Outdoor Experiment

Chih-Chun Chen

January 14, 2023

Contents

1 Terminology and Online Sources

1.1 Terms

- **Pixhawk** is an autopilot/hardware/flight control unit (FCU).
- **PX4** is an open source flight control software for drones and is used on Pixhawk.
- **QGroundControl**
- **MAVLink** is a very lightweight messaging protocol that has been designed for the drone ecosystem.
- **MAVROS** is the MAVLink extendable communication node for Robot Operating System (ROS)
- **Dialects** are XML files that define protocol- and vendor-specific messages, enums and commands.
- **Sik** is a collection of firmware and tools for radios.

1.2 Online Sources

- Check here for toolchain installation, Pixhawk/NuttX development toolchain is necessary for building PX4/Firmware.
- Check here for installation and building of PX4/Firmware.
- PX4 architecture and offboard communication architecture

2 Equipment

This section shows some of the hardware and equipment, includes vehicle airframe, radio devices, and on board computer, used.

2.1 Dji F450 Quadrotor

The airframe used in the experiment is Dji F450. Figure 1 shows the Dji450 Quadrotor, and Label 1 is the global positioning system (GPS) module, Label 2 is the battery which weights 430 grams, Label 3 is the Pixhawk 1 flight control unit (FCU), and Label 4 is the radio telemetry. The takeoff weight, include airframe, battery (430 g), FCU, and radio telemetry, is approximately 1.2 kg. Additional weight is Odroid-xu4 and its case, which is 70g. Diagonal propeller tip-to-tip distance is 0.65 m.

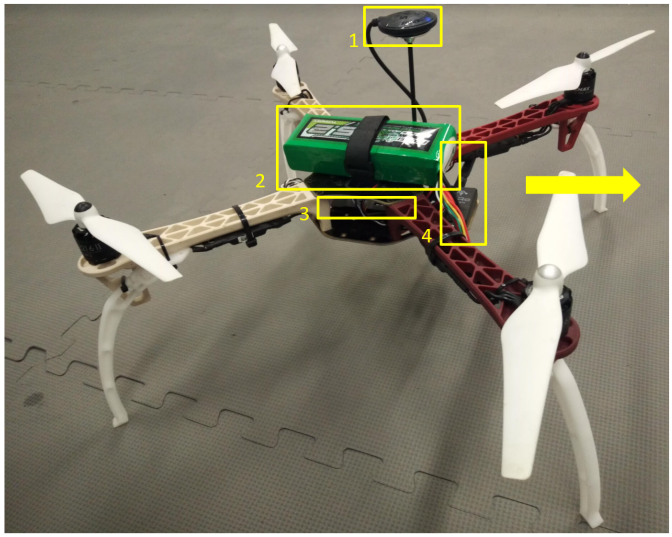


Figure 1: Dji F450 airframe

2.2 Pixhawk

The Pixhawk used in Pixhawk 1 (Figure 3).Label 1 connects to the RC receiver, Label 2 connects to 4 electric speed controllers (ESCs) and motors, Label 3 connects to SiK telemetry radio, Label 4 connects to power/battery, Label 5 connects to GPS, and the TELEM 2 (not labelled in yellow but labelled 2 in blue) connects to the on-board computer.

2.2.1 Parameters

- Since my experimental system does not have RC, there are two parameters required to change for **offboard** to be switchable/callable (usually, offboard is switched by using the RC switch). Parameters: **COM_RC_IN_MODE** and **NAV_RCL_ACT**.

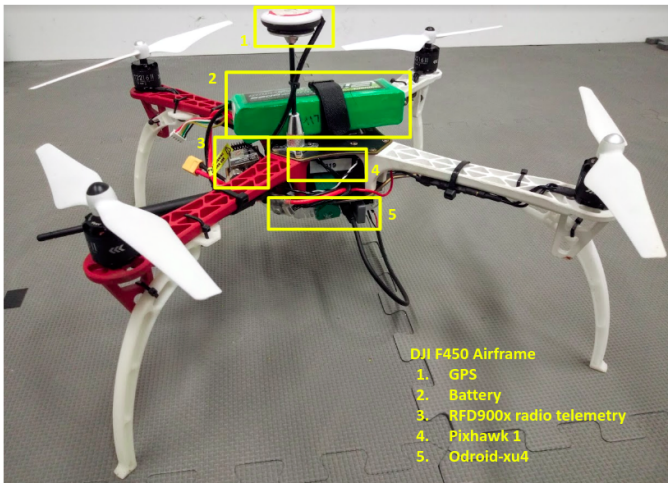


Figure 2: Dji F450 airframe

- When TAKEOFF mode is called and the vehicle reaches 2.5m, somehow the system might switch to AUTO.RTL mode. So, before flying, change the **RTL setting** (altitude and mode) from "QGC → Gear → Safety".
- Multi-agents: **SYS_ID** can be set from QGC → Vehicle Setup → Parameters → MAV_SYS_ID. Read the sysid from rostopic /mavlink/from.
- Setup the **Geofence** for safety.
  - Instruction and reference: here and here.
  - Open QGroundControl and connect pixahwk. From **Gear icon** → **Safety** → **Geofence Failsafe**, set breach action, maximum radius and altitude.
  - For testing, set breach action to **Return**, radius to **2m**.
  - Remove propellers and connect UAV to QGC via usb wire or radio. Turn on RC. QGC shall report vehicle in Manual Mode. Plug in battery and power on the UAV. Arm the UAV from RC switch, and push the throttle a bit to prevent auto disarm. QGC shall report **Armed**. Hold the laptop, vehicle, and RC, and walk!
  - Walk for 2m or more due to uncertainty, at some point, QGC shall report UAV mode switched to **Return to launch (RLT)**. This means the geofence is breached and geofence testing succeed.
  - Disarm UAV from QGC or switch the mode back to manual from QGC and disarm via RC.

2.2.2 RC Transmitter and Receiver

Transmitter-receiver binding (how transmitter connect to receiver so that they can talk to each other):

1. Push the receiver binding button and plug it in to the pixkhawk (left most three)(vertically, black wire facing up).
2. The light on the receiver shall be red and green.
3. Turn on the transmitter, press menu then page, find "bind" (shall be listed in the very button of the page).
4. Press "bind", the green and red lights on the receiver shall start to flash (it means that the

Figure 3: Pixhawk 1

receiver has received the binding command).

5. Restart!! Turn off the transmitter and unplug the receiver. Turn on the transmitter and plug on the receiver.
6. Binding process is completed! (the receiver shall flash green light).

QGroundControl and transmitter:

1. First things first, take down the propellers (safety reason).
2. Go to the firmware page.
3. Plug in the pixhawk, and click on **OK** for updating the firmware (be sure to click fast since it will be refreshed very soon).
4. Turn on the transmitter and calibrate the radio (follow the calibration instruction).
5. Check on the channel that you want for arm by flapping the switch, the channel column on the right of the page shall show you which channel it is. Go to "Parameter" page, and change the RC_ARM_MAP_SW to the right channel.
6. Need to re calibrate the accelerometer (if there is error, even the error does not specified it as an accelerometer issue).
7. CBRK_IO_SAFETY, CBRK_USB_CHK (Set these two value to the one specified on the page, this allows for arming without usb disconnect), and go to Mavlink Console (this is in the fourth icon, the one that looks like a file, on the top of the page), type **reboot** to restart and ARM (flap the transmitter switch) without the usb disconnected.

2.3 Radio Telemetry

2.3.1 Usual SiK telemetry radio

SiK telemetry radio introduction click here or here.

- Github Sik firmware page click here.

- 3DR (300m range, 250 kbit/sec, point-point communication) v.s. RDF900 (900Mhz, 15km range, 750 kbit/sec, multi-point communication).
- Three types of networks: simple pair (peer to peer), asynchronous mesh network, and multipoint network. Multipoint is what is needed.

SiK Telemetry Radio binding/connection steps (Sik Radio Intro, Setting & Config.):

1. Download and open **Mission Planner**
2. Go to **Setup** → **Optional Hardware** → **SiK Radio** page
3. Select the correct COM port and set the baud rate to 57600. Don't need to press "Connect" button here, so the "Connect" button shall be in a disconnected state. All these shall be in the top right corner of gui.
4. Press **Load Setting**. Left box shall show the information and data.
5. Select **Net id** (default is 25). Radio hardware having the same Net id can communicate with each other.
6. Press **Copy to required remote** and **Save Settings**.
7. Plug in another radio and follow the same steps.
8. Plug in one device in computer and another to Pixhawk **TELEM 1**. Then Load Setting again, both right and left sides shall show data.
9. Press top right corner "Connect" button. Data of pixhawk shall show up.

2.3.2 RFD900x

The radio telemetry chosen to be used in the experiment is RFD900x modem (Figure 4). RFD900x can transmit signals from at least a range of 15 km and has the data transfer rate of 750 kbit/sec, and more importantly, it has the point-to-multipoint communication functionality. Point-to-multipoint communication is necessary in the experiment since the single ground control station needs to control and communication with multiple vehicles.

Figure 4: RFD900x model

RFD900x files, documentations, and setup:

- See here for all RFD radio files.
- See here for radio modem **data sheet** and here for **RFD900x data sheet**. The FAQ section is quite useful.
- See here for **software manual**.
- See here for **multipoint firmware** user manual.
- See here for **wire connection** between rfd900 and pixhawk telem1.

- See here for **modem tool user manual**, which is the gui designed by RFD for uploading firmware and setting configuration.
- See here for all the **configurations and parameters**.

