutorial for the first time, it should take about 2-4 hours to complete it.

4. Setting up ROS on RPi

After setting up ROS, I recommend going through the tutorials and reading about ROS a bit through its wiki before trying to use ROS with the APM. This will help you understand what ROS is and how to program using ROS.
http://wiki.ros.org/ and http://wiki.ros.org/roscpp/Tutorials

(Make sure to be able to write a *Publisher* and *Subscriber* with C++, and understand "Using Class Methods as Callbacks," since I will be using it)

Note: Using ROS on the Raspberry Pi is different from using it on a computer, so some commands will not be the same as the ones in the tutorials. Also, you need an external monitor instead of using SSH in order to use RQT packages that are mentioned in the tutorial. Not all of the tutorials will be relevant to this tutorial.

I will be using many ROS terms in this tutorial from now on.

4. Setting up ROS on RPi

- If you are ever having trouble with programming with ROS or installing packages, just look at the tutorials again and see if you can find anything to help you.
- The Raspberry Pi ROS Installation tutorial will be really helpful for finding Raspberry Pi specific commands when installing packages.
- Programming Tip: It will be better to make multiple workspaces and sourcing them separately in order to avoid reinstalling many packages. For example, I have three separate workspaces and use:

```
source /opt/ros/indigo/setup.bash                 #My main ROS Install Location
source /home/pi/catkin_ws/devel/setup.bash        #Workspace used for tutorials
source /home/pi/my_catkin_ws/devel/setup.bash  #Workspace for my own code.
```

I automatically source all of my workspaces on start-up by adding it to the end of my .bashrc file.

5. Using ROS for APM Communication

In order to achieve APM Communication, I used mavros.

Mavros is a very useful package that gives you topics and nodes that you can publish or subscribe to in order to send commands or receive information from the APM through MAVLink during operation.

http://wiki.ros.org/mavros

Then install mavros and all dependencies that it requires.

(The next slide will show you how I installed it.)

5. Using ROS for APM Communication

Here is how I installed all my packages:

- Following the Raspberry Pi ROS installation tutorial, I installed all of the packages I think I will be using at once. (Just to make it faster)
- I installed: mavros, mavros_extras, hector_slam, hector_localization, hokuyo_node, ros_comm, ros_control, and joystick_drivers.
- This was all installed in the pre-existing “ros_catkin_ws” workspace that I created during the installation tutorial.
- This was installed exactly as described in the “Maintaining a Source Checkout” section on the tutorial page: http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Indigo%20on%20Raspberry%20Pi

5. Using ROS for APM Communication

- I used the following commands to install the packages:

```
$ cd ~/ros_catkin_ws

$ rosinstall_generator ros_comm ros_control joystick_drivers mavros mavros_extras hector_slam hector_localization
hokuyo_node  --rosdistro indigo --deps --wet-only --exclude roslisp --tar > avp_custom_1.rosinstall

$ wstool merge -t src avp_custom_1.rosinstall

$ wstool  update -t src

$ rosdep install --from-paths src --ignore-src --rosdistro indigo -y -r --os=debian:wheezy

$ sudo ./src/catkin/bin/catkin_make_isolated --install -DCMAKE_BUILD_TYPE=Release --install-space /opt/ros/indigo
```

- The rebuilding of the workspace took about 8-10 hours this time.
- If errors occur due to missing dependencies, you may need to install them externally as described in the tutorial page.

5. Using ROS for APM Communication

- Once mavros is properly installed, you can test it by running the command: $ roslaunch mavros apm2.launch

(Your APM should be powered on without USB and connected to the Raspberry Pi. No propellers)

- If you do not change the default fcu_url parameter of the apm2.launch file, then you will need to define the fcu_url manually through the Shell when launching. For example, with UART0, it is:

```
$ roslaunch mavros apm2.launch fcu_url:=/dev/ttyAMA0:57600
```

- The baud rate may also be 115200 instead of 57600 depending on the settings of your APM.

Note: FCU = Flight Control Unit

5. Using ROS for APM Communication

- If the connection was successful, your screen should be getting a heartbeat and show some data from the APM.

Programming Tip: It’s better to run programs like mavros and roscore in the background. (Look at Shell tutorial) You can also create multiple windows in PuTTY by right-clicking the window border and “Duplicate Session.” Multiple windows will help you look at more data at once.

