Educational for Drone (eDrone)

574090-EPP-1-2016-1-IT-EPPKA2-CBHE-JP

Educational for Drone
(eDrone)
Acquisition and processing of flight data

Prof. Francesco Picariello, Ph.D.

Co-funded by the
Erasmus+ Programme
of the European Union

Università degli Studi del Sannio

This project has been funded with support from the European Commission. This publication (communication) reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Drone electronics

Drone electronics must handle two types of tasks:

- 1 – To balance the motor in order to obtain the proper orientation and trajectory
- 2 – To handle complex tasks such as autonomy, collision avoidance, computer vision, IP communication

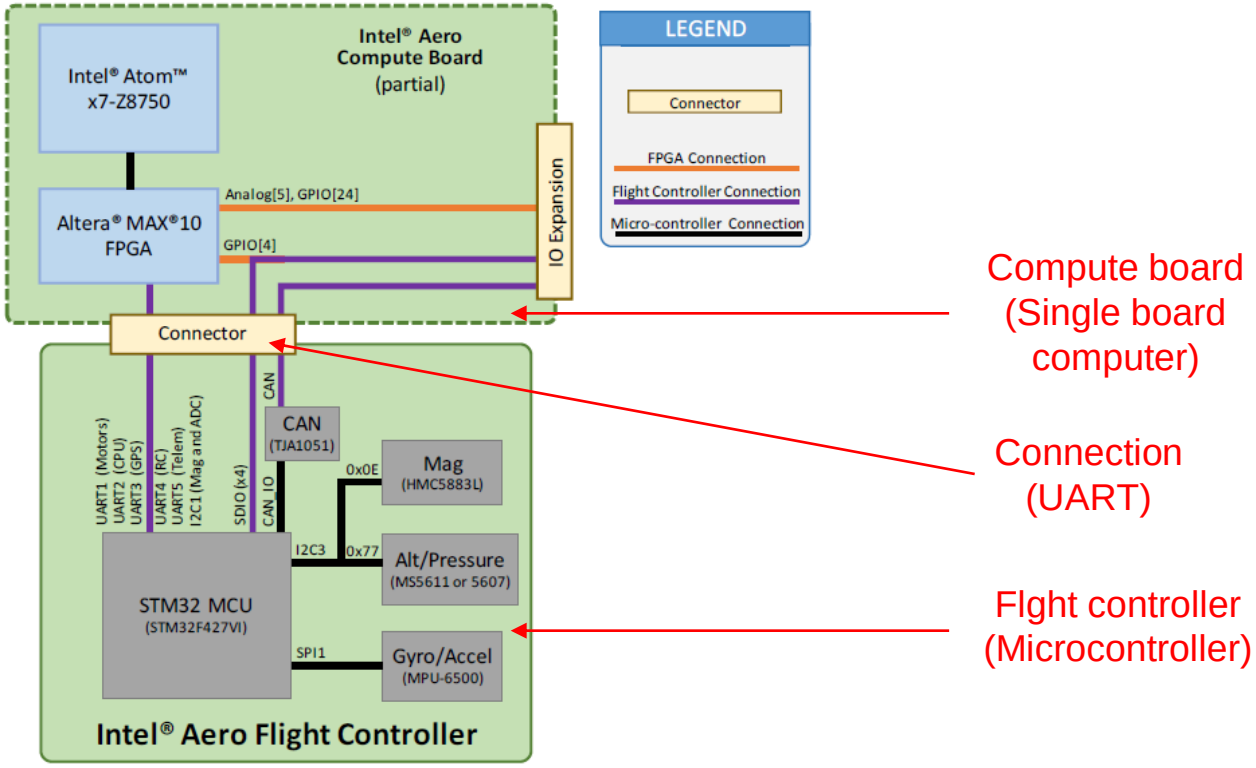
The first type should be accomplished quickly, constantly and without interruption.

That's why this task is managed by a simple microcontroller. *Ex: STM32*

The microcontroller has limited processing power,
but it is a deterministic behavior. You can be sure it will run your software in real-time.