RFD900x setup procedures:

1. Download the lastest firmware for RFD900x multipoint SiK from here and download the modem tools from here (Windows only).
2. Connect radio with computer via FTDI cable.
3. Setup the parameters (remember to copy to remote and save)
   - Common ones: Baud (57600), air speed (64), net id (0, has to be 0 if using multipoint firmware.)
   - Antenna Mode: 1, 2, or 1 and 2. (Choose which antenna is used. Recommend use two antennas.)
   - Node ID (1 to 16). For master/based node, the node connect to gcs, the node id shall be 1.
   - Dest ID (= NODEDESTINATION, 1 to 16 and 255). Set this number to the node that you want to broadcast to (e.g. DestID=1 to broadcast to the master, and this value cannot be the same as the Node ID of its own). Set it to 255 (or 65535) to broadcast to all nodes.
   - NETCOUNT (1) is the total number of networks on the one master node. Note, it is not he number of nodes.
4. Go to terminal tab and type **AT&M0=0,z**, where z is the total number of node/radio going to use. Save (AT&W). Reboot (ATZ).

2.4 Ardupilot (software)

Ardupilot is a software/firmware that has a better gps/estimator than PX4. Proved by experiment.

- Use qgroundcontrol to upload ardupilot firmware (ChibiOS, Multi-rotor/copter). Calibrate the sensors and the ESC (use the RC ARM switch to replace the safety switch if there isn't one).
- Change $SERIAL2\_PROTOCOL$ = MAVLINK2 (mavlink2 is preferred, but may not work with Pixhawk1)
- Fly manually with the Position Hold, Autotune if it isn't stable. (For my case, autotune is not needed.)
- Need to manual fly to autotune the drone system controller.
- Steps for autotune. (If the vehicle is flying fairly well with pos hold (i.e. on drifting when hovering), no need for autotune.)
  1. Set RC[6-10] OPTION to 17
  2. Arm. Switch to Alt Hold.
  3. Fly up to an altitude.
  4. Switch to Autotune and fly manually.

5. After the vehicle stop shaking, it means the autotune is good.
6. Land and disarm with the autotune switch ON to save the data (PID gains).
7. To test for the new PID gains, switch off the autotune

- Ardupilot code and Gazebo simulation
  1. Setup environment and download code.
  2. Setup gazebo silt.
  3. Main difference: OFFBOARD to GUIDED. Rosservice for land and takeoff.
- Somehow when mavros is launched, the `/mavros/state` shows connected, but other ros-topics are empty. The following command can fix this:
  `rosservice call /mavros/set_stream_rate "{message_rate: 10, on_off: 1}"`

2.5 Odroid-XU4 On-board Computer

The Odroid-XU4 showed in Figure 5 is utilized as the on-board computer in the experiment. Each Odroid-XU4 has ubuntu 18.04 as well as all the software and packages required installed. A Wi-Fi module is connected to the Odroid-XU4 for the Odroid-XU4 to communicate with the ground control station via 5GHz local Wi-Fi network.

Figure 5: Odroid-XU4

See 4.5.2 for more information on the Wi-Fi communication. See ODroid wiki, Communicating with ODroid via MAVLink (Ardupilot) for some information regarding ODroid. The following shows the setup steps for brand-new Odroid-XU4 to communicate with computer/laptop with ROS/MAVROS.

1. **Get eMMC or micro SD card**
   Open the plastic protector and detach the eMMC (colored red and has blue dot and number 32 on it) or micro SD card. Connect the eMMC onto the eMMC reader or micro SD card to USB reader.
2. **Download ubuntu onto eMMC or micro SD card**
   Download ubuntu. Choose "Download from **main** server" → "XU4" → "**mate**-odroid-xu4-20190929.img.xz" (this file is about 1GB large). Download etcher (the desktop that optitrack uses shall have this installed). Plug in eMMC/eMMC reader or micro SD card. Open Ether. Choose the downloaded ubuntu file and the USB. Flash! Reference & Details.

3. **Some Setup Before SSH**
   Attach eMMC or micro SD card back to ODroid (remember to change the "Boot Select Switch" on board). Connect Wifi module, monitor, keyboard, and mouse to ODroid. Connect power/battery to ODroid, then it will turn on. The color of desktop shall be green, and the password is **odroid**. Setup the **internet** (DroneNet. Password: fsclab_hhtliu). From the right top "Gear → System Settings → Login Window", **enable automatic/manual login**, otherwise laptop would be be able to ssh to the ODroid. In terminal, type **ifconfig** to get IP address, in this case, it is 192.168.2.117. Note: can't install ros with monitor connected (issue: lockdown front-end...).
4. **SSH**
   From laptop, connect to the same wifi as ODroid, open terminal, and type:
   ssh -X odroid@192.168.2.117
   If successfully connected, it shall show *Welcome to Ubuntu 18.04.3 LTS ...* and the name/title shall change to *odroid@odroid*. It may ask for password, which shall be **odroid**.

   Some shortcut:
   In bashrc (gedit ∼/.bashrc in terminal), write:
   alias odroid="ssh -X odroid@192.168.2.117"
   Open a new terminal and type **odroid**, and it shall try to ssh to odroid. Follow this link to ssh login without password.

   For **Multi-computer**:
   In bashrc, write, for example
   export ROS_MASTER_URI=http://192.168.2.117:11311
   export ROS_IP=192.168.2.xxx
   In this example, these are the lines I will write in my laptop's, has ip address of 192.168.2.xxx, bashrc because I plan to use odroid as master computer in the experiment. And ROS_IP will be the ip address of the computer where the bashrc file is in. So for odroid, both ROS_MASTER_URI and ROS_IP shall be 192.168.2.117. In all, ROS_MASTER_URI will be the ip address of the master computer (http and :11311 is necessary), and ROS_IP is the ip address of the current computer. It is recommended that the master run on onboard computer because if the laptop is the master and crashed during the experiment, ros on onboard computer will also crash and then the drone will crash.
   Testing and Steps: (1) Has to run **roscore** on master laptop. (2) px4.launch on odroid. (3) rostopic list on master laptop and whatever mavros rostopic shall show up.
   Name for each computer:
   In terminal the name shown shall be in the type of a@A. To change A, go to right top "Gear → Network → Host Settings" and change host name.
5. **Install ROS, MAVLINK, and MAVROS**
   Follow the usual steps to install ROS, MAVLINK, and MAVROS. Just keep in mind that, install **ROS-Base** instead of Desktop-Full. Also, see Section 4.1 to install mavlink and mavros and create catkin workspace. I had error of catkin: command not found while I tried to catkin build the newly created workspace, sudo apt-get install python-catkin-tools shall solve the problem. Download MAVLINK and MAVROS in the workspace and catkin build -j2 (j2 is highly recommended). Good luck!! I got a lot of weird problems or errors when following the installing steps. Google them! When I tried catkin build (last step of MAVROS installation), I got error of c++: internal compiler error: Killed (program cc1plus), I created a swap file to solve this.

6. **Connect ODroid to Pixhawk: USB to Telem2**
   In odroid, cd /dev, and check if there is ttyUSB0 or it might be other ttyUSB name. I have never experience case that Pixhawk is not connected via /dev/ttyUSB0. Then,
   sudo chmod 777 /dev/ttyUSB0
   Check the SER_TEL2_BUAD pixhawk parameter by QGC. Remember the value, which will be used when calling px4.launch file. If this parameter name cannot be found, check MAV_1_CONFIG (or MAV_2_CONFIG) and make sure that TELEM2 is assigned. Reboot and SER_TEL2_BUAD shall appear. For my case, SER_TEL2_BUAD=921600, so ...
   roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0:921600
7. **Testing**
   Try **rostopic list** and **echo** on the master computer, see if it is properly receiving signals from the pixhawk.
   I got am error while trying to get /mavros/state and it looked something like: Client [...] wants topic /mavros/state to have datatype/md5sum [mavros_msgs/State/ce783f756cab1193cb71ba9e90fece50], but our version has [mavros_msgs/State/65cd0a9fff993b062b91e354554ec7e9]. Dropping connection. This is a version problem. Reinstall the mavlink and mavros of the computer that has old version and then problem shall be fixed (another easier way to solve this would be copy and paste whatever message reported, e.g. /mavros/state in this case, but this would mess up the package/github).

To send multiple command from the ground control station laptop to the onboard computer, **tmux** is used. A shell script shown in Section 7.4.

3 Control System

PX4 and Ardupilot have their native onboard controllers to convert the desired positions to the desired angles. However, instead of using the PX4 native position control, a cascade PI controller named PX4 Command [1] is used to yield desired angles and thrust. The cascade PI controller acts as an outer loop controller then passes its output to the Pixhawk inner loop attitude control by using the `/mavros/setpoint_raw/attitude` rostopic. The PX4 native position control is waived because the origin of the actual position is unchangeable. This means that when the world frame is altered, its actual position will follow the old world frame which has the origin at the location where it is powered on. Both the PX4 and Ardupilot have this problem. Similar to the GUI ground control station, this cascade PI controller uses MAVROS for communication. Hence, the cascade PI controller is compatible with the Gazebo simulator with the PX4-SITL package. The position controller is run on a new node on the computer and the other controller and dynamics are run in the PX4-SITL package. The control architecture of the simulation and experiment is shown in Figure 6.