- Get familiar with the mavros commands, topics, and nodes by listing them and using “echo” to see what kind of data they have.
- All of the mavros commands and information is here:

http://wiki.ros.org/mavros

5. Using ROS for APM Communication

- If you have trouble arming the APM, then try using Mission Planner to disable all Arming Checks and then try arming the APM again.
- It is recommended to use mavros to get a better understanding of ROS's concepts: topics and nodes, publishers and subscribers, etc..
- Test the APM with the UAV's motors to make sure it is working properly. (DO NOT USE PROPELLORS)
- Once you can arm the motors (`$rosrun mavros mavsafety arm`), try manipulating the motors from the command line by publishing into the "RC/override" topic. This overrides the RC transmitter with your given values. For example: (Next Slide)

5. Using ROS for APM Communication

```
$ rostopic pub mavros/rc/override mavros/OverrideRCIn -l '{channels: [ 0, 0, 1800, 0, 0, 0, 0, 0]}'
```

This command publishes the given message `'{channels: [ 0, 0, 1800, 0, 0, 0, 0, 0]}'` that is of message type `mavros/OverrideRCIn` into the topic `mavros/rc/override` and latches `-l` the message so that it stays.

This causes all channels to be controller by the RC transmitter, but overrides the third channel "Throttle" and sets it to a value of 1800. This should make the UAV motors spin faster (if it's armed). Disarm to stop the motors.

You can see which channels correlate to which radio signal from the RC by reading the topic `mavros/rc/in` and connecting an RC transmitter to the APM. Playing around with the joysticks should change the values of "channel" accordingly.

It is important that you understand how publishing messages and subscribing to topics work before creating your own program.

5. Using ROS for APM Communication

Note: Overridden Channels cannot be controlled by the RC transmitter anymore. They will remain “locked” until you set the channels back to 0, which gives the control back to the RC transmitter.

6. Writing your own Programs

Once you have a good understanding of ROS and mavros, you can start writing your own programs to control the APM.

I recommend creating a new workspace for your own code so that you may compile it without having to rebuild the entire ROS system.

Get help from the tutorial pages for creating your own workspace and making your own package with your own executable programs.

All of your actual code can be written in either C++ or Python, but I will be using C++.

You should already know how to write a basic ROS program.

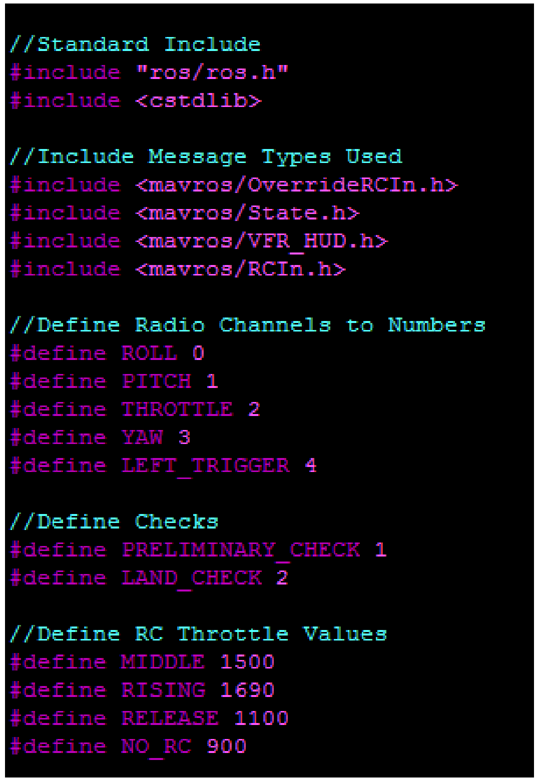
6. Writing your own Programs

Using my code as an example. (There are many ways to do this)

The Beginning of the Code

All message types that are used must be included.

Having DEFINEs really help to make a code easier to read.



```
//Standard Include
#include "ros/ros.h"
#include <cstdlib>

//Include Message Types Used
#include <mavros/OverrideRCIn.h>
#include <mavros/State.h>
#include <mavros/VFR_HUD.h>
#include <mavros/RCIn.h>

//Define Radio Channels to Numbers
#define ROLL 0
#define PITCH 1
#define THROTTLE 2
#define YAW 3
#define LEFT_TRIGGER 4

//Define Checks
#define PRELIMINARY_CHECK 1
#define LAND_CHECK 2

//Define RC Throttle Values
#define MIDDLE 1500
#define RISING 1690
#define RELEASE 1100
#define NO_RC 900
```

6. Writing your own Programs

- The first code I wrote was to make my Quadcopter Takeoff from the ground, hover in the air for a few seconds, and then land.
- Because of this, the only topics I needed to subscribe to were:

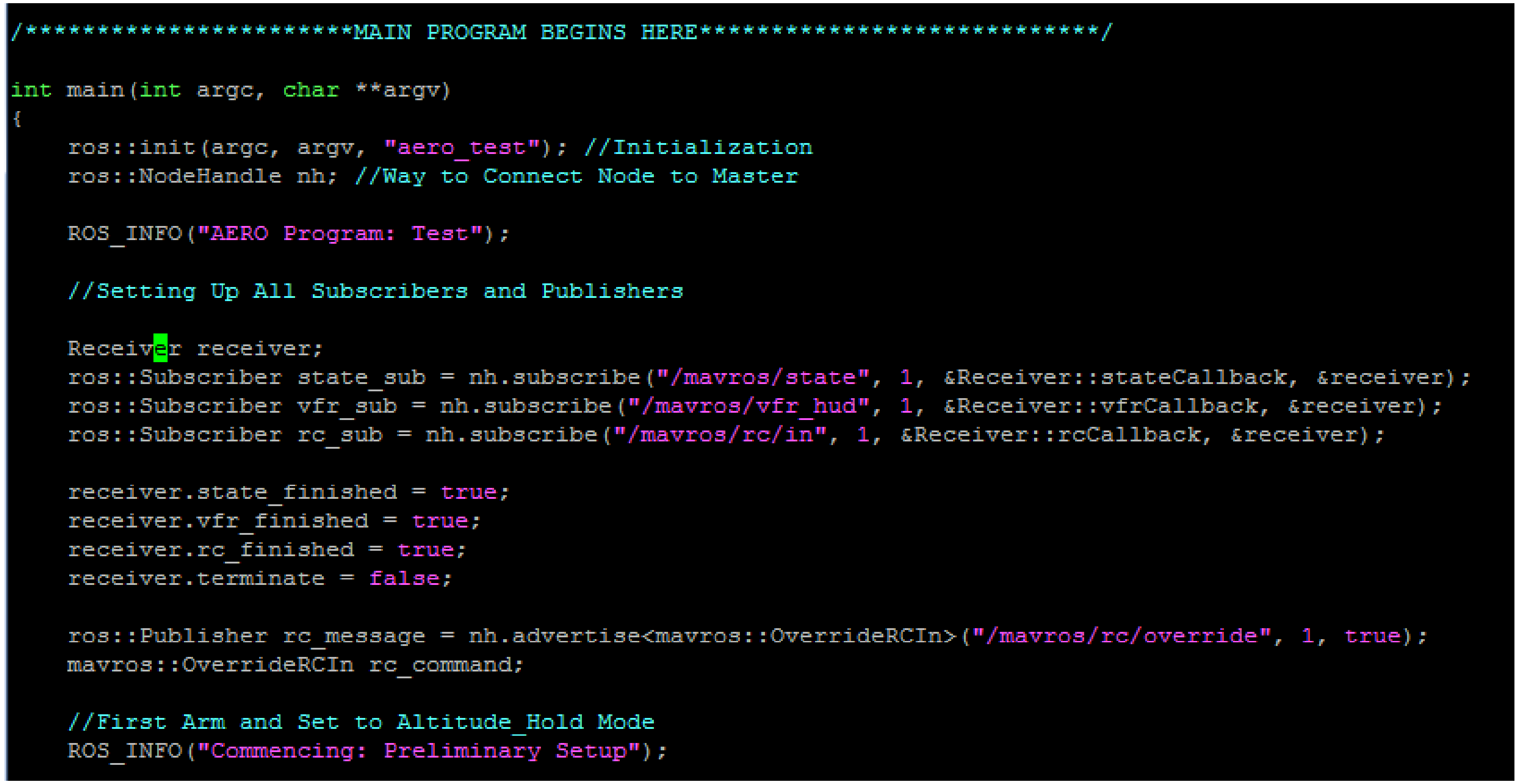
/mavros/state	:Obtain arming status and current flight mode.
/mavros/vfr_hud	:Obtain current altitude.
/mavros/rc/in	:Obtain RC transmitter channel values.

- I only needed to publish to one topic in order to fly the UAV:

/mavros/rc/override :Overrides RC channels with inputted values.

6. Writing your own Programs

My code for setting up all of my Subscribers and Publishers.

