NATIONAL UNIVERSITY OF HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FALCUTY OF TRANSPORTATION ENGINEERING

DEPARMENT OF AEROSPACE ENGINEERING

-------o0o-------

Report #2

MAVLink

Communication protocol for Unmanned Vehicle

Student:	Nguyễn Anh Quang
Student ID:	V1002583
Supervisor:	Prof Emmanuel GROLLEAU

Poitier, Spring 2015

Abstract

This report is a part of the complete report for the Project called" Implementation a flight command for a Quadricopter".

In this report, MAVLink protocol will be presented, its anatomy, its implementation in ArduPilot, etc. This report will also indicate some available Ground Control Station at this time using MAVLink protocol, and then choose the one that is suitable for the project.

In the contribution of this report, the implementation of MAVLink in a developed Ground Control Station will be introduced.

Mục lục - Table of Contents

Danh mục hình ảnh - List of Figures

Danh mục bảng - List of Tables

Chương 1

MAVLink protocol

MAVLink Micro Air Vehicle Communication Protocol is a common protocol for communication between Unmanned Vehicles and Ground Control Station or between Vehicles themselves.

1.1 Overview

1.1.1 General Communication Protocol

"MAVLink was first released early 2009 by Lorenz Meier under LGPL license" [**1**] as an effort to standardize the communication usied for autopilot systems. MAVLink has some advantages compared to other protocols, which make MAVLink one of the most popular protocol to use for autopilot communication at this time.

In order to implement this protocol into an autopilot project, the programmer just needs to add the *.h files (header files) prepared in C/C++ structs defining the meaning of messages into their project and then creates some functions to send these messages via a serial port. As long as the output messages have the right syntax, any developed Ground Control Station (GCS) using MAVLink protocol can understand them.

At this time, MAVLink provides sample messages for almost every needed situation, from basic control to camera transmitting, from calibrating system before taking action to flight plan writing and reading. Moreover, MAVLink is open-source, gives users the ability to modify its messages and to create new messages suitable for any desired purposes.

As an effort to standardize the communication of autopilot vehicles, MAVLink has been supported by many developing GCSs projects, which can be used not only in special GCS systems but also in computers, laptops and even mobile phones. Some of the GCSs for laptop will be presented in the next part of this report to suggest the most suitable one for this project. Additionally, MAVLink is also the communication protocol of autopilot projects such as ArduPilot since it is lightweight, available in C/C++ language, supports creating library files for Python and C# projects, hardware independently, etc. More importantly, MAVLink does not depend on the output serial port, wire or wireless. It can handle messages from any kind of communication, UDP, TCP, USB, Bluetooth, and even console channel, which is used in simulation such as Software In The Loop (SITL) as well as Hardware In the Loop (HIL).

1.1.2 MAVLink packet anatomy.

Figure 1-1 gives the general idea of using MAVLink in an autopilot systems and Ground Control Station. The implementation of this protocol includes two separated objects. Firstly is the work in autopilot system. MAVLink protocol is nothing but definitions of numbers, number sequence and the order to transfer them from system to system. It does not affect the main code or the main structure of the control code, which related to calculations and control variables. While implementation the communication system, programmer will add the library files containing the meaning of each number in messages and then procedure to handle these numbers. In C/C++, these files are *.h files as described in ArduPilot.

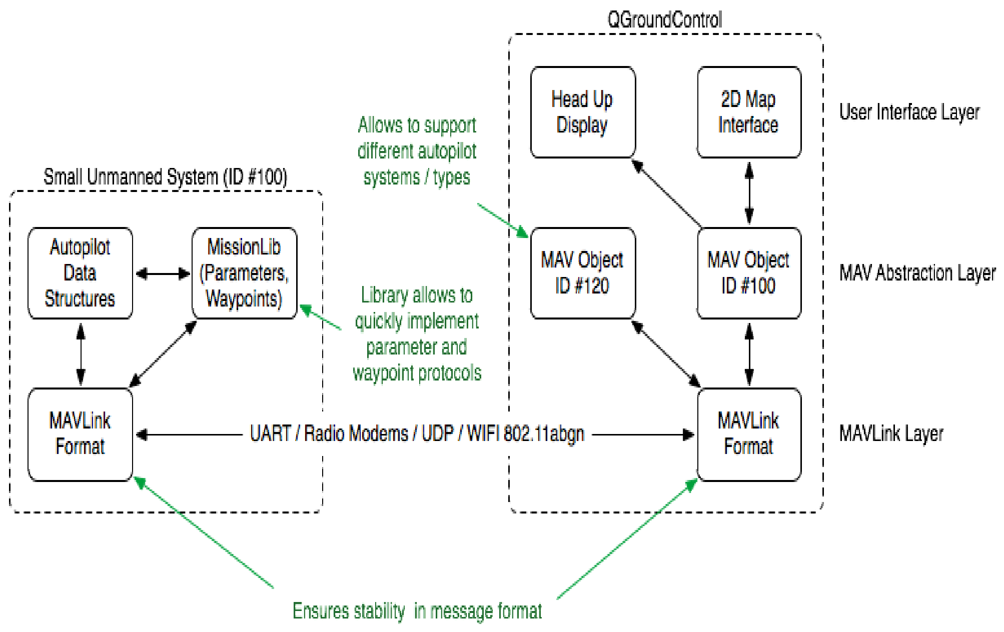

QGroundControl
Head Up Display
2D Map Interface
User Interface Layer
Allows to support different autopilot systems / types
Small Unmanned System (ID #100)
Autopilot Data Structures
MissionLib (Parameters, Waypoints)
MAV Object ID #120
MAV Object ID #100
MAV Abstraction Layer
Library allows to quickly implement parameter and waypoint protocols
MAVLink Format
UART / Radio Modems / UDP / WIFI 802.11abgn
MAVLink Format
MAVLink Layer
Ensures stability in message format

Figure 1-1: Communication between Unmanned System and GCS [2]

Secondly, it is the implementation in Ground Control Station. **Figure 1-1** introduces the architecture of QGroundControl, and as can be seen, its structure is multi-layer. This architecture provides this application the ability to control multiple different autopilot systems simultaneously. The lowest layer is the MAVLink layer with the libraries files, MavLink format and will check the correctness of the received messages, which can be transferred by any protocol. After the confirmation of this layer, the message will passed the message to the MAV abstraction layer. The system ID, which includes in each message, will separate each system with others and GCS will handle each of them separately. The highest layer includes commands and functions to interact with users.

Figure 1-2 presents the general syntax of a MAVLink message. Meanwhile, **table 1** gives more information about the figure.

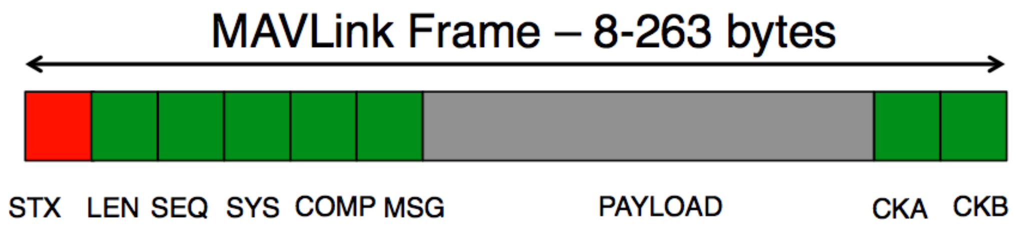
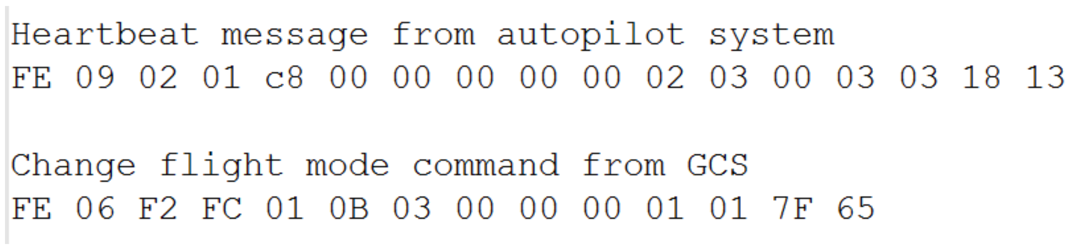


Figure 1-2: MAVLink general syntax [1]

Table 1: Packet Anatomy [1]