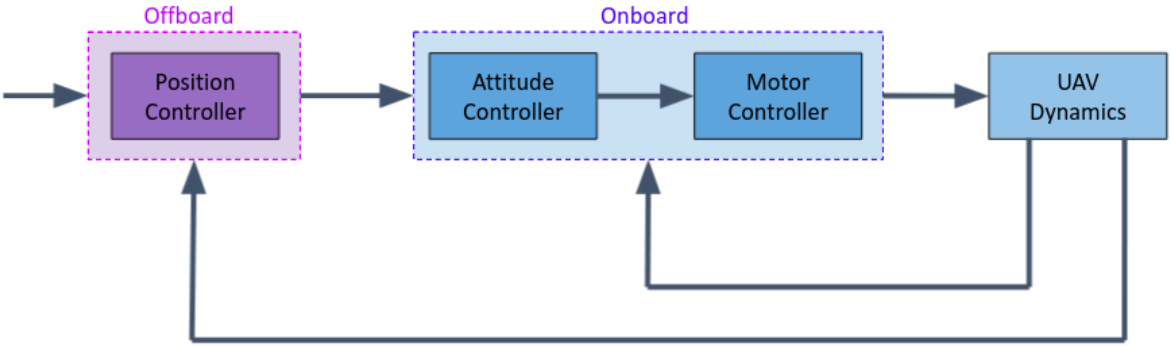


Figure 6: Control architecture

The inputs for the position controller are positions ($\mathbf{x}$) and velocities ($\mathbf{v}$), and the outputs are pitch (θ) and roll (ϕ) angles and thrust force with the gravity considered (T). The controller output can be denoted as $\mathbf{u} = [\theta, \phi, T]^T$. Figure 7 shows the block diagram of the cascade PI controller, and the mathematics expressions are as follows,

$$\mathbf{e} = \mathbf{K_{P_x}}\mathbf{x} + \mathbf{v} \tag{1}$$

$$\mathbf{u} = \begin{bmatrix} \theta \\ \phi \\ T \end{bmatrix} = -\mathbf{K_{P_v}}\mathbf{e} + -\mathbf{K_{I_v}} \int \mathbf{e}dt \tag{2}$$

$$= -\mathbf{K_{P_x}}\mathbf{K_{I_v}} \int \mathbf{x}dt - (\mathbf{K_{P_x}}\mathbf{K_{P_v}} + \mathbf{K_{I_v}})\mathbf{x} - \mathbf{K_{p_v}}\mathbf{v} \tag{3}$$

where $\mathbf{K_{P_x}}, \mathbf{K_{P_v}}, \mathbf{K_{I_v}}$ are the gains matrices and their values are tuned in the simulation and the experiment and denoted as the following,

[1] PX4 Command. https://github.com/LonghaoQian/px4_command. (accessed 2021)

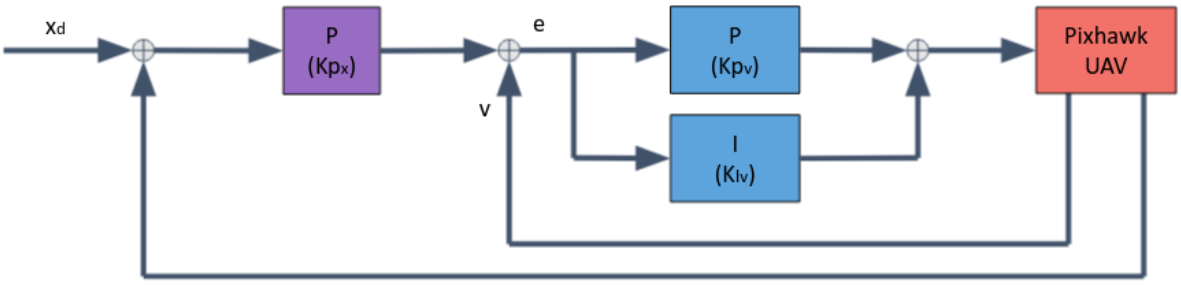


Figure 7: Cascade PI controller block diagram

$$\mathbf{K_{P_x}} = \begin{bmatrix} K_{P_x,x} & 0 & 0 \\ 0 & K_{P_x,y} & 0 \\ 0 & 0 & K_{P_x,z} \end{bmatrix} = \begin{bmatrix} 0.95 & 0 & 0 \\ 0 & 0.95 & 0 \\ 0 & 0 & 1.0 \end{bmatrix} \tag{4}$$

$$\mathbf{K_{P_v}} = \begin{bmatrix} K_{P_v,x} & 0 & 0 \\ 0 & K_{P_v,y} & 0 \\ 0 & 0 & K_{P_v,z} \end{bmatrix} = \begin{bmatrix} 0.08 & 0 & 0 \\ 0 & 0.08 & 0 \\ 0 & 0 & 0.3 \end{bmatrix} \tag{5}$$

$$\mathbf{K_{I_v}} = \begin{bmatrix} K_{I_v,x} & 0 & 0 \\ 0 & K_{I_v,y} & 0 \\ 0 & 0 & K_{I_v,z} \end{bmatrix} = \begin{bmatrix} 0.02 & 0 & 0 \\ 0 & 0.02 & 0 \\ 0 & 0 & 0.02 \end{bmatrix} \tag{6}$$

Below shows the stability proof of this position controller. Let $\mathbf{u'}$ be the acceleration ($\ddot{\mathbf{x}}$) and expressed as follows,

$$\mathbf{u'} = \ddot{\mathbf{x}} \tag{7}$$

$$= -\mathbf{K'_{P_v}}\mathbf{v} - \mathbf{K'_{P_v}}\mathbf{e'} - \mathbf{K'_{I_v}} \int \mathbf{e'}dt \tag{8}$$

$$= -(\mathbf{K'_{P_v}}\mathbf{K'_{I_v}}) \int \mathbf{x}dt - (\mathbf{K'_{P_x}}\mathbf{K'_{P_v}} + \mathbf{K'_{I_v}})\mathbf{x} - (\mathbf{K'_{P_x}} + \mathbf{K'_{P_v}})\mathbf{v} \tag{9}$$

$$\approx \begin{bmatrix} g\theta \\ -g\phi \\ T \end{bmatrix} = \begin{bmatrix} g & 0 & 0 \\ 0 & -g & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \theta \\ \phi \\ T \end{bmatrix} \tag{10}$$

where $\mathbf{e'} = \mathbf{K'_{P_x}}\mathbf{x} + \mathbf{v}$, and $\mathbf{K'_{P_x}}, \mathbf{K'_{P_v}}, \mathbf{K'_{I_v}}$ are the gain matrices to be determined from Equation 3.

Let

$$\mathbf{G} = \begin{bmatrix} g & 0 & 0 \\ 0 & -g & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{11}$$

then the gains can be denoted as the following,

$$\begin{cases} \mathbf{K'_{P_v}}\mathbf{K'_{I_v}} = \mathbf{G}\mathbf{K_{P_v}}\mathbf{K_{I_v}} \\ \mathbf{K'_{P_x}}\mathbf{K'_{P_v}} + \mathbf{K'_{I_v}} = \mathbf{G}(\mathbf{K_{P_x}}\mathbf{K_{P_v}} + \mathbf{K_{I_v}}) \\ \mathbf{K'_{P_x}} + \mathbf{K'_{P_v}} = \mathbf{G}\mathbf{K_{P_v}} \end{cases} \tag{12}$$

Let $\mathbf{h} = \int \mathbf{e}' dt$, then the state space representation can be expressed as follows.

$$\begin{bmatrix} \dot{\mathbf{h}} \\ \dot{\mathbf{e}}' \end{bmatrix} = \begin{bmatrix} \mathbf{e}' \\ \mathbf{K}'_{\mathbf{P_x}} \dot{\mathbf{x}} + \ddot{\mathbf{x}} \end{bmatrix} \tag{13}$$

$$= \begin{bmatrix} \mathbf{e}' \\ -\mathbf{K}'_{\mathbf{P_v}} \mathbf{e}' - \mathbf{K}'_{\mathbf{I_v}} \int \mathbf{e}' dt \end{bmatrix} \tag{14}$$

$$= \begin{bmatrix} \mathbf{0}_{3\times3} & \mathbf{1}_{3\times3} \\ -\mathbf{K}'_{\mathbf{I_v}} & -\mathbf{K}'_{\mathbf{P_v}} \end{bmatrix} \begin{bmatrix} \mathbf{h} \\ \mathbf{e}' \end{bmatrix} \tag{15}$$

After substituting Equation 4, 5, and 6 into Equation 12 to get $\mathbf{K}'_{\mathbf{I_v}}, \mathbf{K}'_{\mathbf{P_v}}$, the matrix in Equation 15 is as follows.

$$\mathbf{A} = \begin{bmatrix} \mathbf{0}_{3\times3} & \mathbf{1}_{3\times3} \\ -\mathbf{K}'_{\mathbf{I_v}} & -\mathbf{K}'_{\mathbf{P_v}} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -0.815 & 0 & 0 & -0.556 & 0 & 0 \\ 0 & -0.125 & 0 & 0 & -0.712 & 0 \\ 0 & 0 & -0.305 & 0 & 0 & -0.234 \end{bmatrix} \tag{16}$$

All real parts of the eigenvalues of $\mathbf{A}$ are negatives. Therefore, the system is stable.

4 Communication

4.1 MAVLink/MAVROS

This section includes the MAVROS installation procedure, list of rostopics and rosservices used in the experiment, and some experience talk.

4.1.1 MAVROS Installation and Testing

- Install MAVLink and MAVROS from here. (If there is any weird error from MAVROS files in the build stage, make sure that the MAVROS folder is in master branch, `git branch` to check the branch and `git checkout master` to switch to master.) (or e.g. `git checkout -b v1.1.0 origin/master`)
- Check HITL with the following code: `roslaunch mavros px4.launch`. (If it shows the error of FCU: DeviceError:serial:open: No such file or directory, such as when the computer and pixhawk are communicated via radio telemetry, use `roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0`)
- Use `rostopic list` to see all the pub/sub topics, and `rostopic echo` to see the topic data. (Recommend topic to check communication success: /mavros/imu/data for pixhawk imu and /mavros/global_position/raw/fix for GPS.)
- See here for MAVROS topics.