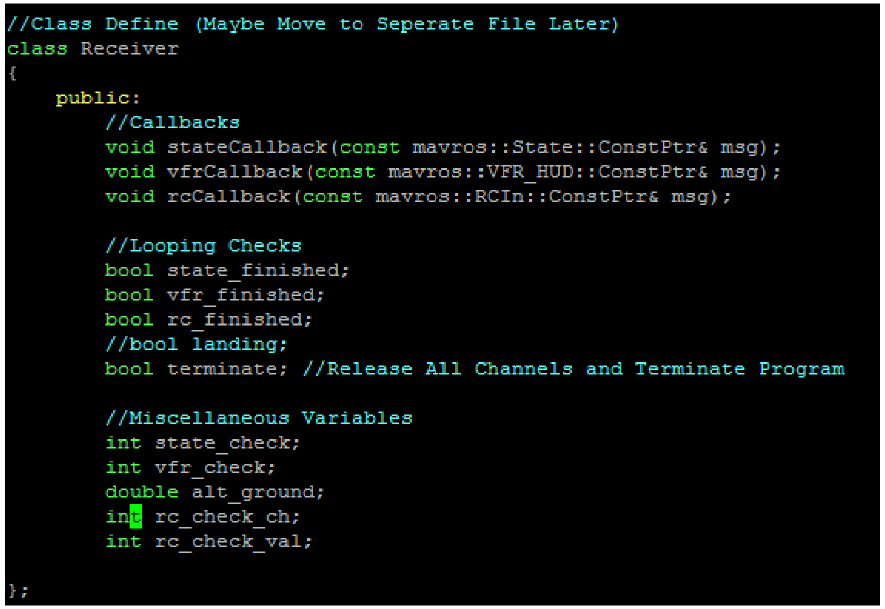


```
/************************MAIN PROGRAM BEGINS HERE******************************/

int main(int argc, char **argv)
{
    ros::init(argc, argv, "aero_test"); //Initialization
    ros::NodeHandle nh; //Way to Connect Node to Master

    ROS_INFO("AERO Program: Test");

    //Setting Up All Subscribers and Publishers

    Receiver receiver;
    ros::Subscriber state_sub = nh.subscribe("/mavros/state", 1, &Receiver::stateCallback, &receiver);
    ros::Subscriber vfr_sub = nh.subscribe("/mavros/vfr_hud", 1, &Receiver::vfrCallback, &receiver);
    ros::Subscriber rc_sub = nh.subscribe("/mavros/rc/in", 1, &Receiver::rcCallback, &receiver);

    receiver.state_finished = true;
    receiver.vfr_finished = true;
    receiver.rc_finished = true;
    receiver.terminate = false;

    ros::Publisher rc_message = nh.advertise<mavros::OverrideRCIn>("/mavros/rc/override", 1, true);
    mavros::OverrideRCIn rc_command;

    //First Arm and Set to Altitude_Hold Mode
    ROS_INFO("Commencing: Preliminary Setup");
```

6. Writing your own Programs

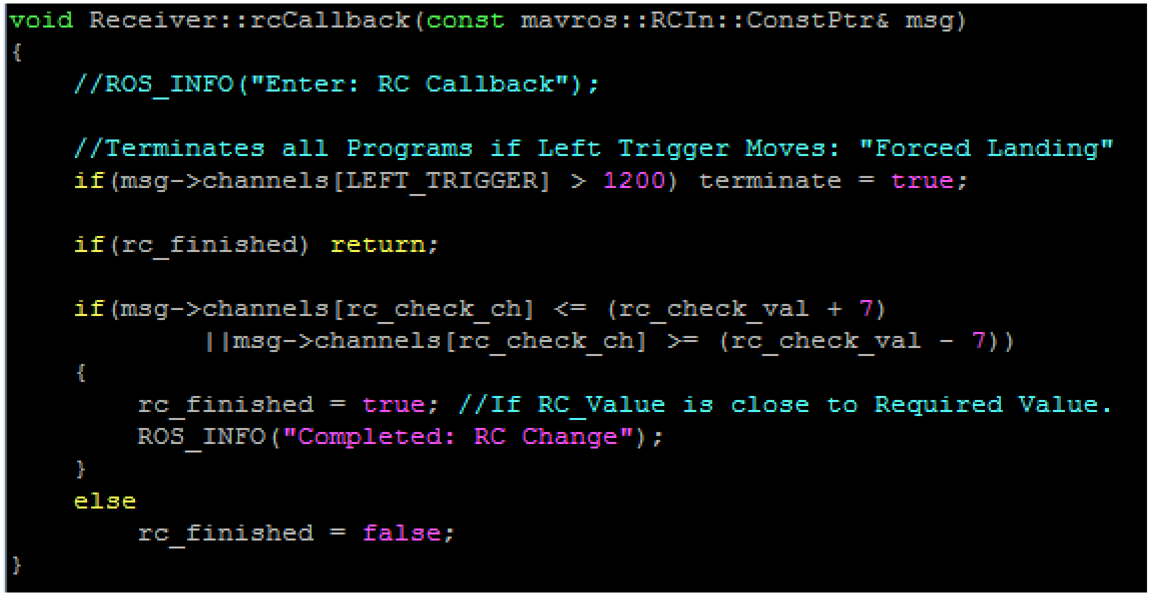
- Callbacks for the Subscribers must also be properly prepared beforehand, as detailed in the tutorial pages.

This is how I define my Callbacks in one Class.



```
//Class Define (Maybe Move to Seperate File Later)
class Receiver
{
    public:
        //Callbacks
        void stateCallback(const mavros::State::ConstPtr& msg);
        void vfrCallback(const mavros::VFR_HUD::ConstPtr& msg);
        void rcCallback(const mavros::RCIn::ConstPtr& msg);

        //Looping Checks
        bool state_finished;
        bool vfr_finished;
        bool rc_finished;
        //bool landing;
        bool terminate; //Release All Channels and Terminate Program

        //Miscellaneous Variables
        int state_check;
        int vfr_check;
        double alt_ground;
        int rc_check_ch;
        int rc_check_val;

};
```