Frame	**Byte Index**	**Content**	**Value**	**Explanation**
STX	0	Packet start sign	v1.0: 0xFe v0.9: 0x55	Signal of a new package
LEN	1	Payload length	0-255	Length of the payload part
SEQ	2	Packet sequence	0-255	Number presenting the order of package, used to detect package loss
SYS	3	System ID	1-255	ID of the sending autopilot system.
COMP	4	Component ID	0-255	ID of the component of the sending system
MSG	5	Message ID	0-255	ID of the message, give the key to decode the package
PAYLOAD	6 to (n+6)	Data	(0-255) byte	Sent data
CKA	n+7	Checksum	SAE-AS4 standard [**3**]	
CKB	n+8			

Two examples for this packet anatomy can be seen in **figure 1-3** as well as the explanation of the heartbeat message in **table 2.**



```
Heartbeat message from autopilot system
FE 09 02 01 c8 00 00 00 00 00 02 03 00 03 03 18 13

Change flight mode command from GCS
FE 06 F2 FC 01 0B 03 00 00 00 01 01 7F 65
```

Figure 1-3: Some messages with MAVLink packet anatomy

Table 2: Detail of a heartbeat message

Group	Byte value	Frame	Explanation (decimal value)
Heading	FE	STX	Default value for MAVLink version 1.0 (254)
	09	Payload length	Payload of this message has 9 bytes
	02	Message sequence	Second message from this system
	01	System ID	Name of this system (1)
	C8	Component ID	IMU component (200)
	00	Message ID	Heartbeat message ID (0)
Data	00 00 00 00 02 03 00 03 03	Payload	- Custom mode (0 0 0): Preflight, not armed - Type (2) : Quadricopter - Autopilot (3): ArduPilotMega - Base mode (0) - System Status (3) : Standby - Version (3) : Version of ArduPilot
Checksum	18	CKA	Large checksum byte (24)
	13	CKB	Small checksum byte (13)

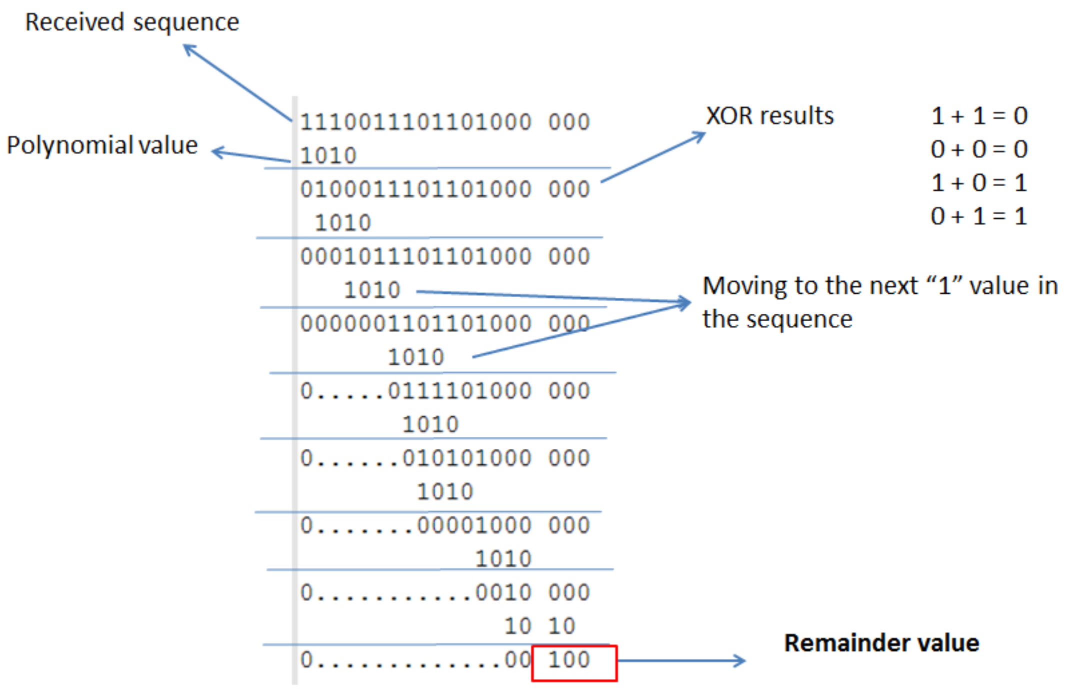
The calculation of the checksum value will be presented in the next part of this report. From the figures and the tables above, it can be seen that the shortest MAVLink packet has a length of 8 bytes, meanwhile the longest one can reach 263 bytes. Moreover, by adding System ID and Component ID, MAVLink can separate not only different autopilot systems, which give it the ability to communicate with multiple systems at one time, but also different components in one system. The packet anatomy of MAVLink also provides the method to detect loss packet in transmitting, which is crucial for a GCS to check the connection with the system and for an autopilot system to trigger a failsafe process when it disconnects from the controller.

1.1.3 MAVLink checksum calculation

1.1.3.1 Overview about Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) is a very popular method used to detect error code implemented in many data transmission schemes [**4**]. By adding this checksum value at the end of every transmitting packet, for example the heartbeat packet mentioned above, receiver can check if the received message misses any frame during the transmitting process. Moreover, because of the variety of method to calculate this checksum, it can also be used as a tool to separate messages for different applications.

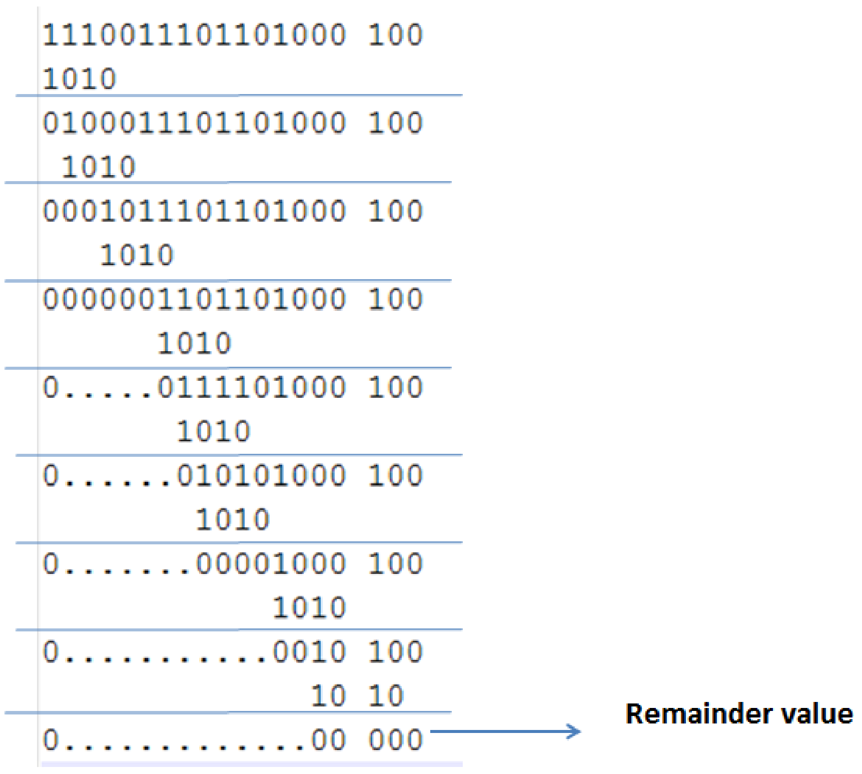
Cyclic Redundancy Check uses the **binary sequence** of the received message and the **polynomial number** to calculate the **remainder value** after several divisions. For example, if the receiver receives the message as [1110 0111 0110 1100], the polynomial is [1010] and we are waiting for a 3-bit CRC, the calculation for the remainder should be as described in **figure 1-4**. As can be seen, the method used in this calculation is XOR (exclusive OR), where 1 + 1 = 0; 0 + 0 = 0; 1 + 0 = 0 + 1 =1.



```
1110011101101000 000
1010
0100011101101000 000
 1010
0001011101101000 000
   1010
0000001101101000 000
      1010
0.....0111101000 000
       1010
0......010101000 000
        1010
0.......00001000 000
            1010
0..........0010 000
              10 10
0............00 100
```

Figure 1-4: A simple Checksum calculation

Now if we replace the reserved place at the right with the true **Remainder Value** (checksum value), the result would be zero:



```
1110011101101000 100
1010
0100011101101000 100
 1010
0001011101101000 100
   1010
0000001101101000 100
      1010
0.....0111101000 100
       1010
0......010101000 100
        1010
0.......00001000 100
            1010
0..........0010 100
              10 10
0............00 000
```