4.1.2 Useful Rostopic and Rosservice

The list of rostopics used in the experiment are shown below.

- `/mavlink/from` – This rostopic shows the system identity of the drone.
- `/mavros/state` – This rostopic shows the connection, arm, and mode statuses of the drone.
- `/mavros/imu/data` – This rostopic shows the orientation of the drone from the inertial measurement unit (IMU) of Pixhawk FCU.
- `/mavros/gpsstatus/gps1/raw` – This rostopic shows the number of satellites sensed by the GPS module and the position of drone in terms of altitude, longitude, and latitude.
- `/mavros/global_position/local` – This rostopic shows the local position/coordinate of the drones in meter (ENU). The default origin is where it first detect GPS or say the place where the drone is powered on.
- `/mavros/setpoint_raw/local` – This rostopic can send the position, velocity, acceleration, and yaw command to the drone.
- `/mavros/setpoint_raw/attitude` – This rostopic can send the orientation, roll rate, pitch rate, yaw rate, and thrust command to the drone.
- `/mavros/global_position/home` – This rostopic can change the origin of /mavros/-global_position/local by giving the desired longitude and latitude.

The list of rosservices used in the experiment are shown below.

- `/mavros/cmd/arming` – This rosservice can command the drone to arm or disarm.
- `/mavros/set_mode` – This rosservice can command the drone to change mode (e.g. takeoff, land, offboard, manual ...)

rostopic echo to look into the rostopics. rostopic pub and rosservice call to command the UAV. Examples below.

- **Send attitude command:** rostopic pub /mavros/setpoint_raw/attitude mavros_msgs/AttitudeTarget "orientation: [100,0.55,120.99,0.99]"
- **ARM:** rosservice call /uav1/mavros/cmd/arming "true"
- **Change Model:** rosservice call /uav1/mavros/set_mode "custom_mode: OFFBOARD"
- **Reboot:** rosservice call /mavros/cmd/command "command: 246, param1: 1"

4.1.3 Multi-UAV

- **Add prefix to all rostopics:** Create a new package and a new folder called launch. Create a launch file to call "mavros px4.launch". Add the "uavID" argument/name in and cover the "include px4.launch" part with a group of namespace (ns), this namespace shall be uavID. This shall add a prefix before every topic, e.g. /mavlink/from changed to /uav1/mavlink/from if uavID:=uav1. (See 1.)
- There is this **MAV_SYS_ID** (> 0) pixhawk parameter to set up. This is useful when there are multiple drones flying and communicating to the gcs at the same time. The MAV_SYS_ID of each uav will be unique. And when roslaunching, set **tgt_system = MAV_SYS_ID**, so the uavID can match with the correct MAV_SYS_ID. (See 2.)
- Check out this website for tgt_system and tgt_component. tgt_system shall be the same as MAV_SYS_ID, and tgt_component shall be 1. (See 3.)
- Each MAVROS node is only in charge of the message of one drone. However, the open source MAVROS package is not capable to identify which MAVROS node shall communicate with which drone. Therefore, the MAVROS code is modified to fix this problem and have it works for point to multi-point communication (inspired by this). See the following code for modifications and see 4 for testes and results.

4 mavros/src/lib/mavros.cpp

```
@@ -233,7 +233,7 @@ void MavRos::mavlink_pub_cb(const mavlink_message_t *mmsg, Framing framing)
233 233   {
234 234         auto rmsg = boost::make_shared<mavros_msgs::Mavlink>();
235 235
236     -       if  (mavlink_pub.getNumSubscribers() == 0)
    236 +       if  (mavlink_pub.getNumSubscribers() == 0 || mmsg->sysid != mav_uas.get_tgt_system())
237 237                 return;
238 238
239 239         rmsg->header.stamp = ros::Time::now();
@@ -254,7 +254,7 @@ void MavRos::mavlink_sub_cb(const mavros_msgs::Mavlink::ConstPtr &rmsg)
254 254   void MavRos::plugin_route_cb(const mavlink_message_t *mmsg, const Framing framing)
255 255   {
256 256         auto it = plugin_subscriptions.find(mmsg->msgid);
257     -       if (it == plugin_subscriptions.end())
    257 +       if (it == plugin_subscriptions.end() || mmsg->sysid != mav_uas.get_tgt_system())
258 258                 return;
259 259
260 260         for (auto &info : it->second)
```

Figure 8: MAVROS code changed for multi-uav experiment

The deep color in the green lines in Figure 8 is the added code that fix the problem. The

`sysid` is the identity, the parameter `MAV_SYS_ID` in Pixhawk FCU, of the drone, and it shall be unique. The `tgt_system` is the input parameter when the MAVROS node is launched. The modified MAVROS code guarantees a MAVROS node will only communicate with the drone that has the `MAV_SYS_ID` the same as `sysid`.

- **Tests. Observations. Experiences.**
  1. Tested the mavlink topic/message for a single pixhawk by adding the namespace from .launch file and keeping the pixhawk (parameter) the same. No matter what the topic name is set up, e.g. /mavlink/from or /uav1/mavlink/from, or /uav2/mavlink/from or whatever, the laptop can always receive data from pixhawk. So the data topic name that sent from the pixhawk to laptop cooperate with the ones send out from laptop.
  2. When MAV_SYS_ID = 22, and tgt_system = 1 or 22, gcs shows state connected, and when tgt_system = 20, state is disconnected.
  3. Tried to change tgt_component, and the pixhawk cannot connect to qgroundcontrol anymore. Thankfully, it can still connect to mission planner, so changed the parameter back to one (tgt_component=1). Don't see the value of changing tgt_component right now.
  4. Run "roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0 tgt_system:=2" and have 3 rfd900x connected at the same time, where one is connected to laptop, another two are connected to pixhawk with MAV_SYS_ID being 1 and 2, respectively. Before the mavros.cpp is modified, the laptop cannot aim for the pixhawk specified and the mavros messages received jump between uav1 and uav2, meaning including command tgt_system:=2 is useless. After code is modified, everything works.
     However, due to two reasons, this set of system **does not completely work** for multiple drones experiment. Firstly, a MAVROS node receive a MAVLink protocol from an unknown drone, then check if the set of message is from the drone it want. Secondly, the radio network has low data transfer rate and thus the signals from multiple drones may interface with each other. The message cannot be sent on time or received with desired frequency. From experiment, single or two drones communication works well, and three drones communication works fine occasionally, and communication for more than four drones is terrible. On-board computers are used to solve for this problem. See more details in Section 4.

4.2 Dronekit

- Websites: dronekit python.
- List of parameters (vehicle.parameters): link.
- See here or Section 7.1 for example code.

4.3 Pymavlink

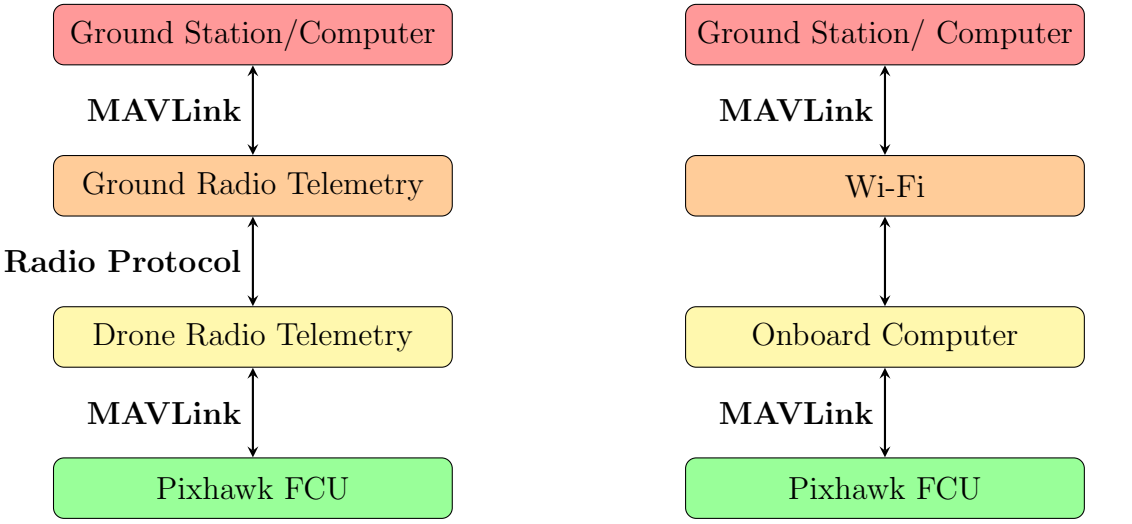
- Websites: Gitbook, mavlink.
- Names to use to obtain messages: Messages (common).
- See here, here, or Section 7.3 for example code.

4.4 XBee

- Websites: Digi XBee Python library, Github.
- See here or Section 7.2 for example code.

4.5 Network

There are two ways, via radio or via Wi-Fi, for the ground control station to communicate with the vehicles. The overall communication architectures are shown in Figure 9 where MAVLink is the message protocol designed for communication in the drone ecosystem. The radio communication has the advantage of large communication range, however, it suffers from low data transfer rate. On the other hand, Wi-Fi communication has higher data transfer rate, but has distance constraints, and has to guarantee power and Wi-Fi accessibility. Consider from the range and power accessibility perspective, radio is first chosen to be utilized in the outdoor experiment. However, Wi-Fi is the method used eventually, and the reason will be explained later.



