Actuator Control via ROS and Raspberry Pi

1 Introduction

The value of a theoretical work highly depends on the possibility of its physical implementation and any related limitations, or lack thereof. Having developed a flight control law, we desire to develop a flying platform with a system architecture that allows us to implement the control algorithms we have constructed. Having such a system architecture will also allow the application of automated system identification algorithms on the aircraft that houses it as well as provide the aircraft with special commands for future projects or endeavors. For these purposes, we require the final testing platform be capable of piloted and automatic take-offs and landings, manual, pilot-assisted or fully automated flight modes, and enable direct actuator control for either scripted maneuvers or for feedback control.

Off the shelf autopilots are a great start since they already provide a lot of the desired functionality. The Pixhawk, more specifically, has some on-board senors, as well as the capability to interface with other sensors, and can directly command control surface servos and electric propulsion motors. It has multiple flight modes which include manual, semi-automatic or fully automatic modes. The latter can involve position control in the form of waypoint navigation, inertial velocity control, or automated maneuvers such as take-off and landing. The Pixhawk offers multiple layers of fault tolerance and contingency measures in case of failure in hardware or communication. It has been used in our lab for years and has proven to be safe and reliable. The only functionality that it lacks, and arguably the most crucial one for flight experimentation, is direct actuator control. However, it is possible to overcome this problem by connecting the Pixhawk to an onboard companion computer. Using what is called the "offboard" mode, the Pixhawk can receive multiple types of commands, like position, velocity, attitude or actuator setpoints, from a serial connection or over wi-fi. This document describes how commanding actuator control using the Pixhawk and an onboard computer is achieved.

2 Hardware Architecture

The Pixhawk lies at the center of the hardware architecture. It receives manual pilot inputs from a transmitter. It also sends telemetry data and receives mission parameters from a ground control station. Having received piloted inputs and mission parameters, the autopilot takes readings from the sensors and sends pulse width modulated signals to the propeller motor electronic speed controller (ESC) and the servos of the control surfaces. For our setup, to power the aircraft's actuators, a propulsion battery is connected to the propeller motor ESC and an avionics battery is connected to the rails, thus powering both the Pixhawk as well as the servos. The avionics battery also powers the co-computer, which is connected to the the pixhawk through a serial link.