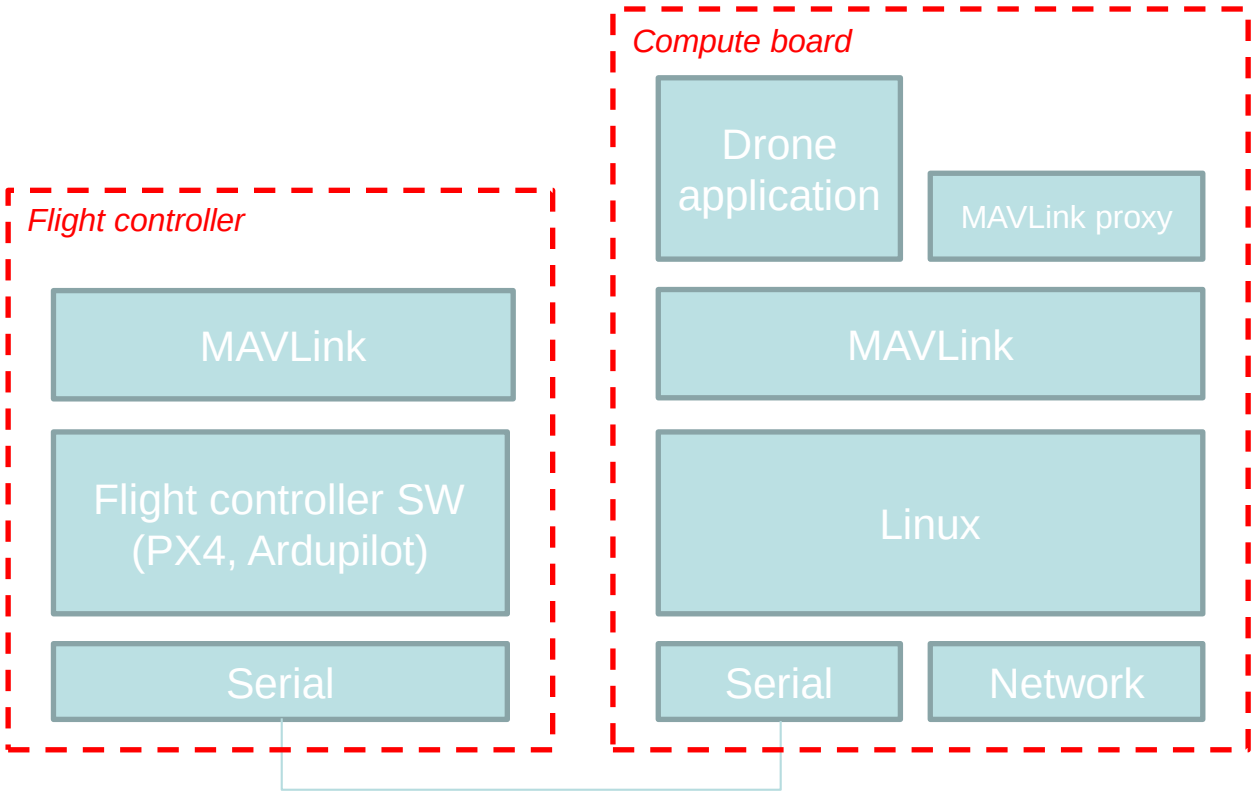
For the second type, a computer board is used, typically running Linux.

Drone electronics



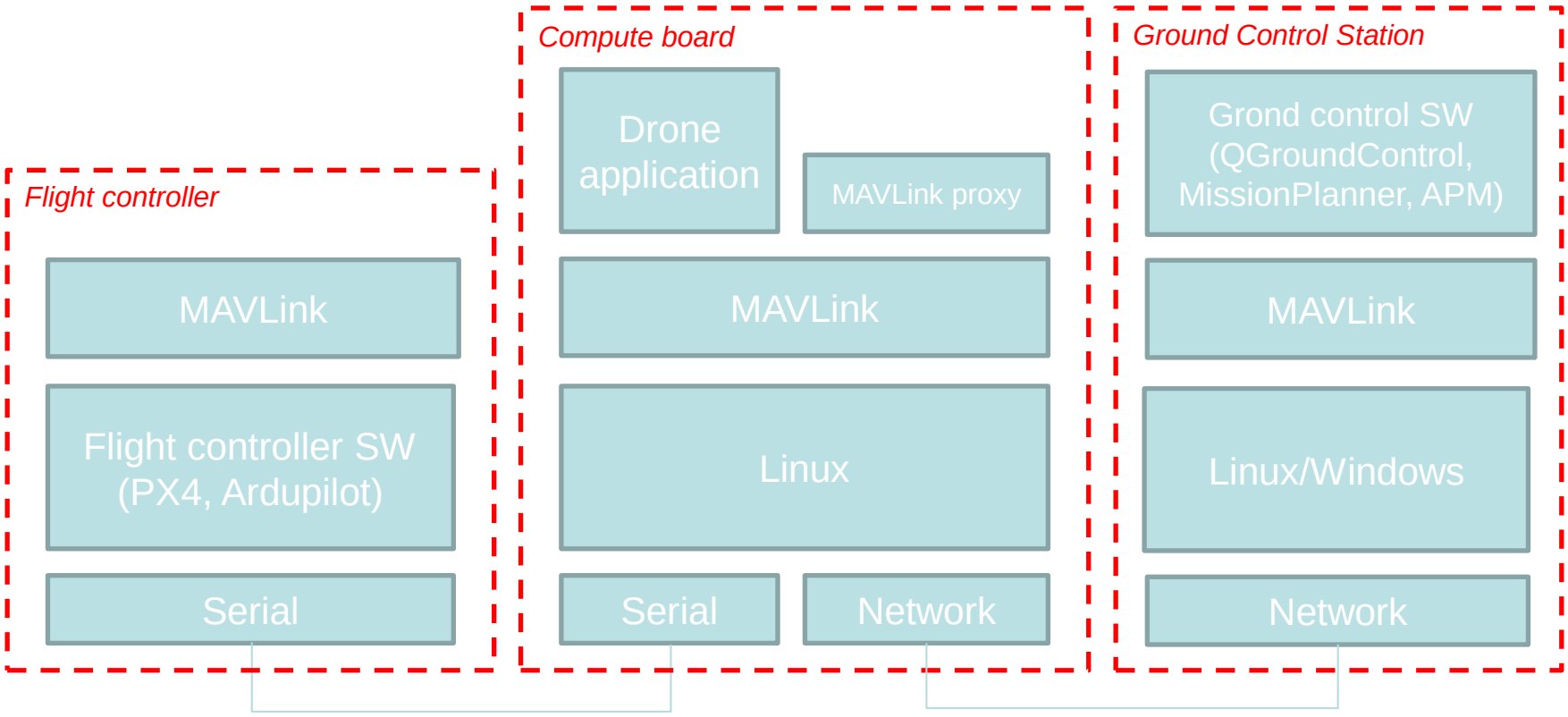
Prof. Francesco Picariello, Ph.D.