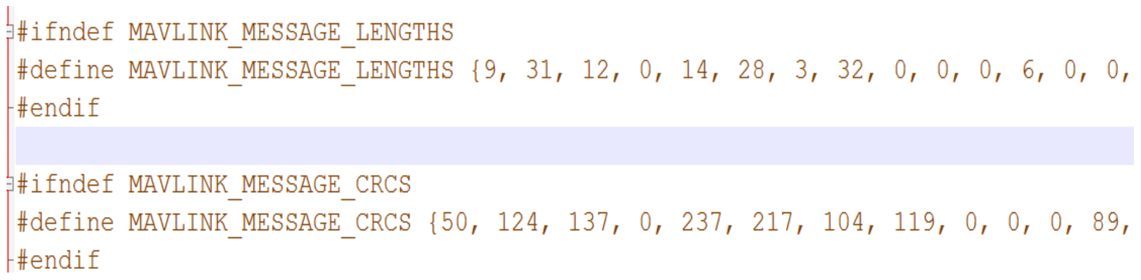
Figure 1-5: Remainder value is Zero as message included CRC

When the message including the CRC is transferred and then loses a bit during transmitting due to noise or other reasons, the receiver will notice the difference by encoding the message and find the remainder value. Nowadays, with the development of telecommunication and transmitting protocol, there are various standards for checksum such as CRC-4, CRC-8, CRC-16, CRC-32, etc. Each of them has their own usages and applications. More details about these checksum calculations can be seen in [**3**] and [**4**].

1.1.3.2 MAVLink CRC calculation

MAVLink uses CRC-16-CCITT standard, which is used in various communication protocol such as X.25, Bluetooth, etc [**3**]. The result of this checksum calculation is a 16 bits remainder value, this value is then put into the packet as CRCs as shown above. For example, the checksum value for the heartbeat message in figure 1-3 after calculating is [0001 1000 0001 0011] in binary, which is 1813 in hexadecimal. This value is then transfer as 18 in CKA and 13 in CKB.

However, from version 1.0, MAVLink developers have made a small change in the calculation of the checksum value. In order to avoid the phenomenon that *"two devices using different message versions incorrectly decode a message with the same length"* [**5**], which occasionally caused problems in version 0.9, an extra CRC calculation step is added into this protocol. As they mentioned in [**5**], now every message will have an extra constant beside the message ID called "the seed". After calculating checksum for every byte in the packet, MAVLink will have one more calculation with this "seed", and then the value of CKA and CKB will be this value. These seeds are defined in the library of MAVLink, for instances, **figure 1-6** introduces the seeds in Ardupilotmega.h. Because Heartbeat message ID is 0, its seed would be 50 as can be seen



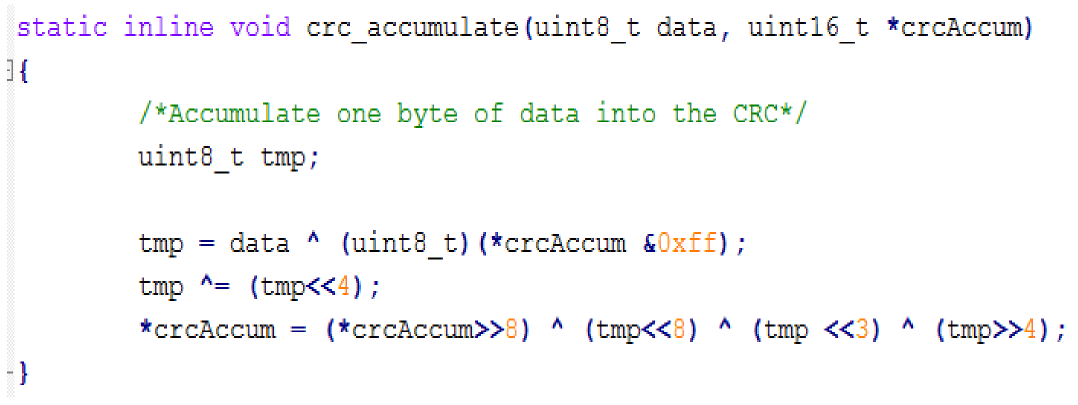
```
#ifndef MAVLINK_MESSAGE_LENGTHS
#define MAVLINK_MESSAGE_LENGTHS {9, 31, 12, 0, 14, 28, 3, 32, 0, 0, 0, 6, 0, 0,
#endif

#ifndef MAVLINK_MESSAGE_CRCS
#define MAVLINK_MESSAGE_CRCS {50, 124, 137, 0, 237, 217, 104, 119, 0, 0, 0, 89,
#endif
```

Figure 1-6: Seeds for extra CRC calculations in Ardupilotmega.h

1.2 Supporting functions in MAVLink library

1.2.1 Overview

Beside the definitions for messages, MAVLink library also provide users with some useful functions to implement it into different projects. A simple example for these functions is the checksum calculation, it can be found in **checksum.h** as shown below.



```
static inline void crc_accumulate(uint8_t data, uint16_t *crcAccum)
{
        /*Accumulate one byte of data into the CRC*/
        uint8_t tmp;

        tmp = data ^ (uint8_t)(*crcAccum &0xff);
        tmp ^= (tmp<<4);
        *crcAccum = (*crcAccum>>8) ^ (tmp<<8) ^ (tmp <<3) ^ (tmp>>4);
}
```

Figure 1-7: Function to calculate checksum in MAVLink library

These supporting functions are provided in C/C++ language; however, there are tools to convert this library into Python, C# and Java. **Figure 1-8** and **table 3** gives a summary about these tools.

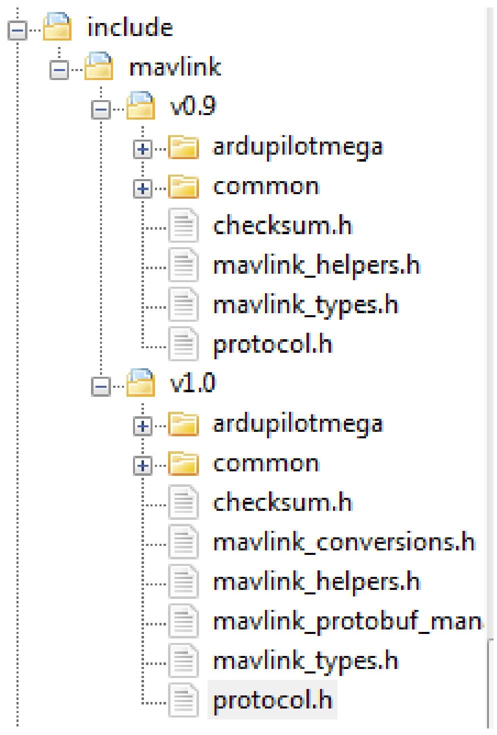


Figure 1-8: Files contain supporting functions in MAVLink v0.9 and v1.0

Table 3: Supporting functions in MAVLink library

Function group	**Reference files**	**Functions**	**Application**
Messages handling group	mavlink_helpers.h	mavlink_parse_char	Convenient function to handle the received messages
Messages sending group	mavlink_helpers.h protocol.h	mavlink_finalize_message_chan	Convenient function to send message and calculate checksum
		mavlink_finalize_message	Convenient function to send message and calculate checksum
		_mavlink_send_uart	Send message
		_mav_finalize_message_chan_send	Convenient function to send message and calculate checksum