(a) Radio communication architecture (b) Wi-Fi communication architecture

Figure 9: Communication architecture

4.5.1 Radio Communication

The ground control station (GCS) computer and each vehicle is required to connected to a radio telemetry. Therefore, `1+n` (amount of drone) number of radio telemetries are needed. The ground control station computer will launch `n` number of MAVROS node where each node has a unique namespace.

4.5.2 Wi-Fi Communication

Each vehicle requires a on-board computer, therefore, `n` number of Odroid-XU4 is needed. All on-board computer and the ground station computer are connected to the same local network. Each on-board computer will launch an open source MAVROS node with an unique group name.

ROS can run across multiple machines and thus the ground station computer can receive messages from all the drones. The group name allows the ROS system on ground station computer to know immediately which set of message is from which drone.

5 Ground Control Station

Steps for creating a qt-ros package:

1. Get the qt-ros package from here.
2. Create the package by using the command:
   `catkin_create_qt_pkg [package_name]`
   Edit the CMakeList.txt and package.xml file. Add the msg and srv folder. Add the needed .msg and .srv file into the folder.

Figure 10 shows the Qt ground control station (GCS) used for multiple drones outdoor experiment. The functionalities of this GCS include:

- Detect all the drones available and show them in the log on the left side of interface.
- Show the data of all detected drones in the log on the right side of interface.
- Set the origin of the world frame for all drones.
- Input the desired final positions for selected drones.
- Send command to the selected drone or all the detected drones.
- Let the drones to follow written paths or fly with path planning.

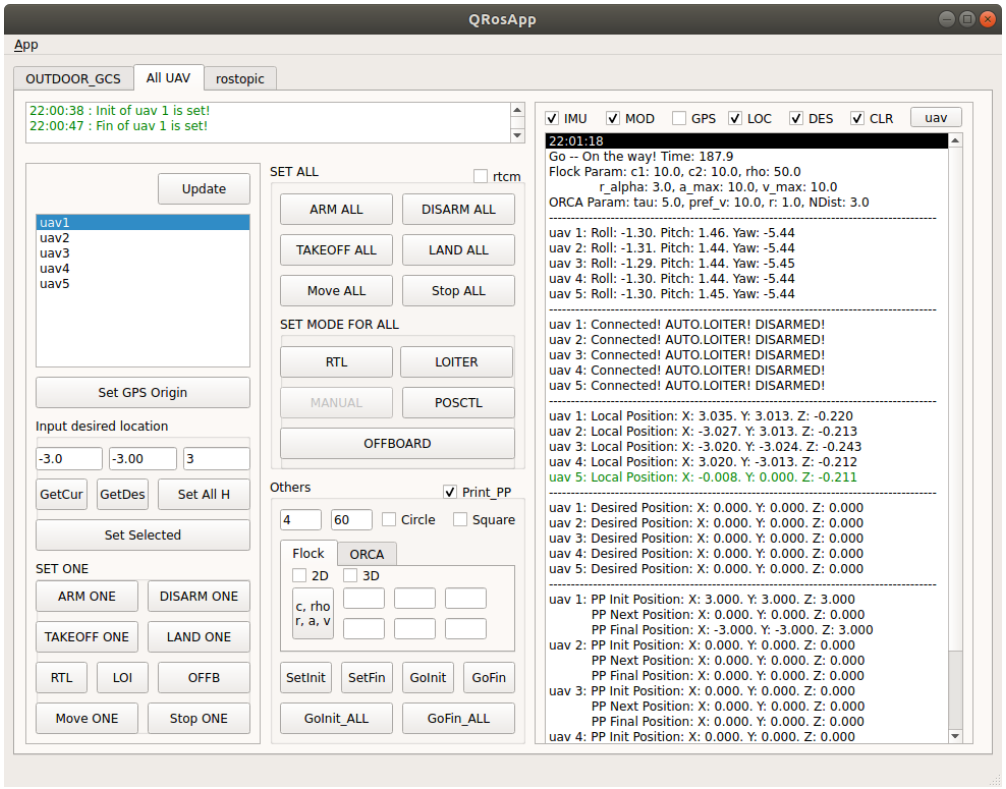


Figure 10: Qt ground control station for outdoor experiment

5.1 Observations

Modes related:

- Available modes for set_mode are: AUTO.PRECLAND, AUTO.FOLLOW_TARGET, AUTO.RTGS, AUTO.LAND, AUTO.RTL, AUTO.MISSION, RATTITUDE, AUTO.LOITER, STABILIZED,

AUTO.TAKEOFF, OFFBOARD, POSCTL, ALTCTL, AUTO.READY, ACRO, and MANUAL. See PX4 native flight stack of this page.

- Used setting mode to AUTO_TAKEOFF to takeoff instead of ∼cmd/takeoff. The height of this command is 2.5m.
- Once the setpoint is sent out, switch mode to **offboard**, then the UAV will move. If the setpoint is not sent, system cannot be switched to offboard mode.
- Somehow, mode would switch to **RTL**. So, remember to change the **RTL** parameters of the Pixhawk or vehicle would rush toward space.

/global_position/... rostopic related:

- ∼global_position/... such topics are empty if there is no GPS fixed and if the system is in MANUAL mode (Not sure what modes will make is nonempty, but AUTO-LOITER for sure).
- Topic type of rostopic global_position/local does not match with the one shown in the ros website.
- The z-component of velocity component of /global_position/local is opposite, e.g. velocity is negative when the UAV is moving up.
- Body frame (/global_position/local) is ENU.
- To have a world frame for multiple agents, publish the selected lat/lon/alt (from /global_position/global) and orientation (from imu) message to /global_position/home. This will change the x- and y-component of the /global_position/local, and z-component does not seem to be changing even an "alt" message is published. However, /setpoint_raw/local topic does not give out the command based on /global_position/local. A controller is needed for uav to travel to the desired location.

Others:

- Many topics are empty.
- Check geofence for safety. Geofence can be setup and upload from the top right corner tab "fence" of planning (A to B) icon of qgroundcontrol. Or parameter and safety tab of "Gear" icon.
- Use ∼**setpoint_raw/local** to set the desired location. Once the setpoint is sent out, switch mode to **offboard**, then the UAV will move. If the setpoint is not sent, system cannot be switched to offboard mode.
- rosservice call /mavros/cmd/set_home "current_gps: true"
- See here for defined frames such as map, odem ... etc.
- rosservice set_home does not change the origin of any frame, including global_position/global, global_position/local, and local_position/pose. The functionality is to tell the system where the "launch" position is. For example, where the uav shall return to when switched to RTL mode.
- Position of /setpoint_raw/local does not match with /global_position/local. Meaning, even though the origin of /global_position/local is changed via /global_position/home, position of /setpoint_raw/local stay the same as before.

6 Gazebo

Setup procedures for PX4 gazebo (reference),

1. Create a folder and download PX4 source code. Remember, do not download it in catkin_ws, it has no catkin property and do not need catkin build.
2. Type `gazebo` in terminal to check if the gazebo is working. Direct to the PX4 repo and `make px4_sitl_default gazebo`. It shall start building and the gazebo window shall pop out. Open QGroundControl, it shall communication with gazebo, and play with it (takeoff, land, action ...).
3. Remember to add line: `source $HOME/src/PX4-Autopilot/Tools/setup_gazebo.bash $HOME/src/PX4-Autopilot $HOME/src/PX4-Autopilot/build/px4_sitl_default > /dev/null 2>&1` in bashrc to loads dependencies for Gazebo simulation of a PX4 vehicle. If this line is missing, various features such as GPS or propeller dynamics will be nonfunctional.
4. Remember to add line: `export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$HOME/src/PX4-Autopilot:$HOME/src/PX4-Autopilot/Tools/sitl_gazebo` to register PX4 as a ROS package (such that rospack can find it, roslaunch can use PX4's launch scripts, etc).
5. Then `roslaunch px4 mavros_posix_sitl.launch` shall work. Play around with QGC.
6. Then try `roslaunch px4 multi_uav_mavros_sitl.launch` for multiple UAVs, which in this file, it has 3 UAVs.
7. Copy `roslaunch px4 multi_uav_mavros_sitl.launch` to somewhere else and modify it. Add/reduce number of UAVs and change their initial locations.
8. Create a shell script (.sh file) that can run multiple commands. The following example includes the px4_command controller and estimator used in gazebo.

```
gnome-terminal --window -e 'bash -c "roslaunch f450 multi_uav_mavros_sitl.
    launch"' \
--tab -e 'bash -c "sleep 10; roslaunch px4_command
    px4_multidrone_pos_estimator_outdoor.launch uavID:=uav1; exec bash"' \
--tab -e 'bash -c "sleep 10; roslaunch px4_command
    px4_multidrone_pos_controller_outdoor.launch uavID:=uav1; exec bash"' \
--tab -e 'bash -c "sleep 10; roslaunch px4_command
    px4_multidrone_pos_estimator_outdoor.launch uavID:=uav2; exec bash"' \
--tab -e 'bash -c "sleep 10; roslaunch px4_command
    px4_multidrone_pos_controller_outdoor.launch uavID:=uav2; exec bash"' \
--tab -e 'bash -c "sleep 10; roslaunch px4_command
    px4_multidrone_pos_estimator_outdoor.launch uavID:=uav3; exec bash"' \
--tab -e 'bash -c "sleep 10; roslaunch px4_command
--tab -e 'bash -c "sleep 10; rosrun outdoor_gcs outdoor_gcs"' \