Software architecture



MAVLINK
(Micro Air Vehicle Link) is an industry-standard protocol for communicating with flight controllers

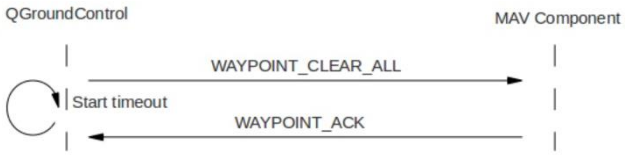
Ground control station



MAVLink

MAVLink message structure

Byte Index	Content	Value	Explanation
0	Packet start sign	v1.0: 0xFE (v0.9:	Indicates the start of a new packet.
1	Payload length	0 - 255	Indicates length of the following payload.
2	Packet sequence	0 - 255	Each component counts up his send sequence. Allows to detect packet loss
3	System ID	1 - 255	ID of the SENDING system. Allows to differentiate different MAVs on the same network.
4	Component ID	0 - 255	ID of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
5	Message ID	0 - 255	ID of the message - the id defines what the payload "means" and how it should be correctly decoded.
6 to (n+6)	Data	(0 - 255) bytes	Data of the message, depends on the message id.
(n+7) to (n+8)	Checksum (low byte, high byte)	ITU X.25/SAE AS-4 hash, **excluding packet start sign, so bytes 1..(n+6)** Note: The checksum also includes MAVLINK_CRC_EXTRA (Number computed from message fields. Protects the packet from decoding a different version of the same packet but with different variables).	



Example of a message (WAYPOINT_CLEAR_ALL) sent by the GCS to a UAV. The UAV receives and executes, and responds with another message (WAYPOINT_ACK).
After sending the initial message, the GCS starts a timer to decide for a timeout state if no ACK messages is received .

Example of an XML Heartbeat Message

```
<message id="0" name="HEARTBEAT">
  <description>The heartbeat message shows that a system is present and responding. The
      type of the MAV and Autopilot hardware allow the receiving system to treat further
      messages from this system appropriate (e.g. by laying out the user interface based on the
      autopilot).</description>
  <field type="uint8_t" name="type">Type of the MAV (quadrotor, helicopter, etc., up to 15 types,
      defined in MAV_TYPE ENUM)</field>
  <field type="uint8_t" name="autopilot">Autopilot type / class. defined in MAV_CLASS
      ENUM</field>
  <field type="uint8_t" name="base_mode">System mode bitfield, see MAV_MODE_FLAGS
      ENUM in mavlink/include/mavlink_types.h</field>
  <field type="uint32_t" name="custom_mode">Navigation mode bitfield, see
      MAV_AUTOPILOT_CUSTOM_MODE ENUM for some examples. This field is autopilot-
      specific.</field>
  <field type="uint8_t" name="system_status">System status flag, see MAV_STATUS
      ENUM</field>
  <field type="uint8_t_mavlink_version" name="mavlink_version">MAVLink version</field>
</message>
```