		_mavlink_resend_uart	resend a message in the same channel
		mavlink_msg_to_send_buffer	Pack a message and send it via a byte stream
		mavlink_msg_get_send_buffer_length	Get the required buffer size for this message
Channels handling group	protocol.h mavlink_helpers.h	mavlink_get_channel_status	Check the status of the channel (number of available free space)
		mavlink_get_channel_buffer	Check the buffer status in the channel (number of available free space)
		mavlink_reset_channel_status	Reset the status of the channel
Checksum group	mavlink_helpers.h checksum.h	mavlink_start_checksum	Return the init value 0xffff
		crc_init	Return the init value 0xffff
		crc_calculate	Return the checksum over a buffer bytes
		crc_accumulate	Return the checksum over a transferred byte
		mavlink_update_checksum	Return the checksum over a transferred byte
		crc_accumulate_buffer	Return the checksum over an array of bytes
Buffer handling group	protocol.h	mav_array_memcpy	Copy the values from the pointed location to the destination (buffer)
		_mav_put_char_array	Place a char array into a buffer
		_mav_put_int8_t_array	Place a int8_t

			array into a buffer
		_MAV_RETURN_char_array	Get a char from buffer
		_MAV_RETURN_uint8_t_array	Get a uint8_t from buffer
		_MAV_RETURN_int8_t_array	Get a int8_t from buffer
Values conversion group	mavlink_conversions.h	mavlink_quaternion_to_dcm	Convert a quaternion to a rotation matrix
		mavlink_dcm_to_euler	Converts a rotation matrix to euler angles
		mavlink_quaternion_to_euler	Converts a quaternion to euler angles
		mavlink_euler_to_quaternion	Converts euler angles to a quaternion
		mavlink_dcm_to_quaternion	Converts a rotation matrix to a quaternion
		mavlink_dcm_to_quaternion	Converts a rotation matrix to a quaternion

1.2.2 Working procedure of a supporting function

As can be seen, MAVLink library provides developers with many convenient functions to speed up process of establishing a communication system with MAVLink protocol. Among them, **mavlink_parse_char** and can be considered as one of the most important function. Knowing how it work will understand the listening structure of MAVLink and simplify the process of integrate MAVLink protocol into a project.

Table 4: mavlink_parse_char function

Function	**mavlink_parse_char** (uint8_t chan, uint8_t c, mavlink_message_t* r_message, mavlink_status_t* r_mavlink_status)		
Inputs	**uint8_t**	chan	Channel receives the message
	uint8_t	c	character (byte) to parse
	mavlink_message_t*	r_message	packet anatomy
	mavlink_status_t*	r_mavlink_status	status flag to control the function
Outputs	**uint8_t**	return 0 if no message could be decoded and 1 in the other case	

When a byte value is read by the serial port, **mavlink_parse_char** will be called to decide what the system should do next. The general idea about this function can be explained by **figure 1-9.**

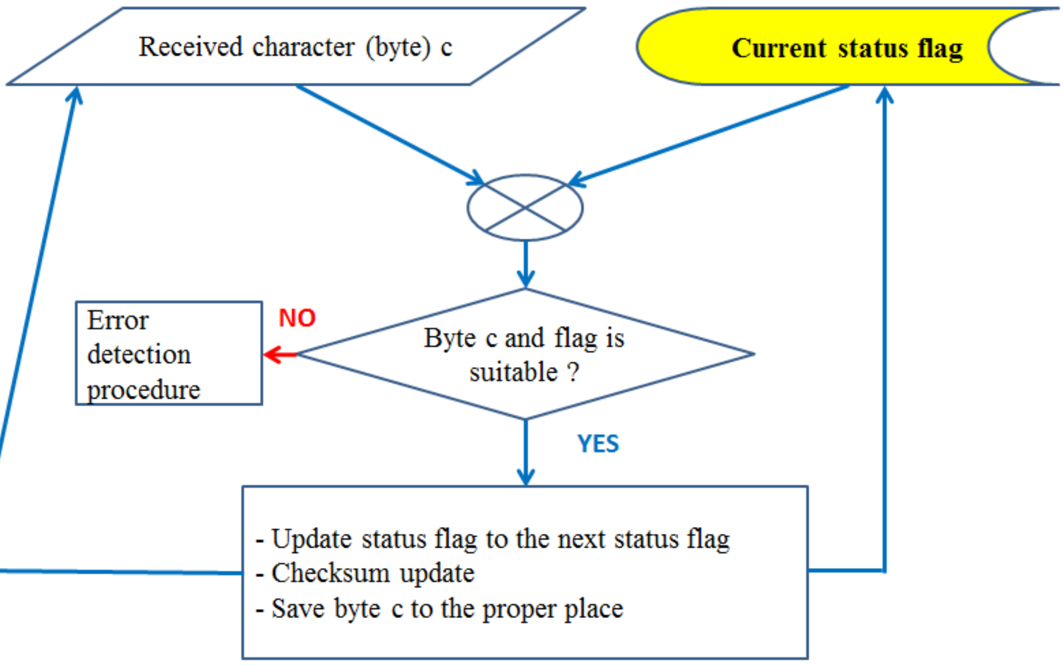

Received character (byte) c
Current status flag
Error detection procedure
NO
Byte c and flag is suitable ?
YES
- Update status flag to the next status flag
- Checksum update
- Save byte c to the proper place

Figure 1-9: mavlink_parse_char general procedure

After receiving a byte, depending on the status flag, the control system will decide what to do with that byte. If the byte and the flag are compatible, the handling process for that flag will be triggered. Although this procedure has some small differences based on the flag, it always includes three main functions: **update the flag**, which will be suitable for the next coming byte; **update checksum**; **save received byte** to memory. After this procedure, mavlink_parse_char is finished and then go back waiting for the next byte. For instance, **figure 1-10** describes the procedure when the STX value comes.

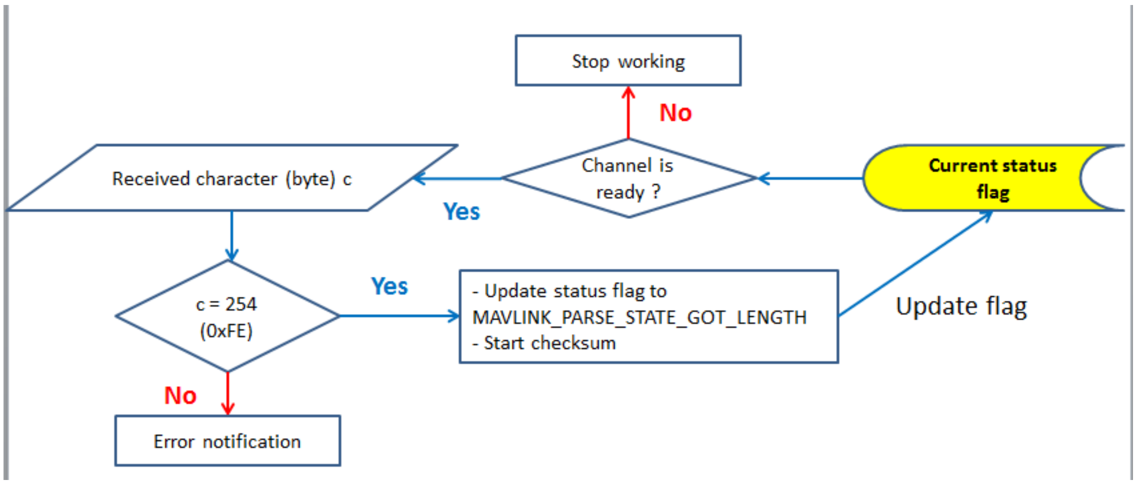


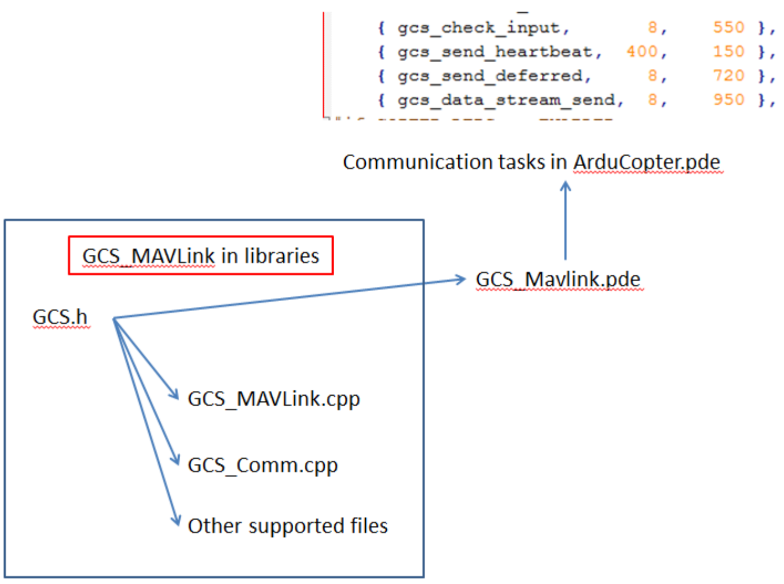
Figure 1-10: A detail procedure for mavlink_parse_char

In **figure 1-10**, MAVLINK_PARSE_STATE_GOT_LENGTH is the next status flag, which will trigger the process handling the next incoming LEN frame. As can be seen, programmer only need to generate function to read the message and then call this function to handle the message, byte by byte, every received frame will be put in right place and waiting for the handling process.