```

7 Appendix

7.1 Dronekit Code

```
import time
from dronekit import connect

# Connect to the Vehicle
# vehicle = connect('/dev/ttyTHS1', wait_ready=True, baud=57600)
vehicle = connect('/dev/ttyACM0', wait_ready=True, rate=4, baud=57600)


while True:

    # IDs (can't get comp.id)
    MAVLink_ID, GCS_ID = vehicle.parameters['SYSID_THISMAV'], vehicle.parameters['SYSID_MYGCS']
    # GPS status: Fix: 0-1: no fix, 2: 2D fix, 3: 3D fix; Number of satellites visible.
    gps_fix, gps_num = vehicle.gps_0.fix_type, vehicle.gps_0.satellites_visible
    # imu: roll, pitch, yaw
    imu = vehicle.attitude
    roll, pitch, yaw = imu.roll, imu.pitch, imu.yaw # in rad
    # lat, long, alt
    location = vehicle.location.global_frame
    lat, lon, alt = location.lat, location.lon, location.alt
    # velocity: vx, vy, vz
    vx, vy, vz = vehicle.velocity[0], vehicle.velocity[1], vehicle.velocity[2]
    # heading
    heading = vehicle.heading # in deg
    # status https://dronekit-python.readthedocs.io/en/latest/automodule.html#dronekit.Vehicle.system_status
    MAV_state = vehicle.system_status.state
```

7.2 Pymavlink Code

```
import time
from datetime import datetime
from serial.serialutil import SerialException
import sys
import os
from pymavlink import mavutil, mavwp
from info import info

# Start a connection
master = mavutil.mavlink_connection('/dev/ttyACM0')
# Wait for the first heartbeat
master.wait_heartbeat()
print("Heartbeat from system (system %u component %u)" % (master.target_system, master.target_component))
# Initialize data stream
rate = 2 # desired transmission rate
master.mav.request_data_stream_send(master.target_system, master.target_component, mavutil.mavlink.MAV_DATA_STREAM_ALL, rate, 1)
# For checksum calculation
```

```
cs = mavutil.x25crc()

ctrl, last_ask_time = True, 0  # if to send/receive control command from key input

# Initialize parameters for drone data
sysID, compID = master.target_system, master.target_component
start_time, start_uptime = master.start_time, master.uptime
SYS_time, sysgps_time, IMU_time_boot, GPS_time_usec, GPSACC_time_boot = 0, 0, 0, 0, 0   # in ms
roll, pitch, yaw = 0, 0, 0                                                              # in deg
fix, num, lat, lon, alt = 0, 0, 0, 0, 0                                                 # in degE7 and mm
vx, vy, vz, heading = 0, 0, 0, 0                                                        # in cm/s and cdeg
MAV_state, battery, failsafe = 0, 0, 99                                                 # int, num in %, bool
mode, arm = 0, 0
command, result = 0, 0

# get already loaded mission
# master.waypoint_clear_all_send()
master.waypoint_request_list_send()
msg = None
while not msg:
    print('Waiting for mission count...')
    msg = master.recv_match(type=['MISSION_COUNT'],blocking=True,timeout=1)
print('Preloaded mission count: ', msg.count)
count, seq = msg.count, 0
while (seq < count):
    master.waypoint_request_send(seq)
    msg = master.recv_match(type=['MISSION_ITEM'],blocking=True,timeout=1)
    if not msg:
        print('MISSION_ITEM is none ...')
        continue
    seq = msg.seq + 1
    print('Preloaded mission seq, command, x, y, z: ', msg.seq, msg.command, msg.x, msg.y, msg.z)


while True:
    try:
        if mode != master.flightmode:
            mode = master.flightmode
            print(mode)
        if arm != master.sysid_state[master.sysid].armed:
            arm = master.sysid_state[master.sysid].armed
            print(arm)
        # Get data from pixhawk via pymavlink
        msg = None
        try:
            msg = master.recv_match(blocking=True)
            msg_type = msg.get_type()
        except SerialException: pass
        if msg == None:
            continue
        elif msg_type == "SYSTEM_TIME":             # system boot time
```

```
            SYS_time, sysgps_time = msg.time_boot_ms, msg.time_unix_usec
        elif msg_type == "ATTITUDE":                # imu: time, roll, pitch, yaw
            IMU_time_boot, roll, pitch, yaw = msg.time_boot_ms, round(msg.roll
    *57.2958), round(msg.pitch*57.2958), round(msg.yaw*57.2958) #msg.roll, msg.pitch
    , msg.yaw
        elif msg_type == "GPS_RAW_INT":             # GPS status: time_usec/boot, fix
    , sat_num
            GPS_time_usec, fix, num = msg.time_usec, msg.fix_type, msg.
    satellites_visible
        elif msg_type == "GLOBAL_POSITION_INT":     # Fused GPS and accelerometers:
    location, velocity, and heading
            GPSACC_time_boot, lat, lon, alt = msg.time_boot_ms, msg.lat, msg.lon,
    msg.relative_alt # originally in degE7 and mm
            vx, vy, vz, heading = msg.vx, msg.vy, msg.vz, msg.hdg # originally in
    cm/s and cdeg
            print(msg)
        elif msg_type == "HEARTBEAT":               # MAV_STATE
            MAV_state = msg.system_status
        elif msg_type == "BATTERY_STATUS":          # Battery status
            battery = msg.battery_remaining
        elif msg_type == "HIGH_LATENCY2":
            if msg.HL_FAILURE_FLAG > 0: # https://mavlink.io/en/messages/common.
    html#HL_FAILURE_FLAG
                failsafe = False
        elif msg_type == "STATUSTEXT":
            if msg.severity < 4: # https://mavlink.io/en/messages/common.html#
    MAV_SEVERITY
                failsafe = False
        elif msg_type == "MISSION_CURRENT":
            current_mission_seq = msg.seq
        elif msg_type == "MISSION_ACK":
            print("MISSION_ACK: ", msg.type) # https://mavlink.io/en/messages/
    common.html#MAV_MISSION_RESULT
        elif msg_type == "COMMAND_ACK": # https://mavlink.io/en/messages/common.
    html#MAV_RESULT
            if (command != msg.command) or (result != msg.result):
                command, result = msg.command, msg.result
            print("COMMAND_ACK: ", command, result)
        elif msg_type ==’POSITION_TARGET_GLOBAL_INT’:
            print(msg)
        # elif msg_type ==’POSITION_TARGET_LOCAL_NED’:
        #     print("HERE!!!!!!!!!!!!!!!")
        elif msg_type == "SERVO_OUTPUT_RAW":
            # print(msg)
            pass
        elif msg_type == "LOCAL_POSITION_NED":
            print(msg)
    except KeyboardInterrupt:
        if ctrl: # and (time.time() - last_ask_time > 3):
            input_command = input("0-9: set mode, 10: arm, 11: disarm, 12: mission,
     13: takeoff, 14: mission start, 99: write csv ...: ")
            try:
                if int(input_command) < 10:
                    input_mode = int(input_command)
                    if input_mode == 8: # convert position mode number
                        input_mode = 16
```

```
                    master.mav.command_long_send(sysID, compID, mavutil.mavlink.
MAV_CMD_DO_SET_MODE, 0,
                                                 mavutil.mavlink.
MAV_MODE_FLAG_CUSTOM_MODE_ENABLED, input_mode, 0, 0, 0, 0, 0)
                elif int(input_command) == 10:
                    master.arducopter_arm()
                    # master.motors_armed_wait()
                elif int(input_command) == 11:
                    master.arducopter_disarm()
                    # master.motors_disarmed_wait()
                elif int(input_command) == 12:
                    mission_num = input("Input mission number: ")
                    wp = mavwp.MAVWPLoader()
                    frame = mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT
                    for i in range(int(mission_num)):
                        wp.add(mavutil.mavlink.MAVLink_mission_item_message(
                            sysID, compID,
                            i,
                            frame,
                            info.mission_mode_mapping[2],
                            0, 0, 0, 0, 0, 0,
                            24+i*0.1, 121+i*0.1, 3))
                    master.waypoint_clear_all_send()
                    master.waypoint_count_send(int(mission_num))
                    ack = 99
                    while ack == 99:
                        msg = master.recv_match(type=['MISSION_REQUEST'],blocking=
True)
                        print(msg)
                        master.mav.send(wp.wp(msg.seq))
                        # https://mavlink.io/en/messages/common.html#
MAV_MISSION_RESULT
                        msg = master.recv_match(type=['MISSION_ACK'],blocking=True,
timeout=0.1)
                        try: ack = msg.type
                        except: pass
                    print("mission result: ", ack)

                elif int(input_command) == 13:
                    takeoff_alt = 20
                    master.set_mode(4)
                    msg = master.recv_match(type=['COMMAND_ACK'],blocking=True)
                    print("takeoff: ", msg.command, msg.result)
                    if (msg.result == 0):
                        master.mav.command_long_send(sysID, compID, mavutil.mavlink
.MAV_CMD_NAV_TAKEOFF,
                                                     0, 0, 0, 0, 0, 0, 0, takeoff_alt)
                    print('takeoff command sent!!')

                elif int(input_command) == 14:
                    master.mav.command_long_send(sysID, compID, mavutil.mavlink.
MAV_CMD_MISSION_START, 0, 0, 0, 0, 0, 0, 0, 0)
                    print('mission start!')

                elif int(input_command) == 15: # position
                    print('position cmd sent 1')
```

```
                    for i in range(5):
                        master.mav.send(mavutil.mavlink.
    MAVLink_set_position_target_global_int_message(0, sysID, compID, 6, int(0
    b100111111000), 18, 220, 1, 0, 0, 0, 0, 0, 0, 9, 9))

                elif int(input_command) == 16: # velocity
                    master.mav.send(mavutil.mavlink.
    MAVLink_set_position_target_global_int_message(0, sysID, compID, 6, int(0
    b010111000000), 0, 0, 0, 0.9, 0.8, 0, 0, 0, 0, 0, 0))

                    print('velocity cmd sent')

                elif int(input_command) == 17: # acceleration
                    for i in range(50):
                        master.mav.send(mavutil.mavlink.
    MAVLink_set_position_target_global_int_message(0, sysID, compID, 6, int(0
    b110000111111),
                                0, 0, 0, 0, 0, 0, 9, 8, 0, 0, 0))
                                 #0b110111000111
                    print('acceleration cmd sent')

                elif int(input_command) == 99:
                    if save_csv:
                        print(data_list)
                        write_csv(data_list)
                        data_list = [['time', 'mode', 'lat', 'lon', 'alt', 'vx', '
    vy', 'vz', 'roll', 'pitch', 'yaw']]
                    else: print('Not commanded to save data...')

            except: pass
            last_ask_time = time.time()
        else: print('not for control command')
```