This is an example of a Callback.



```
void Receiver::rcCallback(const mavros::RCIn::ConstPtr& msg)
{
    //ROS_INFO("Enter: RC Callback");

    //Terminates all Programs if Left Trigger Moves: "Forced Landing"
    if(msg->channels[LEFT_TRIGGER] > 1200) terminate = true;

    if(rc_finished) return;

    if(msg->channels[rc_check_ch] <= (rc_check_val + 7)
            ||msg->channels[rc_check_ch] >= (rc_check_val - 7))
    {
        rc_finished = true; //If RC_Value is close to Required Value.
        ROS_INFO("Completed: RC Change");
    }
    else
        rc_finished = false;
}
```

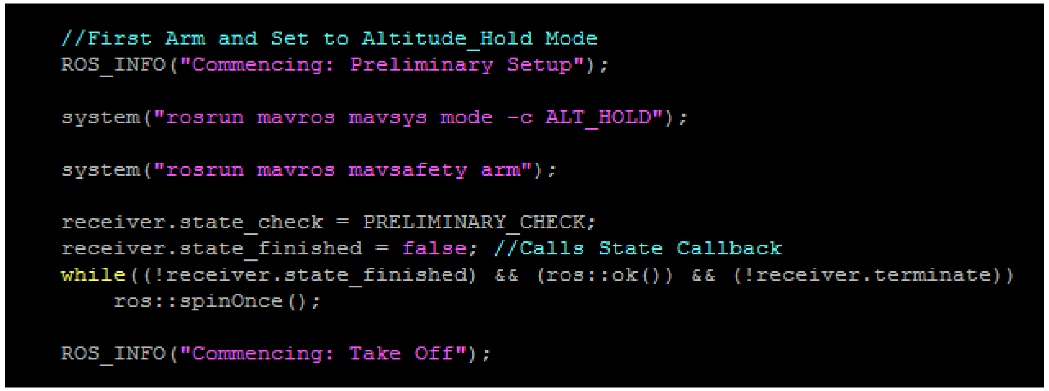
6. Writing your own Programs

- In order to call only one callback at a time without interfering with the other ones, I use Booleans that show if a callback is “finished.” If it is finished, it skips that callback during a *spin*.
- You will notice in my RC_Callback, I have a “terminate” Boolean if the Left Trigger value of the RC Transmitter goes above 1200. This is so that I am manually able to end the program using the RC transmitter. The terminate Boolean will cause the remaining program to be skipped and end. I also assigned the Left Trigger to cause the UAV to Land, which makes for a simple emergency switch.

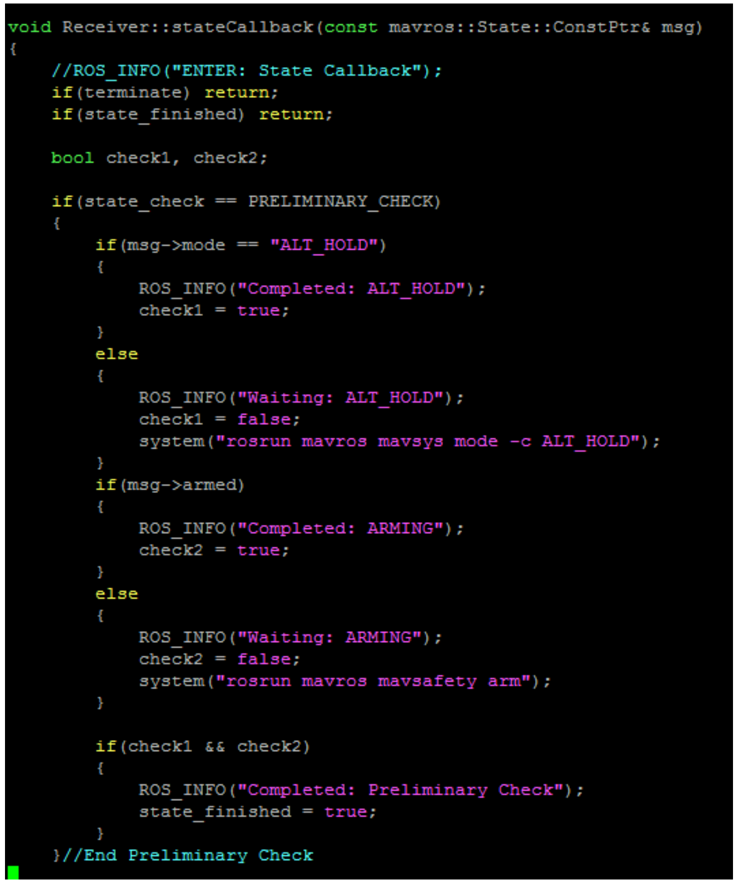
6. Writing your own Programs

Changing Modes and Arming

You can use the “system” function to run commands that would work from the command line, like arming or changing modes. (I use ALT_HOLD for flying and LAND) My program then enters the State Callback in order to check if it is properly armed and in the correct flight mode. The program exits the callback once the checks are passed or the program is told to forcibly terminate.



```
//First Arm and Set to Altitude_Hold Mode
ROS_INFO("Commencing: Preliminary Setup");

system("rosrun mavros mavsys mode -c ALT_HOLD");

system("rosrun mavros mavsafety arm");

receiver.state_check = PRELIMINARY_CHECK;
receiver.state_finished = false; //Calls State Callback
while((!receiver.state_finished) && (ros::ok()) && (!receiver.terminate))
    ros::spinOnce();

ROS_INFO("Commencing: Take Off");
```