Message ID:
0=Heartbeat

Field 1

Field 2

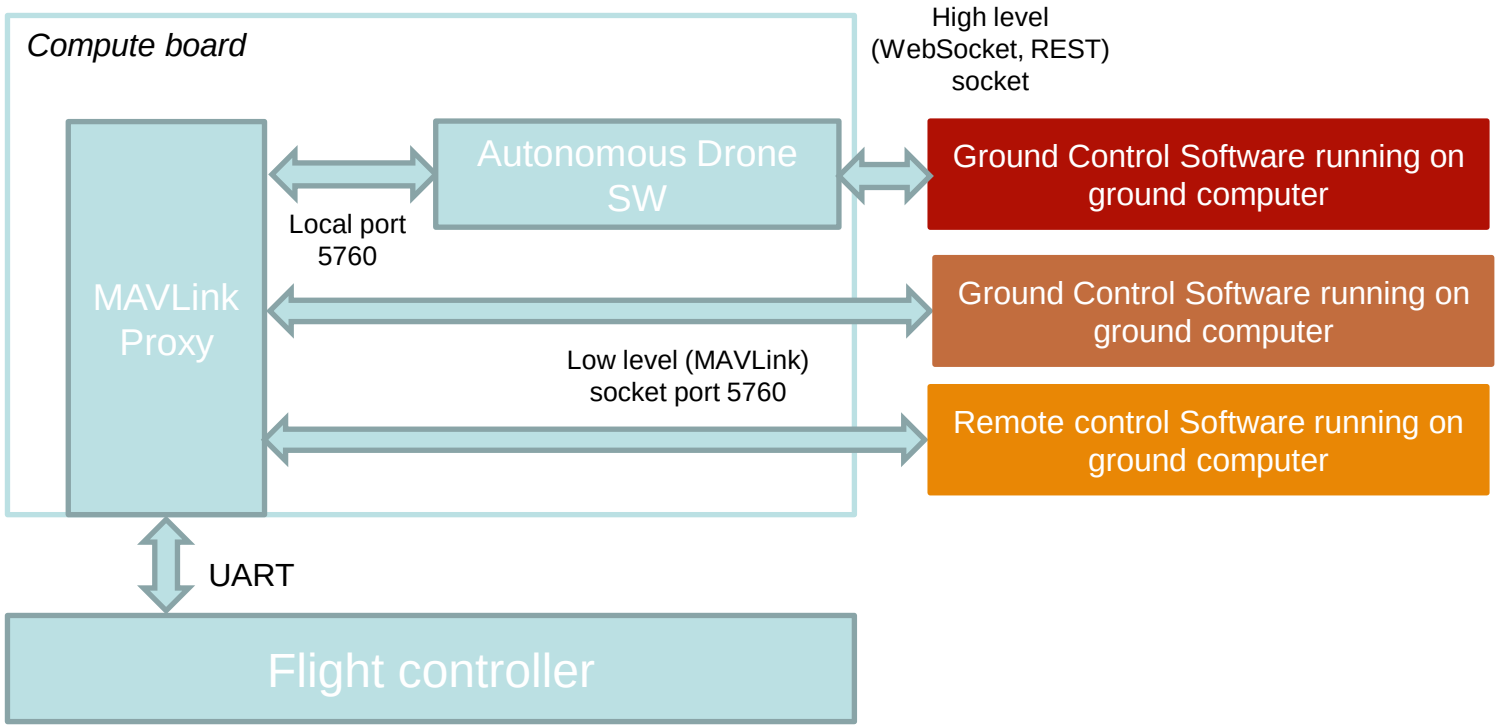
.
.
.

Field n

MAVLink Message Types Examples

- All MAVLink message types 0-150 already defined
- Message ids 0 – 149 are common for all autopilots
  - <message id="0" name="HEARTBEAT">
  - <message id="11" name="SET_MODE">
  - <message id="24" name="GPS_RAW_INT">
  - <message id="41" name="MISSION_SET_CURRENT">
  - <message id="42" name="MISSION_CURRENT">
  - <message id="46" name="MISSION_ITEM_REACHED">
  - <message id="47" name="MISSION_ACK">
  - <message id="76" name="COMMAND_LONG">
  - <message id="77" name="COMMAND_ACK">
  - <message id="147" name="BATTERY_STATUS">
- Message ids 150-250 are autopilot specific, or custom

MAVLink Proxy



Writing your own program

- There are several ways to write a program interfacing with the flight controller:
  - Write on the top of MAVLink using one of the MAVLink implementations in C/C++, Java or Python (Es. pymavlink)
  - Use one of the framework for MAVLink abstraction and interface:
    - DroneCore, http://dronecore.io
    - Module for ROS, the Robotic Operating System, http://wiki.ros.org/mavlink
    - Dronekit, http://dronekit.io

Pymavlink example program

```
#!/usr/bin/python
from __future__ import print_function

import pymavlink.mavutil as mavutil
import sys
import time

mav = mavutil.mavlink_connection('tcp:127.0.0.1:5760')
mav.wait_heartbeat()
mav.mav.command_long_send(mav.target_system, mav.target_component,

mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM, 0, 1,
                          0, 0, 0, 0, 0, 0)
time.sleep(3)
mav.mav.command_long_send(mav.target_system, mav.target_component,
                          mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM, 0, 0,
                          0, 0, 0, 0, 0, 0)
```

Arms the motors

Waits 3 seconds

Disarms the motors

Acquiring sensor data by pymavlink

```
from __future__ import print_function

import pymavlink.mavutil as mavutil

mav = mavutil.mavlink_connection('tcp:192.168.8.1:5760')
mav.wait_heartbeat()
mav.mav.request_data_stream_send(mav.target_system, mav.target_component,
                                 mavutil.mavlink.MAV_DATA_STREAM_ALL, 4, 1)

while(True):
    msg = mav.recv_match(blocking=False)

  msg_type = msg.get_type()
    if msg_type == "ATTITUDE:
        print('%0.2f\t%0.2f\t%0.2f' % msg.roll, msg.pitch, msg.yaw)
```

ROS - Robot Operating System

It may be tricky to integrate several libraries like MAVLINK, VISP, RealSense together. **ROS** is a way to unify the interfaces and simplify the integration of components coming from various sources.