1.3 Utilisation of MAVLink in ArduPilot

1.3.1 Tasks and files

In general, the communication with GCS in ArduPilot is implemented as in **figure 1-3**. In the main pde file of every vehicle directory, ArduCopter.pde for example, there are some tasks for communication. These tasks are put into threading as mentioned before. In this case, among the four, **gcs_send_heartbeat** and **gcs_check_input** are the most important. A heartbeat message is a special message keeping the connection between the autopilot and GCS alive. The GCS will send signals to autopilot to check his heartbeat repeatedly and wait for its response. In case it does not receive the heartbeat, GCS will alarm "Connection lost", even though it still receives other messages from autopilot. With this alarm, the UAV will automatically trigger the "failsafe" procedure and will land down.



```
{ gcs_check_input,        8,    550 },
{ gcs_send_heartbeat,   400,    150 },
{ gcs_send_deferred,      8,    720 },
{ gcs_data_stream_send,   8,    950 },
```

Figure 1-11: Software architecture of MAVLink in ArduPilot

gcs_check_input is the listening and responding task, which is the foundation for any communication with the GCS, this is also the task where the **mavlink_parse_char** function above is called. In order to save time for other tasks which are more important such as reading sensors or calculating rotors output, beside heartbeat message as mentioned above, other messages will only be sent when GCS asks for them. The other two tasks, **gcs_send_deffered** and **gcs_data_stream_send** are supporting tasks for GCS communication. More details of the communication sequence will be discussed in the next part of this report.

ArduPilot communication with GCS is implemented by three files. **GCS_Mavlink.pde** in ArduCopter directory contains the specific commands for multicopter vehicles; two ***.cpp** files in GCS_MAVLink in the libraries directory include functions can be used for any kind of vehicle. Other related files such as the definition of Mavlink packages or tracking message route are also in this Library.

1.3.2 Communication procedure

1.3.2.1 General procedure

The procedure for sending and responding a message of autopilot system in ArduPilot can be described as in **figure 1-4.**

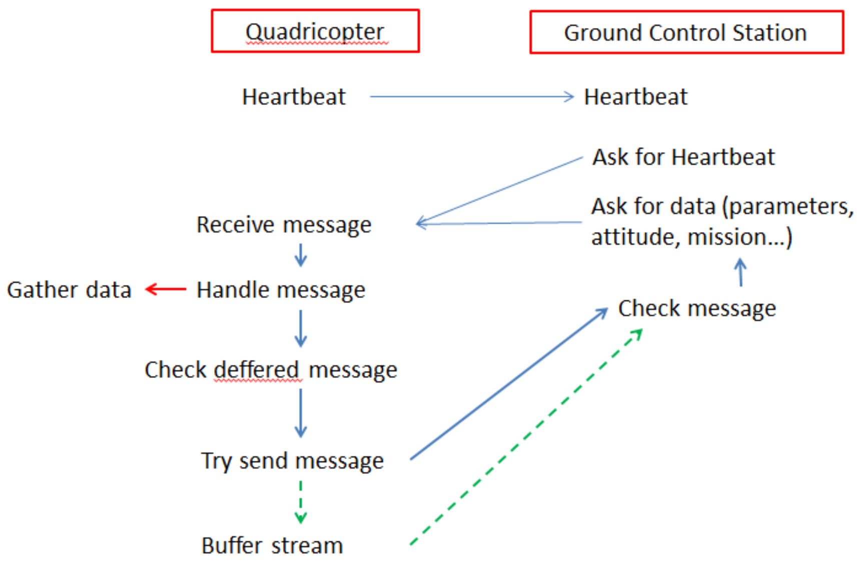


Figure 1-12: Communication procedure

As mentioned, heartbeat is a special message, therefore, it will be sent individually by the autopilot as well as will being asked regularly by the GCS. Other data will be sent when it is required by the GCS, some important data such as attitude, parameters, GPS… will also be asked repeatedly in time. Other messages like changing control values, reading/writing flight plane, flight mode… will be sent when user interact with the GCS. After **receiving the message** from GCS, autopilot systems will **handle it and gather data** from memory or latest read values in case of sensors. Autopilot system will then **check deffered message**, a process in which autopilot will check if the message has been sent recently to prevent responding a message multiple times or if the bandwidth is full. After that, autopilot will **try to send** the message, avoiding interrupting sending important messages by lower priority messages. In ArduPilot, there are two ways to send a message. The first way is the simple way using the basic write method of each controller board and send message as individual character. The second way is called “data stream”, which will be used in case of a big amount of data need to be sent immediately. In this method, all data will be sent in a stream all at one. For example, in a transferring control constants test between autopilot system and GCS via wifi, using data stream will at least 10 times faster than the simple way (4.86 second in data stream and 57.1 second without data stream). This duration is extremely important since when an autopilot system is activated, it cannot use all of its bandwidth just for sending and receiving parameters.

1.3.2.2 Specific procedure

With each messages, there will be a specific procedure for them. For example, **figure 1-5** describes the reading mission protocol between GCS and MAV component.

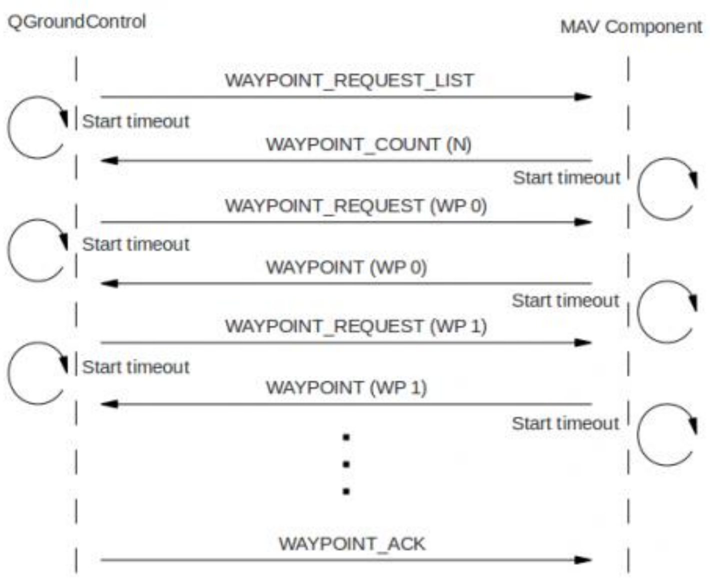


Figure 1-13: Reading mission protocol [6]

In this case, since many messages have to be sent and received by both components, there is an **ack message** at the end of the protocol. This is also a special message, which will be used to confirm that the protocol is finished and next protocol can be called. In some case, an ack message will contain the result of a received command.

More details about specific protocols can be found in [**1**].

Chương 2

Comparison of different ground control station using MAVLink

2.1 Ground Control Stations using MAVLink protocol.

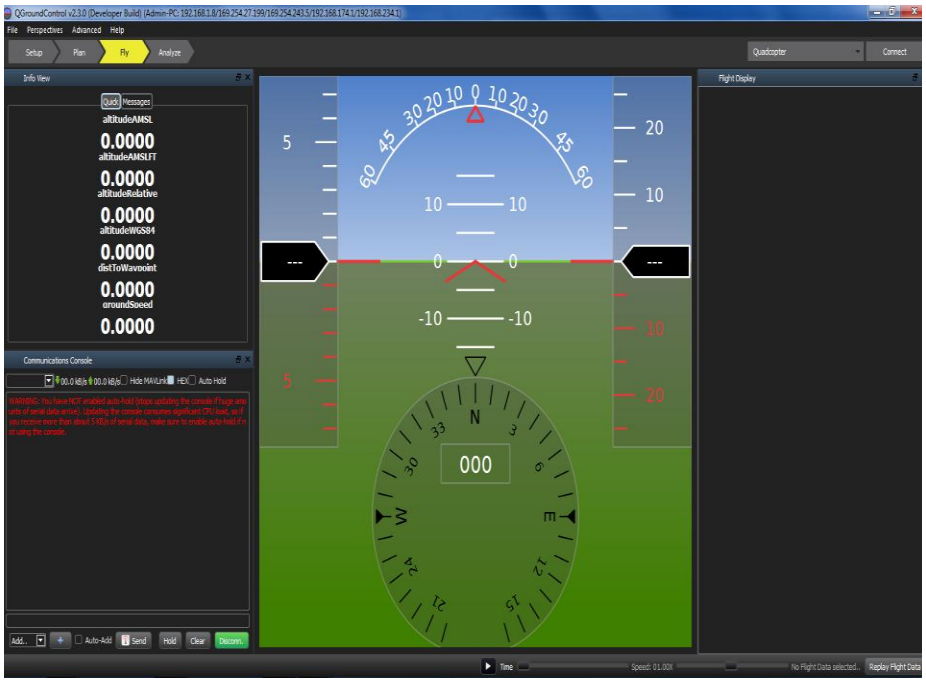
As mentioned, MAVLink protocol has become popular and has been used worldwide in several GCS. In this part of the report, three different GCS using MAVLink protocol will be presented, as well as their pros and cons. From these discussions, this report will point out the most suitable GCS for using in this project.