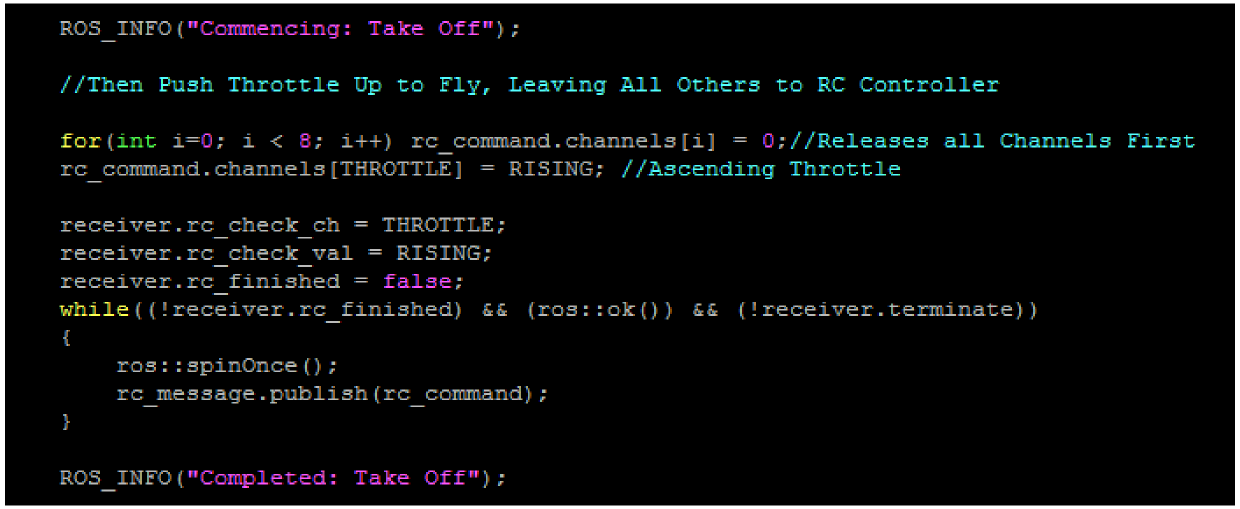
```
void Receiver::stateCallback(const mavros::State::ConstPtr& msg)
{
    //ROS_INFO("ENTER: State Callback");
    if(terminate) return;
    if(state_finished) return;

    bool check1, check2;

    if(state_check == PRELIMINARY_CHECK)
    {
        if(msg->mode == "ALT_HOLD")
        {
            ROS_INFO("Completed: ALT_HOLD");
            check1 = true;
        }
        else
        {
            ROS_INFO("Waiting: ALT_HOLD");
            check1 = false;
            system("rosrun mavros mavsys mode -c ALT_HOLD");
        }
        if(msg->armed)
        {
            ROS_INFO("Completed: ARMING");
            check2 = true;
        }
        else
        {
            ROS_INFO("Waiting: ARMING");
            check2 = false;
            system("rosrun mavros mavsafety arm");
        }

        if(check1 && check2)
        {
            ROS_INFO("Completed: Preliminary Check");
            state_finished = true;
        }
    }//End Preliminary Check
```

6. Writing your own Programs

Controlling the Motors

The motors are controller by sending override commands.
This is done in my code by changing the “channel” values of the previously created rc_command message, and then publishing it to the override topic.

The rc_callback is then called to make sure that the value was properly changed.



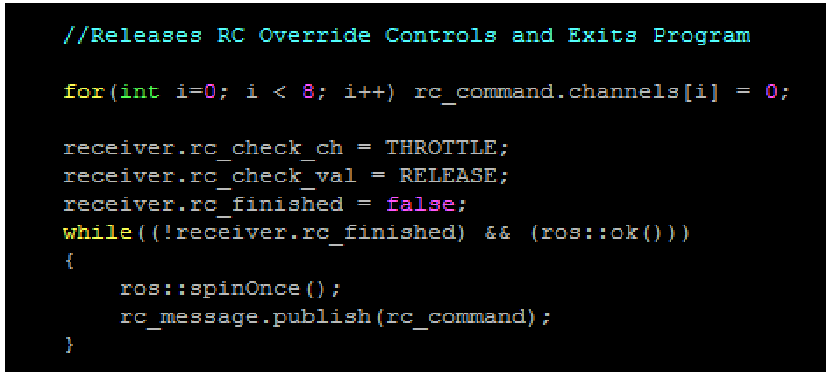
```
ROS_INFO("Commencing: Take Off");

//Then Push Throttle Up to Fly, Leaving All Others to RC Controller

for(int i=0; i < 8; i++) rc_command.channels[i] = 0;//Releases all Channels First
rc_command.channels[THROTTLE] = RISING; //Ascending Throttle

receiver.rc_check_ch = THROTTLE;
receiver.rc_check_val = RISING;
receiver.rc_finished = false;
while((!receiver.rc_finished) && (ros::ok()) && (!receiver.terminate))
{
    ros::spinOnce();
    rc_message.publish(rc_command);
}

ROS_INFO("Completed: Take Off");
```