ROS is not an operating system, it's a stack running on top of the Linux OS (Yocto or Docker-Ubuntu in our case).

ROS has modules such as:

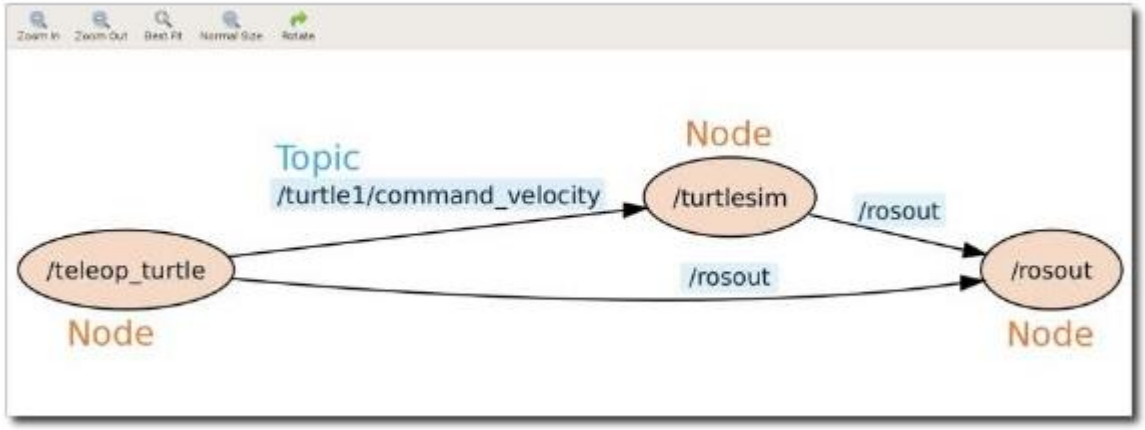
- MavROS for MAVLINK http://wiki.ros.org/mavros
- VISP http://wiki.ros.org/visp
- OpenCV http://wiki.ros.org/vision
- RealSense http://wiki.ros.org/RealSense

ROS Philosophy

- Peer to Peer
  - ROS systems consist of many small programs (nodes) which connect to each other and continuously exchange messages
- Tools-based
  - There are many small, generic programs that perform tasks such as visualization, logging, plotting data streams, etc.
- Multi-Lingual
  - ROS software modules can be written in any language for which a client library has been written. Currently client libraries exist for C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, and more.
- Thin
  - The ROS conventions encourage contributors to create stand-alone libraries/packages and then wrap those libraries so they send and receive messages to/from other ROS modules.
- Free & open source, community-based, repositories

ROS BASICS

- ROS offers a message passing interface that provides inter-process communication.
- A ROS system is composed of nodes, which pass messages, usually in two forms:
  - ROS messages are published on topics and are may-to-many
  - ROS services are used for synchronous request/response

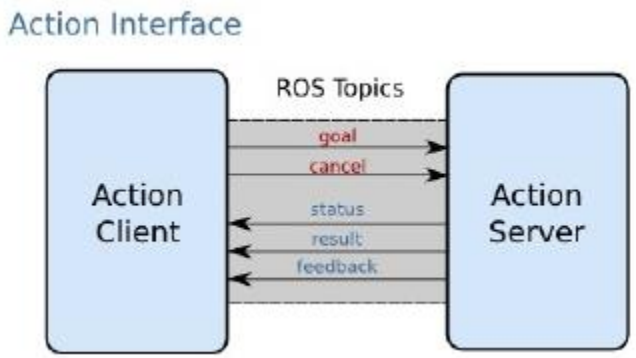


SERVICES

- Services allow one node to call a function that executes in another node
- The server node which provides the service specifies a callback to deal with the service request and advertises the service
- The client node which calls the service then accesses this service through a local proxy
- Similar to Java RMI mechanism

ACTIONS

- Actions are used in case when the time required for a function to return a value is undetermined
- Actions are implemented using three different topics namely goal, result and feedback. So it is essentially a higher level protocol that determines how these topics should interact