2.1.1 QGroundControl

QGroundControl is an Open Source Mircro Air Vehicle Ground Control Station/ Operator Control Unit, which is based on PIXHAWK's Groundstation project [7].

There are several advantages of this GCS. QGroundControl is multi-platform, which means it is suitable for Windows, Linux and MacOS operation systems. This GCS is also an open-source project which developers can change build a personal version of it. More importantly, this GCS can support more than two hundred vehicles in parallel; therefore, it is suitable for projects requiring multiple unmanned systems working together.

Additionally, the interface of QGroundControl is the most friendly among the three, it even includes the voice report to attract the attention of users in specific cases and a communication console in which users can see the hex signal sent by the autopilot system. This signal display is very important in case of debugging for a testing firmware.

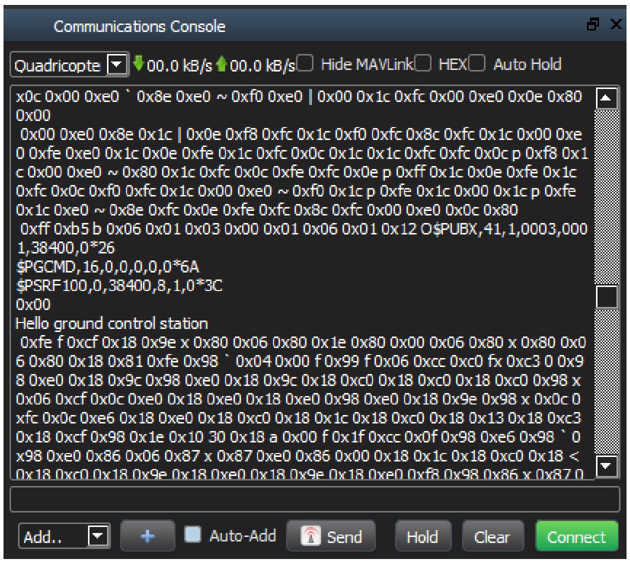


WARNING: You have NOT enabled auto-hold (stops updating the console if huge amounts of serial data arrive). Updating the console consumes significant CPU load, so if you receive more than about 5 KB/s of serial data, make sure to enable auto-hold if not using the console.

Figure 2-1: QGroundControl interface

However, there is one major problem with this GCS. After several tests, using both sample code [**2**] and ArduPilot code to test the communication with FlyMaple controller board, it is clear that FlyMaple cannot receive the messages sent by this GCS. It is not the problem that FlyMaple does not understand these messages. In fact, FlyMaple does not receive any character or byte from QGroundControl. Since FlyMaple still can receive and understand messages from the other two GCS, this problem can only be explained either by the fact that QGroundControl is not compatible with FlyMaple, even though it still can receive message from this board and understand the MAVLink protocol in that message perfectly; or by the fact that there is some undocumented configuration options that I missed

Although there are many advantages, the problem in communication discards QGroundControl as the Ground Control Station for the quadricopter. Nevertheless, QGroundControl is convenient for debugging and testing the sending function of the controller board, therefore, many tests in this project have used this GCS as a testing environment for descending traffic, as can be seen from the other parts in this report.



```
x0c 0x00 0xe0 ` 0x8e 0xe0 ~ 0xf0 0xe0 | 0x00 0x1c 0xfc 0x00 0xe0 0x0e 0x80
0x00
 0x00 0xe0 0x8e 0x1c | 0x0e 0xf8 0xfc 0x1c 0xf0 0xfc 0x8c 0xfc 0x1c 0x00 0xe
0 0xfe 0xe0 0x1c 0x0e 0xfe 0x1c 0xfc 0x0c 0x1c 0x1c 0xfc 0xfc 0x0c p 0xf8 0x1
c 0x00 0xe0 ~ 0x80 0x1c 0xfc 0x0c 0xfe 0xfc 0x0e p 0xff 0x1c 0x0e 0xfe 0x1c
0xfc 0x0c 0xf0 0xfc 0x1c 0x00 0xe0 ~ 0xf0 0x1c p 0xfe 0x1c 0x00 0x1c p 0xfe
0x1c 0xe0 ~ 0x8e 0xfc 0x0e 0xfe 0xfc 0x8c 0xfc 0x00 0xe0 0x0c 0x80
 0xff 0xb5 b 0x06 0x01 0x03 0x00 0x01 0x06 0x01 0x12 O$PUBX,41,1,0003,000
1,38400,0*26
$PGCMD,16,0,0,0,0,0*6A
$PSRF100,0,38400,8,1,0*3C
0x00
Hello ground control station
 0xfe f 0xcf 0x18 0x9e x 0x80 0x06 0x80 0x1e 0x80 0x00 0x06 0x80 x 0x80 0x0
6 0x80 0x18 0x81 0xfe 0x98 ` 0x04 0x00 f 0x99 f 0x06 0xcc 0xc0 fx 0xc3 0 0x9
8 0xe0 0x18 0x9c 0x98 0xe0 0x18 0x9c 0x18 0xc0 0x18 0xc0 0x18 0xc0 0x98 x
0x06 0xcf 0x0c 0xe0 0x18 0xe0 0x18 0xe0 0x98 0xe0 0x18 0x9e 0x98 x 0x0c 0
xfc 0x0c 0xe6 0x18 0xe0 0x18 0xc0 0x18 0x1c 0x18 0xc0 0x18 0x13 0x18 0xc3
0x18 0xcf 0x98 0x1e 0x10 30 0x18 a 0x00 f 0x1f 0xcc 0x0f 0x98 0xe6 0x98 ` 0
x98 0xe0 0x86 0x06 0x87 x 0x87 0xe0 0x86 0x00 0x18 0x1c 0x18 0xc0 0x18 <
0x18 0xc0 0x18 0x9e 0x18 0xe0 0x18 0x9e 0x18 0xe0 0xf8 0x98 0x86 x 0x87 0
```

Figure 2-2: Debugging with signal received display in QGroundControl

2.1.2 Mission Planner

According to its homepage [**8**], *"Mission Planner is a free, open-source, community-supported application developed by Michael Oborne for the open-source APM autopilot project"*, it is also developed and supported by APM, the same developing group as ArduPilot. As a result, this GCS is fully compatible with autopilot systems using ArduPilot firmware. The interface of this program is also user friendly, as can be seen in **figure 2-3**, it also contains many pros as QGroundControl (multiple-system-paralleling control, the ability to use in simulation, etc).