6. Writing your own Programs

- A callback to check the status of your commands is necessary because it takes time for the published command to be received by the APM. That is why you should continuously publish the override messages until a change is seen from the subscribed mavros/rc/in topic.
- The same is true for mode changes and arming, so after every command I use a callback to check that the command was successful before continuing the program.
- I also use the VFR_HUD callback in order to check the altitude of the UAV so that it will stop rising once it reaches a certain altitude.

6. Writing your own Programs

- Before finishing your program, make sure to release all RC Override controls before exiting [even during a forced termination]. This will return the full control of the APM back to the RC Transmitter. Do this by publishing 0s to all channels of the mavros/rc/override topic.



```
//Releases RC Override Controls and Exits Program

for(int i=0; i < 8; i++) rc_command.channels[i] = 0;

receiver.rc_check_ch = THROTTLE;
receiver.rc_check_val = RELEASE;
receiver.rc_finished = false;
while((!receiver.rc_finished) && (ros::ok()))
{
    ros::spinOnce();
    rc_message.publish(rc_command);
}
```

6. Writing your own Programs

- Make sure to work through your program one at a time. After finishing one section, compile and check that it works before continuing into another section.
- Comment throughout your code to make it easier to understand, and use ROS_INFO to print out the program status into the command line so that you may test it without using the motors all of the time.
- It is helpful to test your programs with comments (ROS_INFO) before using motors to save time and battery life.
- Make sure that your program ENDS and will not get stuck in a while loop. Always use ros::ok() in while loops so that the Raspberry Pi can end it when shutting down or with CTRL-C.

7. Performing Tests

- Always perform tests without propellers first for safety.
- After you compile your code in your workspace (following tutorial), you should be able to execute it through rosrun.

- It will be helpful to use ROS_INFO functions to be able to see the status of the program through the command window.

7. Performing Tests

- In order to test the program with propellers and in flight, you will have to make your program start without using the command line. This can be done through an external button on the Raspberry Pi, or by having your program start running automatically on boot-up.
- To make it start on boot-up, follow these directions:

http://www.stuffaboutcode.com/2012/06/raspberry-pi-run-program-at-start-up.html

- MAKE SURE YOUR SCRIPT ENDS PROPERLY. You may make your Raspberry Pi **unbootable (break)** if you do not.
- [Now would be a good time to **backup all of your files and data.**]

7. Performing Tests

This is what my script looks like.

You will first need to source your ROS files.

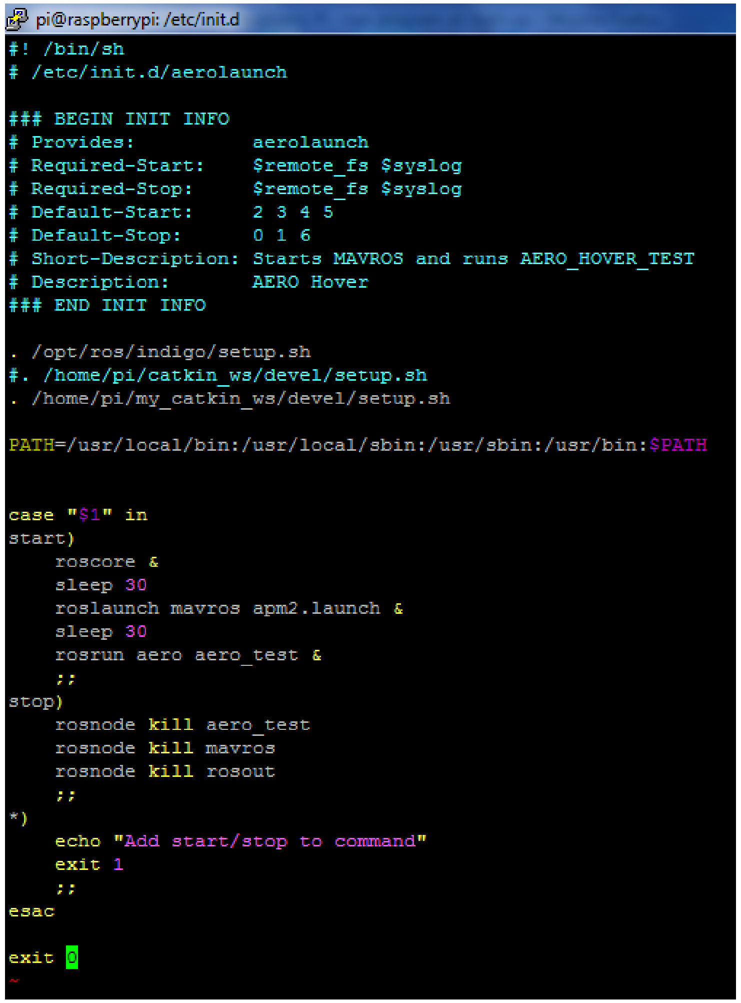
Then make sure to add your bin files into your path file. (This is because of errors with PIP)

Then you will need to have a start) and stop) functions for your boot-up program.