MAVROS

MAVROS is a MAVLink extendable communication node for ROS with UDP proxy for Ground Control Station.

```
#!/usr/bin/env python
import rospy from
sensor_msgs.msg import Imu

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "\nlinear acceleration:\nx: [{}]\ny: [{}]\nz: [{}]"
    .format(data.linear_acceleration.x, data.linear_acceleration.y, data.linear_acceleration.z))

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("/mavros/imu/data", Imu, callback)
    rospy.spin()

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

Prints linear acceleration values as soon as they have been received

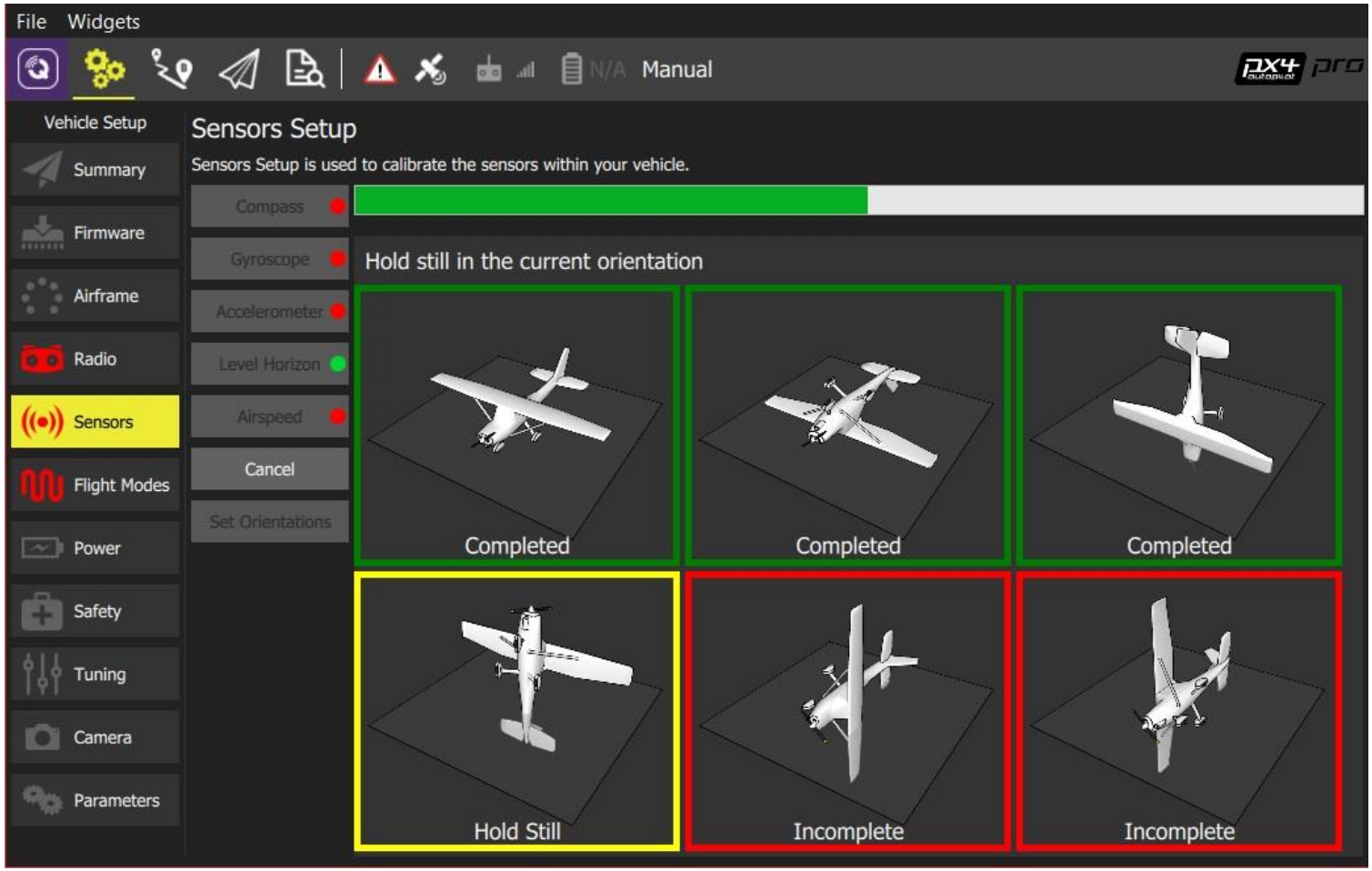
Declares that the node subscribes to the topic of type sensor_msgs.msgs.Imu

Keeps the node from exiting until it has been shutdown

QgroundControl

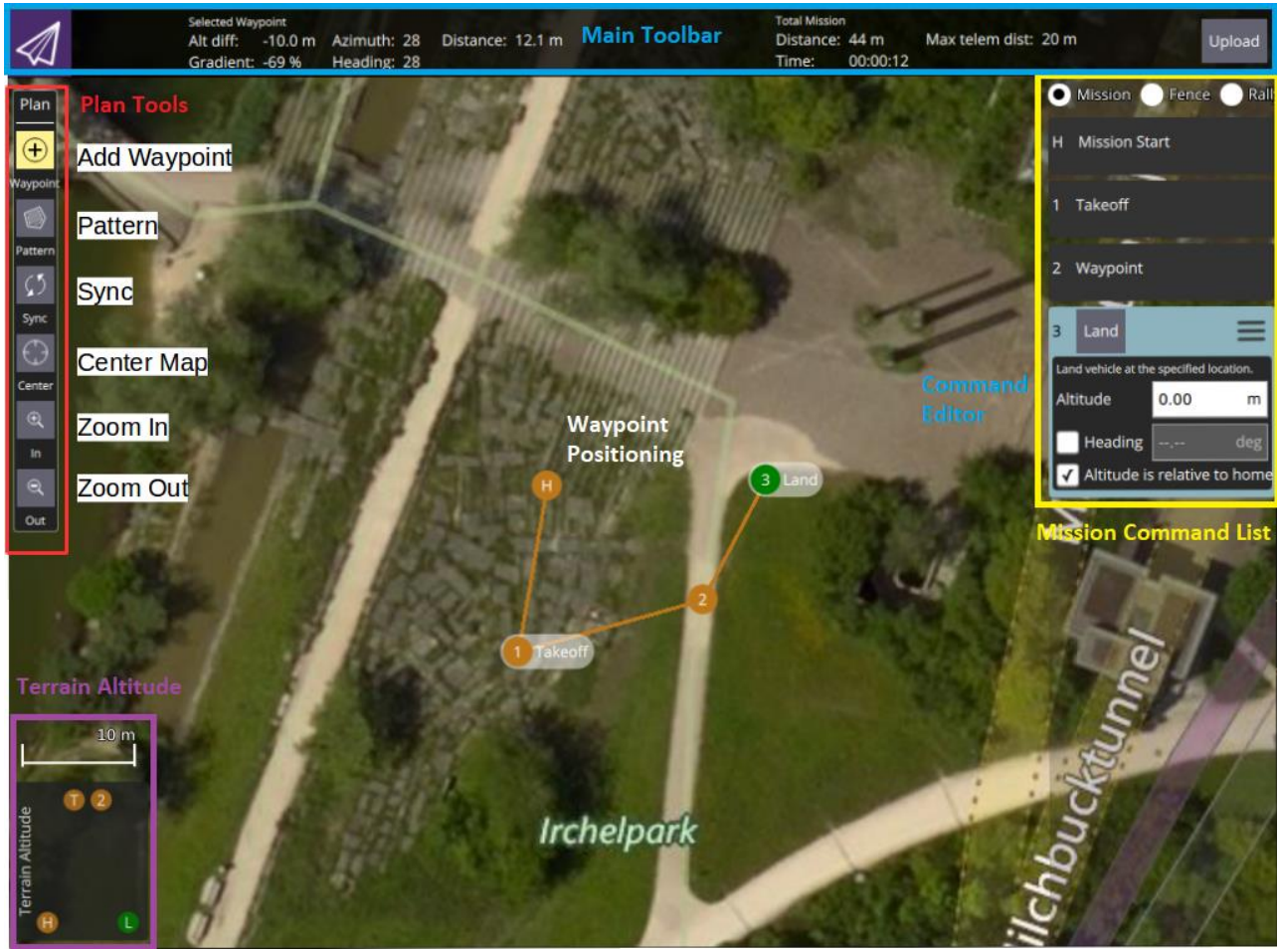
QGroundControl provides full flight control and vehicle setup for PX4 or ArduPilot powered vehicles. It provides easy and straightforward usage for