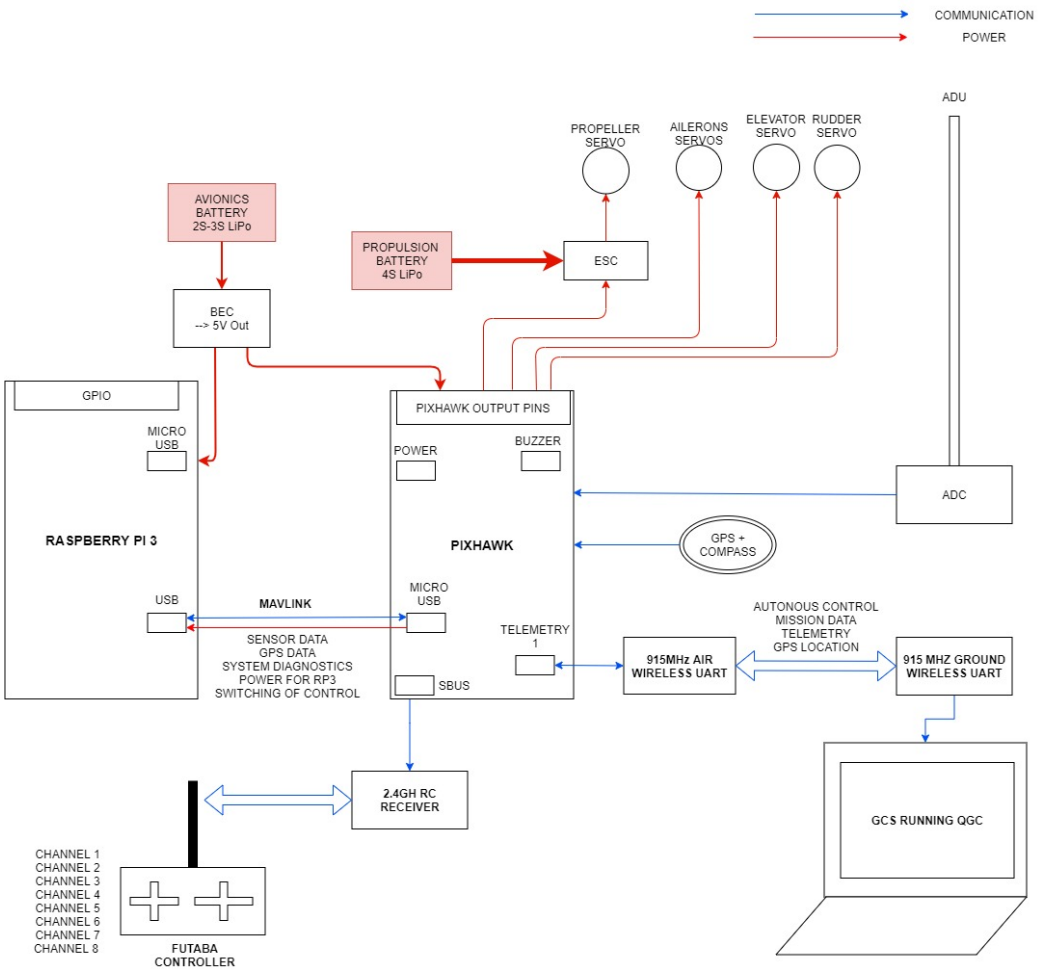

Figure 1: Hardware Architecture

3 Software Architecture

Pixhawk communicates with a ground control station using MavLINK, a protocol for communicating with sUAS. In order to relay and receive information from the autopilot, one requires a programming environment that can support such a communication protocol. For our purposes, we use ROS since it is easy to integrate and offers an easy way to achieve the desired actuation goals. ROS has a massive user community and has been widely utilized for robotics research. It does not require learning a new programming language but takes in already established ones, such as C++ and Python. However, one must understand the architecture of ROS to be able to make use of it.

3.1 Installation of Prerequisites

Since the Raspberry Pi does not come with ROS pre-installed, one would have to manually install ROS after making sure the co-computer has an operating system that supports ROS.

Remark: The installation procedure described below is specifically for the Raspberry Pi 4 with Ubuntu Mate 20.04.2 LTS (Focal Fossa) as an operating system and "Noetic Ninjemys" as the compatible ROS distribution since the versions of the Raspberry Pi, Ubuntu and ROS mentioned are the latest at the time of writing this document. However, any other versions or variations of the hardware or the software should have a similar installation procedure.

3.1.1 Installing Ubuntu

If the Raspberry Pi does not have an operating system which supports ROS, it is imperative to install one that does. We chose to install Ubuntu Mate since it is a popular and well supported operating system that can run ROS as well as other libraries we require on the Raspberry Pi. To install Ubuntu, download the 64 bit Ubunutu Mate port for the Raspberry Pi from the the official website. Since the downloaded file is a compressed file of the image with the extension .xz, it is required to decompress it before mounting it. On Ubuntu, one may use the command in the terminal, which is accessible from the drop down menu in the top left corner or by pressing Ctrl+Alt+T:

```
xz -d ubuntu-mate***.img.xz
```

where the three asterisks denote the version number. On Windows, one may simply use 7-zip to extract the image file.

After decompressing the image file, write the image file on the sd card which came along with the Raspberry Pi using the Raspberrry Pi imager. The process is straight forward. First, select the image file as the operating system. Next, insert the sd card in the computer. Then, select the drive associated with the sd card. Finally, click on the write button. The imager will take a few minutes to finish writing.

Once the writing is done, place the sd card in the Raspberry Pi and power it on using a 5V, 3A minimum power supply to proceed with the Ubuntu installation. The co-computer will need to be connected to a monitor, a keyboard and a mouse. During the installation, the user will configure the settings of Ubuntu like the language, time and date, username and password, and wifi if applicable. The installation wizard will guide the user through the entire process. It may be convenient to set the username and password as the "nsl" and "uav123!" since all others in the lab have the same username and password. This will also prove helpful when transferring scripts from one co-computer to another.

At this point, it may be preferable to connect the Raspberry Pi to the internet, either by Ethernet cable or by WiFi and update Ubuntu. If the process does not start automatically, the user may use the following command

```
sudo apt update
sudo apt upgrade
```

After updating Ubuntu, the operating system is ready for the ROS installation process.