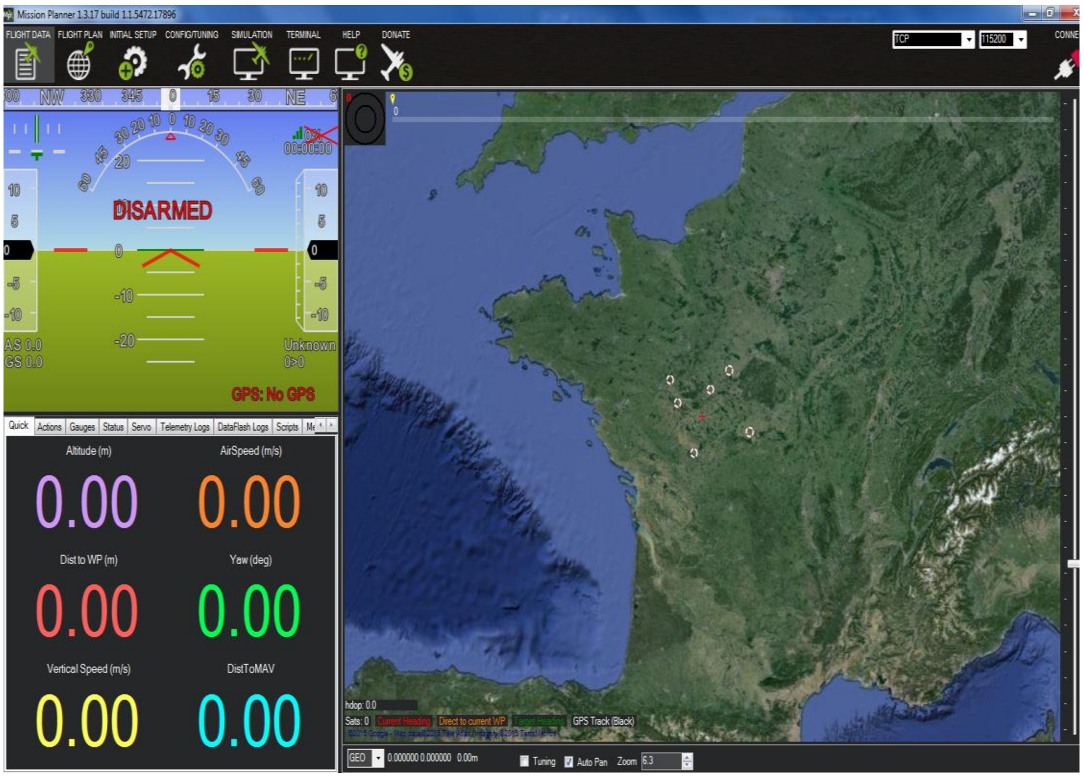
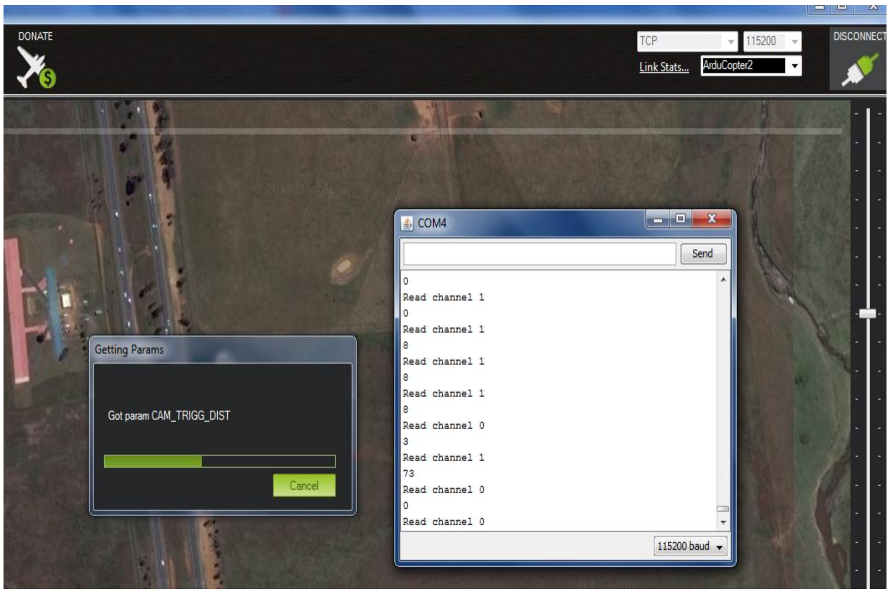


Figure 2-3: Mission Planner interface

Nevertheless, Mission Planner has its own drawbacks. The first inconvenience of this GCS is that it does not show the hex signal it received from the autopilot. Although Mission Planner provides users some space to display the text messages it received, this place only displays the text, not the whole package hex value like QGroundControl. The second problem with Mission Planner is that it is Windows compatible only, which is a major minus comparing to other GCS supporting multiple Operation Systems. However, this GCS has passed multiple tests for FlyMaple, which examines not only the possibility to receive the messages from the control system via wifi but also the ability to send the messages to the board and receive the responses from the board perfectly. For the objective of communication via TCP protocol with FlyMaple, Mission Planner is obviously the best solution so far.



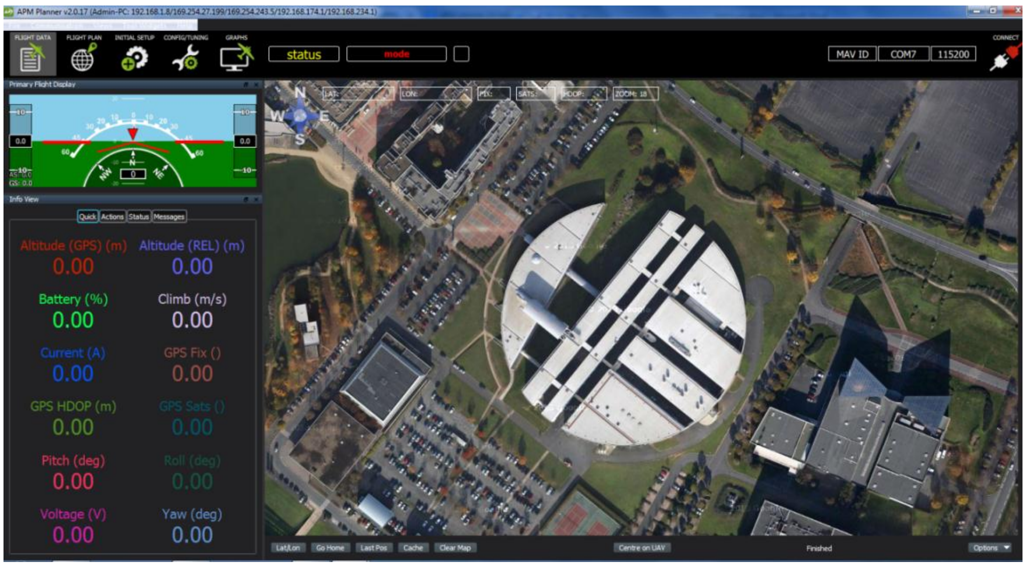
Got param CAM_TRIGG_DIST

```
0
Read channel 1
0
Read channel 1
8
Read channel 1
8
Read channel 1
8
Read channel 0
3
Read channel 1
73
Read channel 0
0
Read channel 0
```

Figure 2-4: Parameters transferring test with Mission Planner

2.1.3 APM Planner 2.0

APM Planner 2.0 is the combination between the Mission Planner and QGroundControl. It is believed to have the pros of both older GCS program: *"APM Planner 2.0 is the next generation of ground control station. It is the offspring of Mission Planner and QGround Control, combining the simple user interface of Mission Planner and cross platform capability of QGround Control."* [**9**]

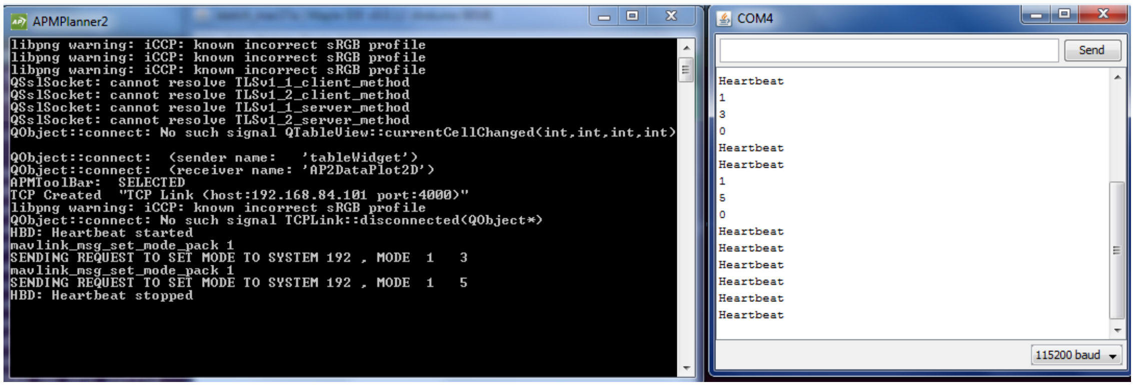


APM Planner v2.0.17 (Admin-PC: 192.168.1.8/169.254.27.199/169.254.243.5/192.168.174.1/192.168.234.1)