7.3 Xbee Code

```
from struct import *
from pymavlink import mavutil, mavwp
from digi.xbee.devices import DigiMeshDevice,RemoteDigiMeshDevice,XBee64BitAddress

# Connect xbee1
xbee001 = DigiMeshDevice('/dev/ttyUSB0', 115200)
remote002 = RemoteDigiMeshDevice(xbee001,XBee64BitAddress.from_hex_string("0013
    A20040F5C5DB"))
xbee001.open(force_settings=True)

# pack and send packets
def sent_pkt():
    pkt_bytearray = bytearray([255])
    pkt_bytearray.extend(pack('<BBBBiiiii', msgID, sysID, compID, commID, seq, cmd,
     lat, lon, alt))
    # store computer system time and gps time
    pkt_bytearray.extend(pack('i',int((utctime.minute*60 + utctime.second)*1e3 +
    round(utctime.microsecond/1e3))))
    # calculae checksum
    chks.accumulate(pkt_bytearray[:])
    pkt_bytearray.extend(pack('H', chks.crc))
```

```
19      # try: xbee001.send_data_broadcast(pkt_bytearray[i])
20      except: pass
21
22  # Unpack and read packets
23  def read_pkt():
24      try:
25          # first try to read via xbee, if xbee is connected
26          received = xbee001.read_data()
27          seq = data[4]
28          lat = unpack('i',data[5:9])[0]
29          data = received.data
30      except:
31          pass
```

7.4 Tmux and Shell Script

Credit to Helson Go.

This section provides the code used for the ground control station laptop to ssh to the onboard computer and call multiple commands. The following .sh file takes three arguments, session, host ip, and UAV id. Session is a name for the a specific tmux window, and can be named randomly just not to be the same as other tmux windows. Host ip is the ip address of the onboard computer where we wish to ssh to. UAV id is the UAV id/namespace for launching the ros nodes. In this example, three windows are opened, and they run mavros, estimator, and controller as shown in Line 56-58.

```
1  rosrun f450 local_launch.sh ros1 odroid@192.168.2.179 uav1
```

Above shows a command on how to call the file if this .sh file is called local_launch.sh and is located inside the f450 rospackage. ros1 is the session name. odroid@192.168.2.179 is the onboard odroid computer ip address, and uav1 is the namespace for the ros nodes.

```
1  #!/bin/bash
2
3  SESSION=$1
4  HOST_IP=$2
5  UAV=$3
6
7  tmuxstart() {
8      if [[ $(tmux has-session -t "$1") -eq 0 ]] ; then
9          echo "Killing previous session with name $1"
10         tmux kill-session -t  "$1"
11     fi
12     #rest of tmux script to create session named "sess"
13     tmux new-session -d -s "$1"
14 }
15
16 splitandrun() {
17     tmux send-keys -t $1 "tmux split-window -h $2 && tmux select-layout even-
       horizontal" ENTER
18 }
19
20 sendcmd() {
21     tmux send-keys -t $1 "$2" ENTER
22 }
23 rostopic list >/dev/null 2>&1
```

```
if [ $? -ne 0 ] ; then
    echo "rosmaster not started! Exiting."
    exit 1
fi

IP=$(sed 's&.*@\(\)&\1&' <<< ${HOST_IP})

until ping -c1 ${IP} >/dev/null 2>&1; do
    echo "Pinging $IP...";
done

read -p "Destination $IP reached. Press enter to begin tmux session and ssh to
   remote vehicle"

tmuxstart ${SESSION}

# Split panes then ssh to the vehicle in each pane
splitandrun ${SESSION} "ssh -X ${HOST_IP}"
splitandrun ${SESSION} "ssh -X ${HOST_IP}"

# ssh to the vehicle in the original pane
sendcmd 0 "ssh -tt -X ${HOST_IP}"

# Must wait, otherwise panes other than 0 may not initialize properly.
    echo "Wait for panes to fully initialize."

for COUNTDOWN in 3 2 1
do
    echo $COUNTDOWN
    sleep 1
done

sendcmd 0 "roslaunch px4_command mavros_multi_drone.launch uavID:=$UAV fcu_url:=/
   dev/ttyUSB0:921600"
sendcmd 1 "roslaunch px4_command px4_multidrone_pos_estimator_outdoor.launch uavID
   :=$UAV"
sendcmd 2 "roslaunch px4_command px4_multidrone_pos_controller_outdoor.launch uavID
   :=$UAV"

## Create the windows on which each node or .launch file is going to run

gnome-terminal --tab -- tmux attach -t ${SESSION}
```

7.5 Firmware and MAVLink

This section recorded the procedures I have done to setup the MAVLink, includes customizing message and xml file, firmware, building and uORB.

7.5.1 Some Information for Custom Message

1. Download the pre-built MAVLink source files if you're working in a C/C++ project and using standard dialects. Otherwise, install mavlink and define custom messages.

2. Install Mavlink. Type `export PYTHONPATH=path_to_mavlink` in `~/.bashrc`. Now, it is ready to Generate MAVLink Libraries. We can check if mavlink is installed properly by using a command from this link: `python3 -m mavgenerate`. A small window named **MAVLink Generator** shall pop out
3. Defining Custom MAVLink Messages! MAVLink enums, messages, commands, and other elements are defined within **XML** files and then converted to libraries for supported programming languages using a **code generator**.
    - Click here to see the **MAVLink XML file schema/format**. This link have a lot of setting description, e.g. dialects can use id = 180-229 for custom messages.
    - Messages vs Commands (when to use one or the other)
    - MAVLink XML files are located in mavlink directory under /message_definitions/v1.0/.
    - Click here to **create a dialect/xml file**
    - Send and receive mavlink message
    - Use code generator to create **MAVLink libraries** from **XML** message definitions. Use mavgenerate (GUI) or mavgen (command line) for generation.
4. Generate Source Files for ROS and download MAVROS
5. Using C MAVLink Libraries
6. Add the new uORB topic!
    - PX4 middleware: (1) PX4 internal communication mechanisms (**uORB**); (2) between PX4 and offboard systems like companion computers and GCS (e.g. **MAVLink**, RTPS).
    - uORB is an asynchronous publish() / subscribe() messaging API used for inter-thread/inter-process communication
    - New uORB topics can be added either within the main **PX4/Firmware repository**, or can be added in an out-of-tree message definitions.
    - Check here for toolchain installation, Pixhawk/NuttX development toolchain is necessary for building PX4/Firmware.
    - Check here for installation and building of PX4/Firmware. Git clone. Build by make command (in my case, `make px4_fmu-v2_default`). Connect Pixhawk with computer by USB. Then run upload (`make px4_fmu-v2_default upload`).

7.5.2 Custom Message Steps

1. **Create MAVLink Message**

   Download PX4/Firmware from Github. (Might also need to download Pixhawk/NuttX development toolchain from here.) (Name of PX4/Firmware may changed, for example PX4/PX4-Autopilot.)

   Go to `/Firmware/mavlink/include/mavlink/v2.0/message_definitions`, and create a **.xml** file. I have the file named as **utias.xml**. Write whatever message I need there.

   Install Mavlink. (See the previous section for more details.) Open the terminal and cd to the mavlink directory and open the GUI of `mavgenerate.py` by typing the following command: `python3 -m mavgenerate`

    - XML: Select the .xml that is for library generation. In this case it would be: /Firmware/-mavlink/include/mavlink/v2.0/message_definitions/utias.xml
    - Out: /Firmware/mavlink/include/mavlink/v2.0/

- Language: C++11
- Protocol: ideally 2.0
- Validate/validate unit: checked validates XML specifications

Press Generate. A new folder, named *utias* in this case, will appear in the output directory chosen. Below is the **utias.xml**.

```
    <mavlink>
    <include>common.xml</include>
    <include>ardupilotmega.xml</include>
    <version>3</version>
        <messages>
            <message id="228" name="utias">
            <description>Testing msg</description>
                <field type="float[4]" name="quat">
                Quaternion components, w, x, y, z (1 0 0 0 is the null-
    rotation)
                </field>
                <field type="float[3]" name="pos">Position XYZ</field>
                <field type="float[3]" name="vel">Velocity XYZ</field>
                <field type="float[3]" name="gps">Lat, Long in degE7 and alt</
    field>
            </message>
        </messages>
    </mavlink>

```

2. **Create uORB Message**

   Go to `/Firmware/msg` and create a **.msg** file. I have the file named as **utias.msg**. Write whatever message I need there. An example is shown below. The contents better match with the **.xml** file created in the previous step.