3.1.2 ROS and MAVROS installation

The entire installation process is managed through the Ubuntu terminal, which is accessed through the drop down menu in the top left corner of the screen or by pressing Ctrl+Alt+T. Installing ROS requires an active internet connection and cannot be installed without one. At several stages of the installation, the user may be prompted to accept changes to the disk space to proceed. The user must accept the changes. In other times, the user might encounter an error stating that the command entered requires an installation of a prerequisite and recommend what to command to send to do so. The user is recommended to copy the command and use it before repeating the step that failed. The following website contains most of the installation steps and is more user friendly from copying and pasting perspective. To start the installation process, the user first sets the co-computer up to accept sorftware from the ROS website using the following command:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
```

Note that the above command is a single line of code. The user will be prompted to enter the password and has to do so to proceed. Next, the user has to set his keys up using the following command:

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

If the user has trouble connecting to the server, in the command, they my try using 'hkp://pgp.mit.edu:80' or 'hkp://keyserver.ubuntu.com:80' instead. Alternatively, one may use the curl command. However, before doing so they must install the feature:

```
sudo apt install curl
```

Having installed curl, the user may use the following command to set the keys (note that the command below should be a single line):

```
curl -sSL 'http://keyserver.ubuntu.com/pks/lookup?op=get&search=0xC1CF6E31E6BADE8868B172B4F42ED6FBAB17C654' |
sudo apt-key add -
```

Before beginning the actual installation, the user has to make sure that the package indeces are up to date using

```
sudo apt update
```

If an error appears at this step, a probable cause would be an internet connection issue. Even if the Raspberry Pi is connected to the internet, there might be issues with proxy or firewall settings. This issue is fairly common if connected to the university campus internet. Make sure to resolve this issue and retrying the last step. Having completed the last step, the co-computer is now ready to install ROS. There are multiple versions of ROS, each offering varying functionality. To save space and resources on the Rpi, it is recommended to install the "bare bones" version using the following command:

```
sudo apt install ros-noetic-ros-base
```

Installation will take a while but when it is done, ROS will be finally installed on the Raspberry Pi and should be able to run. However, to do so the following command has to be executed every new terminal before using any ROS related function:

```
source /opt/ros/noetic/setup.bash
```

To automatically source this script every time a new shell(terminal) is launched, the user must the following command one time only:

```
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

and

```
echo "source /opt/ros/noetic/setup.zsh" >> ~/.zshrc
source ~/.zshrc
```

Before the user can use many ROS tools, they will need to initialize rosdep. rosdep enables the easy installation of system dependencies and is required to run some core components in ROS. If the user has not yet installed rosdep, do so as follows:

```
sudo apt install python3-rosdep
```

Note that if the above command did not work, the user should use rosdep2 instead of rosdep. Afterwards, with the following, the user can initialize rosdep:

```
sudo rosdep init
rosdep update
```

This wraps up the bare bones setup for ROS noetic. If for any reason, the user requires a certain ROS package which is not included in the bare bones version, one can always manually install any specific package using

```
sudo apt install ros-noetic-PACKAGE
```

where PACKAGE in the above command would be the name of the package in question. MAVROS does not come with the installation of ROS, regardless of the version and will have to be installed using the steps described earlier. Before doing so, however, one will have to install some prerequisites. They include libraries and tool that ROS relies on to compile and build its workspace:

```
sudo apt-get install ros-noetic-catkin python3-catkin-tools
sudo apt install python3-wstool
sudo apt install python3-catkin-lint python3-pip
pip3 install osrf-pycommon
```

Afterwards, MAVROS is ready to be installed:

```
sudo apt-get install ros-noetic-mavros ros-noetic-mavros-extras
```

Then install GeographicLib datasets:

```
wget https://raw.githubusercontent.com/MavLINK/mavros/master/mavros/scripts/install_geographiclib_datasets.sh
sudo bash ./install_geographiclib_datasets.sh
```

This concludes the installation process for ROS and MAVROS for the co-computer. The Raspberry Pi is now ready to perform its desired tasks.