beginners, while still delivering high end feature support for experienced users.

QGroundControl: Sensor calibration



Prof. Francesco Picariello, Ph.D.

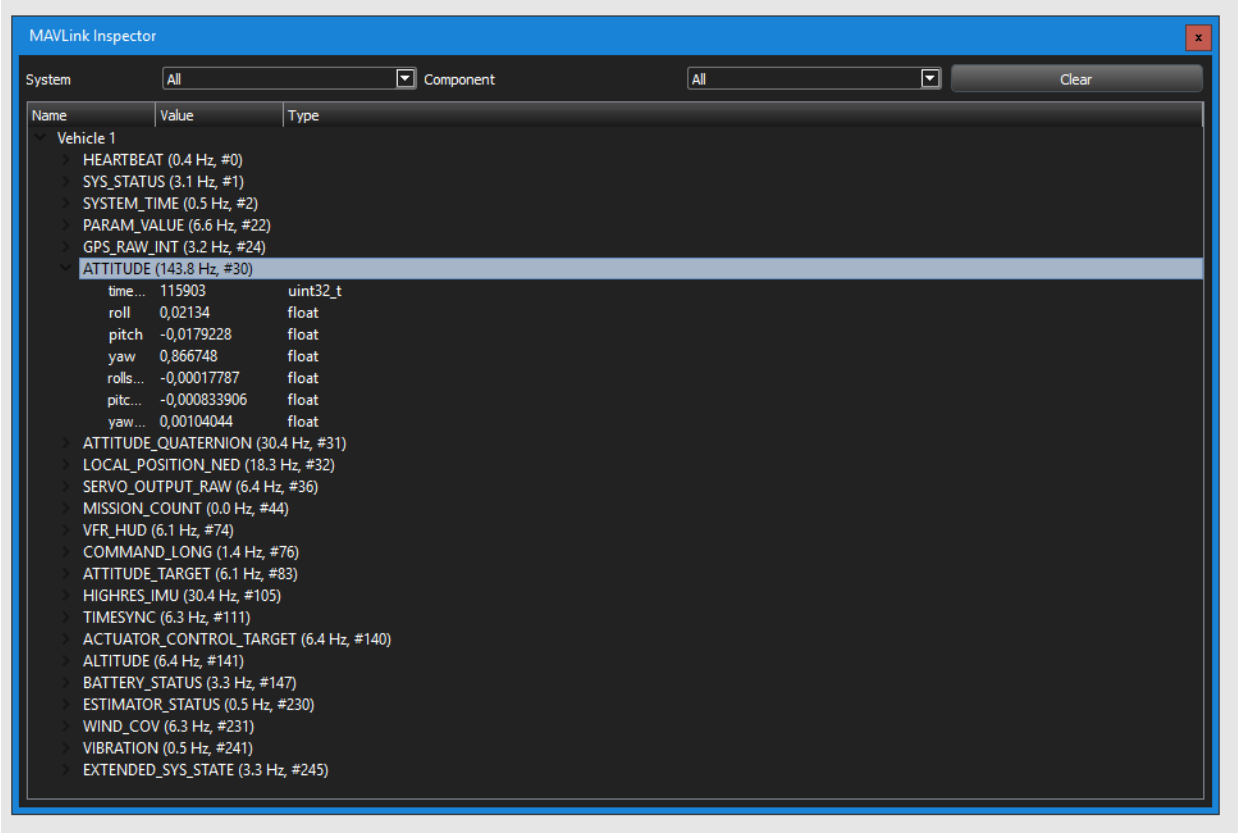
QgroundControl: Mission Plan



Prof. Francesco Picariello, Ph.D.

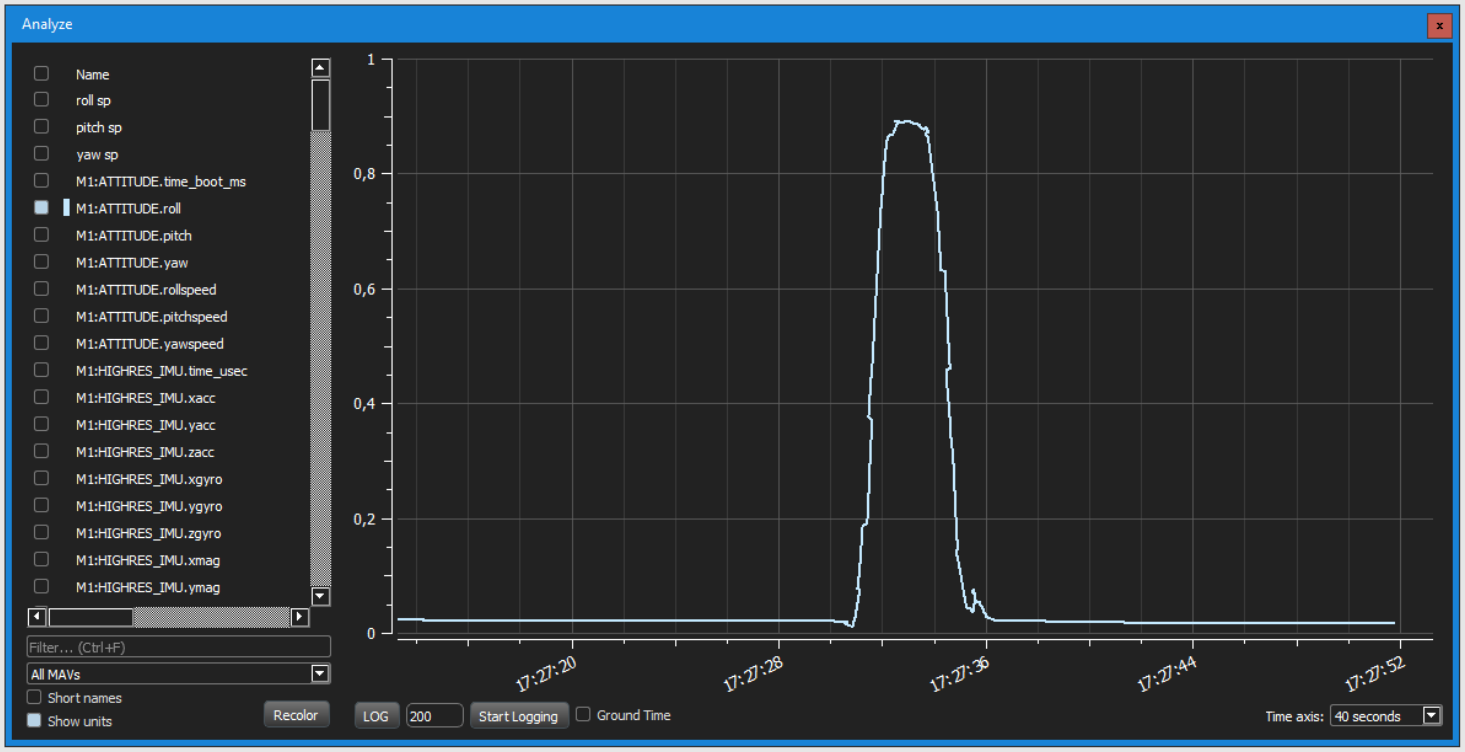
Didactic Module
2-8 July 2018

Mavlink inspection



Prof. Francesco Picariello, Ph.D.

Sensor data monitoring by Qgroundcontrol



Prof. Francesco Picariello, Ph.D.

Didactic Module
2-8 July 2018