   Add the file name (**utias.msg**) to the `/Firmware/msg/CMakeLists.txt` list. From this, the needed C/C++ code is automatically generated.

```
    uint64 timestamp      # time since system start (microseconds)
    float32[4] quat         # quat
    float32[3] pos        # X Y Z coordinate in meters
    float32[3] vel        # velocity
    uint32[3] gps       # Latitude, Longitude in degrees, Altitude

```

3. **Send Custom Message**

   Open `/Firmware/src/modules/mavlink/mavlink_messages.cpp`. Add the following into the code:

```
#include <uORB/topics/utias.h>
#include <v2.0/utias/mavlink_msg_utias.h>
```

   A new class, named **MavlinkStreamUtias**, shall be created in `mavlink_messages.cpp`. And append the stream class to the *streams_list* at the bottom of code.

```
class MavlinkStreamUtias : public MavlinkStream
{
public:
```

```
    const char *get_name() const override
    {
      return MavlinkStreamUtias::get_name_static();
    }

    static constexpr const char *get_name_static()
    {
      return "UTIAS";
    }

    static constexpr uint16_t get_id_static()
    {
      return MAVLINK_MSG_ID_utias;
    }

    uint16_t get_id() override
    {
      return get_id_static();
    }

    static MavlinkStream *new_instance(Mavlink *mavlink)
    {
      return new MavlinkStreamUtias(mavlink);
    }

    unsigned get_size() override
    {
      return _utias_sub.advertised() ? MAVLINK_MSG_ID_utias +
   MAVLINK_NUM_NON_PAYLOAD_BYTES : 0;
    }

private:
    uORB::Subscription _utias_sub{ORB_ID(utias)};

    /* do not allow top copying this class */
    MavlinkStreamUtias(MavlinkStreamUtias &) = delete;
    MavlinkStreamUtias &operator = (const MavlinkStreamUtias &) = delete;

protected:
    explicit MavlinkStreamUtias(Mavlink *mavlink) : MavlinkStream(mavlink)
    {}

    bool send(const hrt_abstime t) override
    {
        utias_s utias;

      if (_utias_sub.update(&utias)) {
            mavlink_utias_t msg{};

            memcpy(msg.quat, utias.quat, sizeof(msg.quat));
            memcpy(msg.pos, utias.pos, sizeof(msg.pos));
            memcpy(msg.vel, utias.vel, sizeof(msg.vel));
            memcpy(msg.gps, utias.gps, sizeof(msg.gps));

            mavlink_msg_utias_send_struct(_mavlink->get_channel(), &msg);

```

```
            return true;
        }

        return false;
    }
};


static const StreamListItem streams_list[] = {
    create_stream_list_item<MavlinkStreamHeartbeat>(),
    create_stream_list_item<MavlinkStreamStatustext>(),
    ...
    create_stream_list_item<MavlinkStreamUtias>()
};

```

4. **Receive Custom Message**

   Open `/Firmware/src/modules/mavlink/mavlink_receiver.h`. Add the following into the code:

   `#include <uORB/topics/utias.h>`

   `#include <v2.0/utias/mavlink_msg_utias.h>`

   Add a function that handles the incoming MAVLink message in the MavlinkReceiver class in mavlink_receiver.h: `void handle_message_utias(mavlink_message_t *msg);`

   Add an uORB publisher in the MavlinkReceiver class in mavlink_receiver.h:

   `uORB::Publication<utias_s> _utias_pubORB_ID(utias);`

   `orb_advert_t _utias_msg_pub;` (This is **not** added yet)

   Implement the handle_message_utias function in mavlink_receiver.cpp. Called the function in *MavlinkReceiver::handle_message()*.

```
void MavlinkReceiver::handle_message(mavlink_message_t *msg)
{
    switch (msg->msgid) {
    case MAVLINK_MSG_ID_COMMAND_LONG:
      handle_message_command_long(msg);
      break;
    ...
    case MAVLINK_MSG_ID_utias:
      handle_message_utias(msg);
      break;

    default:
      break;
    }
    ...

void MavlinkReceiver::handle_message_utias(mavlink_message_t *msg)
{
    mavlink_utias_t utias;
    mavlink_msg_utias_decode(msg, &utias);

    utias_s utias_msg{};
    utias_msg.timestamp = hrt_absolute_time();
```

```
    memcpy(utias_msg.quat, utias.quat, sizeof(utias.quat));
    memcpy(utias_msg.pos, utias.pos, sizeof(utias.pos));
    memcpy(utias_msg.vel, utias.vel, sizeof(utias.vel));
    memcpy(utias_msg.gps, utias.gps, sizeof(utias.gps));

    _utias_pub.publish(utias_msg);
}

```

5. **Enable MAVLink stream**
   Open `/Firmware/src/modules/mavlink/mavlink_main.h`. Add the following into the code:
   `#include <v2.0/utias/mavlink_msg_utias.h>`
   Open `/Firmware/src/modules/mavlink/mavlink_main.cpp`.
   Find *"Mavlink::configure_streams_to_default(const char *configure_single_stream)"* and in *"case MAVLINK_MODE_NORMAL"* add the following into the code:
   `configure_stream_local("UTIAS", 10.0f);`

6. **Assign Values to uORB msg**
   Open `/Firmware/src/modules/mc_pos_control/mc_pos_control_main.cpp`. Add the following into the code: (mc_pos_control is chosen because it is continuously updating.)
   `#include <uORB/topics/utias.h>`
   and add the following in the *private* declaration section.
   `uORB::Publication<utias_s> _utias_pub;`
   In *MulticopterPositionControl::MulticopterPositionControl(bool vtol)* add
   `_utias_pub(ORB_ID(utias)),`
   Assign the values and publish the message at the proper code line/location. See the code for example.

```
MulticopterPositionControl::MulticopterPositionControl(bool vtol) :
    SuperBlock(nullptr, "MPC"),
    ModuleParams(nullptr),
    WorkItem(MODULE_NAME, px4::wq_configurations::nav_and_controllers),
    _vehicle_attitude_setpoint_pub(vtol ? ORB_ID(mc_virtual_attitude_setpoint)
    : ORB_ID(vehicle_attitude_setpoint)),
    _utias_pub(ORB_ID(utias)),
    _vel_x_deriv(this, "VELD"),
    _vel_y_deriv(this, "VELD"),
    ...

utias_s utias_msg{};
utias_msg.timestamp = time_stamp_now;
utias_msg.pos[0] = _states.position(0);
utias_msg.pos[1] = _states.position(1);
utias_msg.pos[2] = _states.position(2);
utias_msg.vel[0] = _states.velocity(0);
utias_msg.vel[1] = _states.velocity(1);
utias_msg.vel[2] = _states.velocity(2);

_utias_pub.publish(utias_msg);

```

7. **Make. Build. Debug. (Firmware)**
   Change directory to `/Firmware/` and build using `make px4_fmu-v2_default`. Study the errors and fix them. Some observations and sessions,

   - struct mavlink_name_t, e.g. mavlink_utias_t, is defined in /mavlink/v2.0/utias/mavlink_msg_utias.h, which is a structure that include all the parameters defined in the .xml file.
   - struct name_s, e.g. utias_s, is defined based on the uORB message(.msg file).

8. **Build QGroundControl**
   This section added the extra mavlink library into QGC. Steps are shown in here. In short, the steps are git clone the mavlink QGC files, run ./qgroundcontrol-start.sh in terminal, download QtCreator, launch Qt Creator and open the qgroundcontrol.pro project, build by using the hammer icon, and execute by pressing the run icon, then the QGC gui shall pop out.

   Some errors, solutions, and things to notice:

   - qgroundcontrol-start.sh is not in the root directory. Use `locate` to find it.
   - Missing file errors while running ./qgroundcontrol-start.sh. Try to find the files and copy and paste them to the location indicated. If there is weird problems comment out the last line of code: "$*HERE/QGroundControl*" "$@".
   - Creating custom MAVLink message in qgroundcontrol/libs/mavlink/include/mavlink/v2.0 (the file will be the same as the one in Firmware/mavlink, and remember to have both common.xml and ardupilotmega.xml included). Create a user_config.pri in the QGC root directory, and add the line `MAVLINK_CONF = custom_msgs_name`. This will tell QGC to build the customized MAVLink library.
   - Version of QtCreator is too old. Remember to click on the version needed in the downloading stage and set the new version to default in the project → kit manage (→ Qt version) page.

7.5.3 Summary

MAVLink	**uORB**
Communication between PX4 and GCS	**PX4 internal communication**
.xml file	**.msg file**
Use **generator** to get library	Add msg name to **CMakeLists** (**Auto**-generate)

Figure 11: MAVLink and uORB comparison

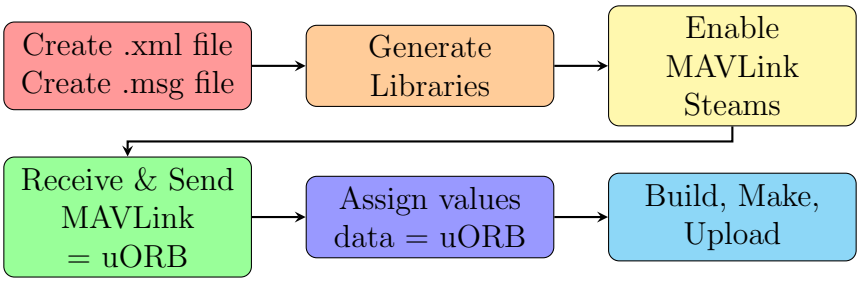


Figure 12: Steps to customize message