3.2 Interfacing with the Pixhawk Using the Co-computer

3.2.1 Establising Basic connection with the Pixhawk

Having installed ROS and MAVROS, the user can now interact with the Pixhawk using the Raspberry Pi. To do so, one would have to open a terminal window using Ctrl+Alt+T and type in, assuming the Pixhawk is running PX4:

```
roslaunch mavros px4.launch
```

If the Pixhawk is running Ardupilot, the user will, instead, have to use:

```
roslaunch mavros apm.launch
```

Using one of the commands above will establish a connection between the Pixhawk and the Raspberry Pi via MavLINK and create ROS topics relevant to the data it can receive or send to the Pixhawk. The node launched receives telemetry data from the Pixhawk, much like a GCS would, and publishes the data to its related topic. The node also subscribes to the other nodes it created and passes any messages published in those topics. In some cases, upon using the command above, the user may encounter the following error:

```
[FATAL] [<time.sec>]: FCU:DeviceError:serial:open: Permission denied
```

The error indicates that the node was denied access to the USB ports. To overcome this issue, the user may is the following command:

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

This command would grant access to the USB ports for all users. However, thisis not a permanent solution since the permission override would reset when the Raspberry Pi is rebooted. For a more permanent soluation one would have to use:

```
sudo usermod -a -G dialout <username>
sudo chmod a+rw /dev/ttyACM0
```

The topics shown below are obtained from establishing a connection with PX4. Ardupilot will have similar topics but they may either be named differently. After establishing a connection with the Pixhawk, the user can see a list of all the ROS topics in a new terminal window by hitting Ctrl+Shift+N and typing: If the Pixhawk is running Ardupilot, the user will, instead, have to use:

```
rostopic list
```

The MAVROS topics that appear are:

- /mavlink/from
- /mavlink/to
- /mavros/actuator_control
- /mavros/battery
- /mavros/global_position/compass_hdg
- /mavros/global_position/global
- /mavros/global_position/gp_vel

- /mavros/global_position/local
- /mavros/global_position/rel_alt
- /mavros/gps/fix
- /mavros/gps/gps_vel
- /mavros/imu/atm_pressure
- /mavros/imu/data
- /mavros/imu/data_raw
- /mavros/imu/mag
- /mavros/imu/temperature
- /mavros/local_position/local
- /mavros/mission/waypoints
- /mavros/mocap/pose
- /mavros/px4flow/ground_distance
- /mavros/px4flow/raw/optical_flow_rad
- /mavros/px4flow/temperature
- /mavros/radio_status
- /mavros/rc/in
- /mavros/rc/out
- /mavros/rc/override
- /mavros/safety_area/set
- /mavros/setpoint_accel/accel
- /mavros/setpoint_attitude/att_throttle
- /mavros/setpoint_attitude/cmd_vel
- /mavros/setpoint_position/local
- /mavros/setpoint_velocity/cmd_vel
- /mavros/state
- /mavros/time_reference
- /mavros/vfr_hud
- /mavros/vision_pose/pose

- /mavros/vision_speed/speed_vector
- /mavros/visualization/track_markers
- /mavros/visualization/vehicle_marker
- /mavros/wind_estimation

To display messages published to a topic, one can write:

```
rostopic echo <topic-name>
```

For example, to read RC values, one would write:

```
rostopic echo /mavros/rc/in
```

To write a single message to a topic, the command to be used is:

```
rostopic pub <topic-name> <topic-type> [data...]
```

For example, one could write the following:

```
rostopic pub /mavros/actuator_control mavros_msgs/ActuatorControl  '[1.0, -1.0, -.5, .5, 0.0, 0.0, 0.0, 0.0]'
```

This would issue a one time command to the actuators of the aircraft, which include the servos as well as the propellers. As a result, the aircraft would perform the desired actuation for only a moment and would return back to its actuators' zero set point. To issue a more continuous command, one may use the rate option "-r" to repeat the command. The option "-r" is followed by the desired rate at which the message repeats. If no numeric value is given afterwards, it is set to repeat at 10Hz by default. To repeat the previous message at 25Hz, one may write:

```
rostopic pub -r 25 /mavros/actuator_control mavros_msgs/ActuatorControl  '[1.0, -1.0, -.5, .5, 0.0, 0.0, 0.0, 0.0]'
```