The “&” is for running the program in the **background**, allowing the script to **END**. (really important!)

“sleep 30”(ms) gives the program time to finish.

aero_test is the name of my program.

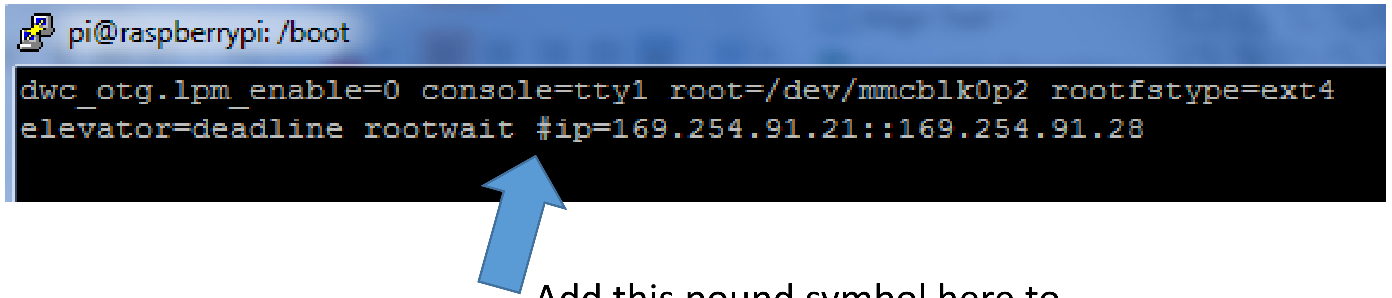


```
#! /bin/sh
# /etc/init.d/aerolaunch

### BEGIN INIT INFO
# Provides:          aerolaunch
# Required-Start:    $remote_fs $syslog
# Required-Stop:     $remote_fs $syslog
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Starts MAVROS and runs AERO_HOVER_TEST
# Description:       AERO Hover
### END INIT INFO

. /opt/ros/indigo/setup.sh
#. /home/pi/catkin_ws/devel/setup.sh
. /home/pi/my_catkin_ws/devel/setup.sh

PATH=/usr/local/bin:/usr/local/sbin:/usr/sbin:/usr/bin:$PATH


case "$1" in
start)
    roscore &
    sleep 30
    roslaunch mavros apm2.launch &
    sleep 30
    rosrun aero aero_test &
    ;;
stop)
    rosnode kill aero_test
    rosnode kill mavros
    rosnode kill rosout
    ;;
*)
    echo "Add start/stop to command"
    exit 1
    ;;
esac

exit 0
~
```

7. Performing Tests

- A good way to test if your program is booting properly is to check through an external monitor (no SSH). This will show you if the program ends, and how long it take for it to boot.
- If it takes too long to boot because it is looking for a network, comment out the ip addresses in /boot/cmdline.txt .

  (This means you can't SSH anymore)



```
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4
elevator=deadline rootwait #ip=169.254.91.21::169.254.91.28
```

Add this pound symbol here to uncomment the ip addresses.

7. Performing Tests

- If everything was done correctly with no problems, you should be able to fly your UAV autonomously with a simple program.
- It takes 2 Minutes from turning on my Raspberry Pi for my program to start running.
- Make sure that your program does not do anything dangerous, and always make sure that your program can be terminated using the RC Transmitter at any time for safety.
- Autonomous Flight is dangerous, so please be safe!

- The Raspberry Pi only copies what a RC Transmitter can do. So properly tune your UAV before flight for better results.

8. Troubleshooting Tips

- If you get an error message, copy and paste the error message into google. You might find a solution.
- If your code does not compile, keep fixing it until it does. Do small checks using comments.
- Programming is not hard, it just takes a long time. Keep googling and looking for a solution. Have patience.
- If you have problems with ROS, look at the ROS wiki page.
- If you have problems with the Raspberry Pi, google the problem. Hundreds of people on earth use the RPi, so someone should be able to find a solution.

End

If you have any questions,

you may contact me at

austinpuk@gmail.com

I am not a professional, but I will try my best to help.

Varanon Austin Pukasamsombut 留学生
Field Robotics Group, Tohoku University
Prof. Yoshida Kazuya, Assoc. Prof. Keiji Nagatani

March 3, 2015