As a result, the aircraft will hold the actuators to the levels imputed. More specifically, the ailerons will deflect to their maximum assigned deflection, forcing the aircraft to roll right, the elevators will deflect to their minimum, causing a nose down maneuver, the rudder would deflect halfway to the left and the propellers would spin at 50% throttle. To provide a more dynamic actuator control, one would have to create a script ROS can execute unless the user can input individual commands fast enough. Other commands for ROS topics that may prove useful include:

- bw <topic-name> : displays bandwidth used by topic.
- delay <topic-name> : displays the delay for topic which has header.
- find <msg-type> : finds topics by type (class/format) of messages
- type <topic-name> : displays type (class/format) of messages
- hz <topic-name> : displays the average rate of a topic over the entire time

3.2.2 Creating Workspace

Before writing a script, the user will have to create a workspace for catkin to build it. The first step is to create the directory, which will include a source (src) folder. For convenience, the workspace will be set in the home directory (i.e. the path from the root directory would be /home/<username>). The command to do so is:

```
mkdir -p ~/<workspace>/src
cd ~/<workspace>/
catkin_make
```

The tilde in the above commands stands for the home directory. The command "makedir" creates new paths/directories in the system if they did not previously exist and the command "cd" changes the current working directory to the target directory. The second line changes the current terminal working directory to the location of the created workspace. The command "catkin_make" creates necessary folders and files. If one looks in the current directory, they would notice there is a "build" and "devel" folder with files inside. Also, there is a "CMakeLists.txt" file in the "src" folder.

Now that the workspace has been created, it is time to create the package. A catkin package needs to meet a couple of bare requirements: it needs to have its own folder and it needs to have one "package.xml" file and one "CMakeLists.txt" file. It is recommended to put the catkin packages in the source (src) folder of the workspace. To do so, the following commands are used in the terminal:

```
cd ~/<workspace>/src
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

The first command above changes the directory to the "src" folder inside the workspace folder, assuming the previous working directory is the workspace folder. The second command creates a package under the name <package_name>. The arguments which follow the package name are the package dependencies. Dependencies are libraries which the package relies on to be compiled and built by ROS. Dependencies are optional and one may add dependencies after creating a package should the need arise. Some common dependencies include "rospy", which allows ROS to inerpret python scripts, "roscpp", which allows ROS to interpret C++ scripts, and "std_msgs" which contains the basic structure of standard ROS message types. For instance, one may create a package called "offboard" which will contain a C++ script using the following command:

```
catkin_create_pkg offboard roscpp std_msgs
```

Creating the package will also create the "package.xml" and "CMakeLists.txt" files. One may edit these two files to change ROS specific parameters related to the package including the dependencies. The package will also have an "src" folder. This folder will serve as the location of the scripts the user will write. The user may input the script either by using the terminal, or by navigate to the folder using the native file explorer and manually create the file using the text editor or copy the script from another source. To use the terminal, one will have to navigate to the "src" folder inside the package and create the script file. For example, assuming the user wants to create a C++ script, they would use:

```
cd <package_name>/src
nano <script>.cpp
```

The last command will open a black script editor in the terminal where the user can write their script. The next section will discuss the general layout of a C++ script in ROS and will layout how to build and run the script

3.2.3 Writing Considerations for a ROS C++ Script

The general sctructure of a C++ script for ROS follows the same outline of a generic C++ code but requires certain commands to interface with ROS. The script begins with file inclusion preprocessor directives, followed by global variable and function definitions if necessary. After that, the main function, which holds the bulk of the code may be written. Instead of spending time on every detail of a typical C++ script, this section will focus on how a ROS specific C++ script is different.

Since the C++ script will require communicate with a MAVROS node, it is necessary to include file headers for message or service structures MAVROS uses. One must include the ROS header as well as a header file for every type of message expected to be used. To do so, one may write:

```
#include <ros/ros.h>
#include<mavros_msgs/<type>.h>
```

where <type> can be replaced by the necessary message type found in the following ROS wiki article. For example, if the user requires to receive RC signal values, they will have to use:

```
#include <ros/ros.h>
#include<mavros_msgs/RCIn.h>
```

Before moving on to the main function, one must write callback functions which will allow storing sensor values from ROS topics. Callback functions are functions which are not used in the script itself but are passed as arguments in other functions and are used by the system. The way to do so is by defining a global variable of the right message type to store the data, and then writing the finction as such:

```
mavros_msgs::<type> <variable>;
void <callback_function>(const mavros_msgs::<type>::ConstPtr& msg){
    <variable> = *msg;
}
```

The first line above defines the global variable of the proper message type. The second line defines the callback function as a void, meaning that we do not expect the function itself to return any value, and declares the type of argument it takes. The ampersand indicates that the function does not take the value of the message the argument "msg" but it points to its location in memory. The "ConstPtr" is a class template inside the message type that stores a pointer to a dynamically allocated object which, in this case, would be the argument of the function. This would tag the complex pointer to the right message type. The third line assigns the global variable to the values of the message. The asterisk calls the value of the message instead of the pointer "msg". A callback function declaration to read the state of the Pixhawk (i.e. whether it is connected or armed and what mode it is in) can look something like:

```
mavros_msgs::State current_state;
void state_cb(const mavros_msgs::State::ConstPtr& msg){
    current_state = *msg;
}
```

Sometimes, especially if the ROS rate is high enough, segmentation errors may occur. Segmentation faults happen when a program tries to read from memory which no longer exists or for which it doesn't have permissions. It is helpful to include a boolean to indicate that the the call back function has successfully updated its respective global variable. The boolean would be set to true before the termination of the callback function. Whenever the code needs to use the global variable in question in the main function, the user will have to check whether the boolean is true. After using the variable, the boolean must be set to zero. The following is an example of how to do so for an RCin callback function:

```
bool RCIn_flag = false;
mavros_msgs::RCIn rc_input;
void rcin_cb(const mavros_msgs::RCIn::ConstPtr& msg){
        rc_input = *msg;
         RCIn_flag = true;}
```

After having defined all necessary callback functions, one may proceed with writing the main function. While the main function of a generic C++ script does not take any arguments, a ROS C++ script's main function takes in two arguments, an integer "argc" and a character "argv", as such:

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

The two arguments are used in the first line of the main function to initialize the roscpp node are used in a function in the first line of the main function to initialize the ROS node:

```
ros::init(argc, argv, "<node_name>");
```

"ros::init" uses "argc" and "argv" to parse remapping arguments, a feature of ROS that allows launching the same nodes under different configurations from the command lines. <node_name> is the name assigned to the node by default. It cam be overridden by the remapping arguments. Node names must be unique across all ROS systems.

After initializing the node, the script will need to start the roscpp node. A common way of doing so is by creating a handle for the node:

```
ros::NodeHandle <nh>;
```

When the node handle is created, it will call the "ros::start()" function. In the same manner, the "ros::shutdown()" will be called when the node handle is destroyed or the program exits. The node handle also gives access to subscribing to or publishing in nodes. To subscribe to a node, the following command must issued:

```
ros::Subscriber <subscription> = <nh>.subscribe<mavros_msgs::<type>>("<topic-name>",
↪  <buffer>, <callback_function>);
```

In the above line, <subscription> is a ROS subscriber variable, <nh> is the node handle variable, <type> is the type of MAVROS message, same as one of the types included at the beginning of the script, <topic-name> is the name of the topic to be subscribed to, <buffer> is the size of the queue of messages to keep from the topic, and <callback_function> is a callback function defined earlier.

The command to request publishing access to a ROS topic follows along the same guideline. The most noticeable difference, besides the different variable declaration and the node handle function used, is that there is no callback function:

```
ros::Publisher <publisher> = <nh>.advertise<mavros_msgs::<type>>("<topic-name>",
↪  <buffer>);
```

After requesting access to subscriber and publisher privileges, the ROS rate, the nominal rate at which the program will loop. In the following case, <rate> is the desired loop rate in Hertz:

```
ros::Rate rate(<rate>);
```

To make use of subscriber and publisher access, the program will need to prompt ROS to "spin". Spinning allows ROS to process network management/scheduling threads behind the scene. It also allows the callback functions defined earlier to be called. One the most common ways to spin is to use the "ros::spin()" function. However, the function will not stop from executing until the node has been stopped or interrupted. Instead, the following is used:

```
ros::spinOnce();
```

To run the script continuously, a while loop with an adequate termination condition is required. The loop needs to run continuously until prompted to stop. To that end, it is recommended to use a while loop with the function "ros::ok()" as an argument. "ros::ok()" returns false when the node has finished shutting down, from either an error or a user input. At the end of the loop, the function "rate.sleep()" is invoked so that ROS waits the appropriate time to maintain the desired rate. It is also advisable to call the "ros.spinOnce()" function before closing the loop so that the messages from MAVROS are updated.

One of the main objectives of the C++ script is to publish actuator commands in the corresponding topic. To that end, a variable of the correct message type needs to be created. In the while loop, the variable is assigned values corresponding to desired mission parameters. Once the variable is assigned in the loop, it can be published using the publisher handle defined earlier. The overall structure of the while loop should look like:

```
mavros_msgs::<type> <publish_var>;
while(ros::ok()){

    //This is where the user writes the scripted tasks for the while loop...
    <publisher>.publish(<publish_var>);

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

For example, the following script publishes the same message as the last example in section 3.2.1:

```
#include <ros/ros.h>
#include <mavros_msgs/ActuatorControl.h>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "offb_node");
    ros::NodeHandle nh;
    ros::Publisher actuator_pub =
    ↪  nh.advertise<mavros_msgs::ActuatorControl>("mavros/actuator_control", 100);
    ros::Rate rate(25.0);
    mavros_msgs::ActuatorControl act_con;
    while(ros::ok()){
        act_con.controls[0] = 1.0;
        act_con.controls[1] = -1.0;
        act_con.controls[2] = -.5;
        act_con.controls[3] = .5;
        actuator_pub.publish(act_con);
        ros::spinOnce();
        rate.sleep();
    }
    return 0;
}
```

This concludes the major considerations to write a roscpp script. There are some minor elements that need to be considered as well. These elements are not necessary for a script to run but might be helpful to the user under some applications. To begin with, the script may involve applications which require time or duration. ROS has the ability to setup a simulated clock for nodes which works seamlessly to access current time using the classes "ros::time" and "ros::duration". "ros::time" is used for a specific date and time (e.g. right now or 5pm January 5th, 1982...) whereas "ros::duration" is used to specify periods of time. One should be wary of arithmetic between the two classes:

- duration + duration = duration
- duration - duration = duration
- time ± duration = time
- time - time = duration
- time + time = undefined

For example, if the script requires you to obtain the current time, the following can be used:

```
ros:Time current_time = ros::Time::now();
```

Note that the example above give the current time in the message format ROS uses. One may obtain the current time in seconds as a double (number) using the "tosec()" function. This funcion works on both time and duration classes as such:

```
double t = ros::Time::now().toSec();
```

Another consideration to keep in mind is, when referencing files other than the main script, there is no relative path. All directories must written as root directories. For example, a ROS script which opens a "data.csv" file in the same directory and reads the first line and outputs it is:

```
#include <ros/ros.h>
#include <fstream>
#include <string>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "offb_node");
    ros::NodeHandle nh;
    ros::Rate rate(25.0);
    ifstream doc;
    doc.open("/home/<username>/<workspace>/src/<package_name>/src/data.csv");
    string line;
    getline(doc, line);
    ROS_INFO(line);
    return 0;
}
```

Now that the script is written, it is time to build it and run it.

3.2.4 Running a ROS C++ Script

This section picks up form section 3.2.2 and assumes the the script is ready. Now that the package has been created, the workspace will have to be rebuilt. Before doing so, however, one must make sure that the "CMakeLists.txt" file is properly configured to build an executable for the script. To ensure that ROS creates a node for the script, the following lines must be present in the text file:

```
add_executable(<node> src/<script>.cpp)
add_dependencies(<node> ${<node>_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(<node>
    ${catkin_LIBRARIES}
)
```

In the above, <node> is the desired name of the node based on the script, <script>.cpp. The package should be ready to be built and sourced. To do so, the command "catkin_make is used. However, since the command has already been used to set the workspace up, it will detect that build files already exist and will not go through with rebuilding the workspace. To force ROS into rebuilding the workspace, the following must be used (note that this must be done in the workspace directory):

```
cd ~/<workspace>
catkin_make --force-cmake
```

If, for whatever reason, ROS does not rebuild the package(s), which is common if no new nodes or packages have been added but some have been edited, it is possible to delete the "build" and "devel" folders and use the command again.

Before running the script, the "setup.bash" file in the "devel" folder must be sourced. Sourcing a bash file is analogous to typing the commands in the bash file like one would in a terminal. In the case of the file in question, it will add the location of the workspace to ROS as well as setting environmental variables up to allow ROS to function. This is done using the following command:

```
source devel/setup.bash
```

Running any ROS node requires the ROS master to be running. This is achieved by using the command "roscore" in a terminal. Once the command is entered, no new commands can be entered in that terminal until "roscore" is shutdown using CTRL+C. In a new terminal window, one can run the desired node using the command:

```
rosrun <package_name> <node>
```

For example, if one wishes to run the offboard example, which requires a mavros connection to the Pixhawk, from the previous section, the following commands need to be used, each in a separate terminal window:

```
roscore
roslaunch mavros px4.launch
rosrun offboard offb_node
```

To avoid opening multiple terminal windows and to run the node using a single command, one may use the "roslaunch" command. "Roslaunch" automatically starts "roscore" if it detects it is not running already and allows running/launching other nodes. Doing so, however, would require creating a launch file for the node in question. For proper file management, it is recommended to write the launch file in a folder called "lauch" inside the package folder. The lauch file can be created like any text file but should have the extension ".launch". To create a launch file, "<launch>.launch", which also launches the mavros node which communicates with the Pixhawk, the following script must be contained within the file:

```
<launch>
    <include file="$(find mavros)/launch/px4.launch" />
    <node name="<package_name>" pkg="<package_name>" type="<node>" />
</launch>
```

It is also possible to set launch parameters or options for other launch files if they allow it. For example, a launch file to run the same node as before while having a connection with the Pixhawk with a baud rate of 9216000 would be:

```
<launch>
    <include file="$(find mavros)/launch/px4.launch" >
        <arg name="fcu_url" value="/dev/ttyACM0:9216000" />
    </include>
    <node name="<package_name>" pkg="<package_name>" type="<node>" />
</launch>
```

In the above example, "fcu_url" is the name of the argument/option for the px4 launch file and "/dev/ttyACM0:9216000" is the value assigned to it. "/dev/ttyACM0" is the location of the USB port in Ubuntu.

After creating the launch file, the workspace should be rebuilt and the "setup.bash" file needs to be resourced per the commands given earlier in the section. To launch the file, the following command is used:

```
roslaunch <package_name> <launch>.launch
```

3.2.5 Automatic Startup script

Now that the launch file has been configured and the "roslaunch" option works, it may be convenient to have the Raspberrry Pi start a script automatically so that one does not have to interface with the device after every power up to run the script. Rather, the script would automatically start up as soon as the device fully boots up. To begin the process, the user will have to write a shell file. The shell file will include the same command one uses in the terminal. Note that this section assumes that he workspace has already been built using the "catkin_make" command. For the user's convenience, the shell file will be placed in the home directory. The user will have to create the shell file by creating a text file and calling it "<ROS_startup>.sh". In the file, the user should write:

```
source /home/<username>/<workspace>/devel/setup.bash
roslaunch <package> <node_launch>.launch
```

After completing the shell file, it is time to create the service file. To do so, one will have to open the terminal and type the following commands:

```
cd /etc/systemd/system
sudo nano ROS_startup.service
```

Inside the service file,the following must be written:

```
[Unit]
Description = ROS_startup
After = remote-fs.target
After = syslog.target
[Service]
ExecStart = /bin/bash /home/<username>/ROS_startup.sh
Restart = on-abort
[Install]
WantedBy = multi-user.target
```

Once the service file is created and written, the "systemd" process, a software suite which provides an array of system components, needs to be reset to recognize the new service file:

```
sudo systemctl daemon-reload
```

To automatically start the ROS startup service, the user must write:

```
sudo systemctl enable ROS_startup.service
```

After doing so, the user will have successfully enabled the automatic start up of the intended node on boot up and may reboot the device to verify that the process works. Other "systemd" commands which the user may find useful are:

- start: runs a service once
- stop: stops a running service
- status: outputs execution status of a service
- disable: removes a service from automatically starting on boot up

For example, one may write:

```
sudo systemctl status ROS_startup.service
```