Altitude (GPS) (m) 0.00
Altitude (REL) (m) 0.00
Battery (%) 0.00
Climb (m/s) 0.00
Current (A) 0.00
GPS Fix () 0.00
GPS HDOP (m) 0.00
GPS Sats () 0.00
Pitch (deg) 0.00
Roll (deg) 0.00
Voltage (V) 0.00
Yaw (deg) 0.00

Figure 2-5: APM Planner 2.0 interface

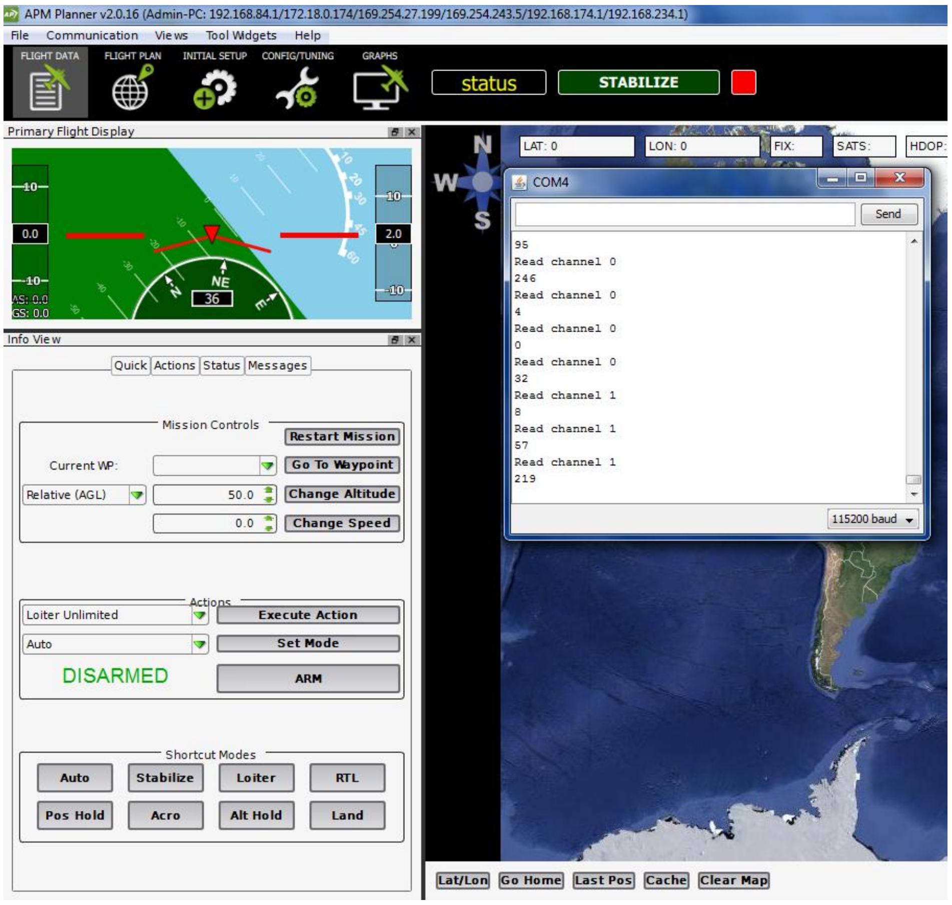
Although it can send and receive messages in certain cases, this GCS failed in the test of changing control parameters, which will be mentioned later in this report. In this case, it is also not as good as the Mission Planner above. Nevertheless, with the ability to communicate in some situation, this GCS will be used as a double-check for Mission Planner, in particular parameters changing tests and mission reading/writing tests.



```
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
QSslSocket: cannot resolve TLSv1_1_client_method
QSslSocket: cannot resolve TLSv1_2_client_method
QSslSocket: cannot resolve TLSv1_1_server_method
QSslSocket: cannot resolve TLSv1_2_server_method
QObject::connect: No such signal QTableView::currentCellChanged(int,int,int,int)

QObject::connect:  (sender name:   'tableWidget')
QObject::connect:  (receiver name: 'AP2DataPlot2D')
APMToolBar:  SELECTED
TCP Created  "TCP Link (host:192.168.84.101 port:4000)"
libpng warning: iCCP: known incorrect sRGB profile
QObject::connect: No such signal TCPLink::disconnected(QObject*)
HBD: Heartbeat started
mavlink_msg_set_mode_pack 1
SENDING REQUEST TO SET MODE TO SYSTEM 192 , MODE  1   3
mavlink_msg_set_mode_pack 1
SENDING REQUEST TO SET MODE TO SYSTEM 192 , MODE  1   5
HBD: Heartbeat stopped
```

```
Heartbeat
1
3
0
Heartbeat
Heartbeat
1
5
0
Heartbeat
Heartbeat
Heartbeat
Heartbeat
Heartbeat
Heartbeat
```

Figure 2-6: Changing mode test with APM Planner 2.0



```
95
Read channel 0
246
Read channel 0
4
Read channel 0
0
Read channel 0
32
Read channel 1
8
Read channel 1
57
Read channel 1
219
```

Figure 2-7: A test with attitude and sensors using APM Planner 2.0

A summary of the pros and cons can be seen from the table below. To sum up, **Mission Planner** is the best solution for this project and will be the main GCS in real fly testing. **QGroundContro**l is suitable for basic tests and debugging, meanwhile, **APM Planner** will be in charge of function tests to see if the results of Mission Planner are correct.

Table 5: Comparing 3 Ground Control Stations

Test results and properties of GCS	**QGroundControl**	**Mission Planner**	**APM Planner 2.0**
Easy to learn	√	√	
Documentation available	√	√	
Open source	√	√	√
Windows compatible	√	√	√
Linus/MacOS compatible	√		√
Command display	√		√
Receiving Messages	√	√	√
Sending Messages		√	√
Display hex code received from autopilot systems	√		
Parameters loading		√	√
Advance commands		√	
Suggesting function	**Debugging**	**GCS for real flight**	**Double check for Mission Planner**

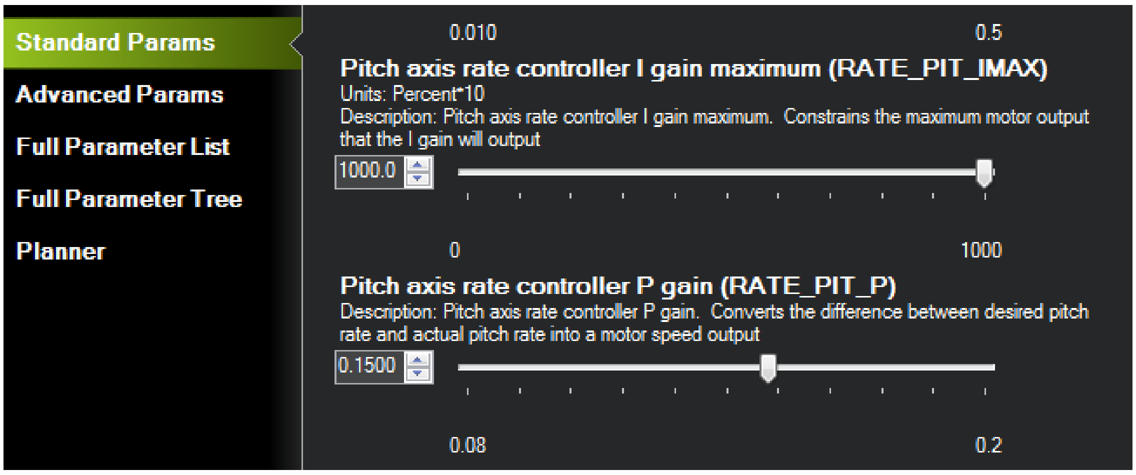
2.2 Tests with Ground Control Stations

In communication between autopilot system and Ground Control Station, changing control parameters and reading/writing missions is very important since without them, autopilot system cannot fulfill its objectives.

2.2.1 Changing control parameters

In ArduPilot, there are more than 300 control parameters. These parameters are related to PID calculating, to sensors sensitivity, to radio controlling, etc. As indicated above, among three GCSs using MAVLink protocol, Mission Planner is the only one that can handle these high level commands. A simple test was done using Mission Planner and double checked by APM Planner 2.0

In order to change control parameters in Mission Planner, users can use CONFIG/TUNNING tab, and then choose the constant that need to be changed.



0.010 0.5

Pitch axis rate controller I gain maximum (RATE_PIT_IMAX)
Units: Percent*10
Description: Pitch axis rate controller I gain maximum. Constrains the maximum motor output that the I gain will output
1000.0

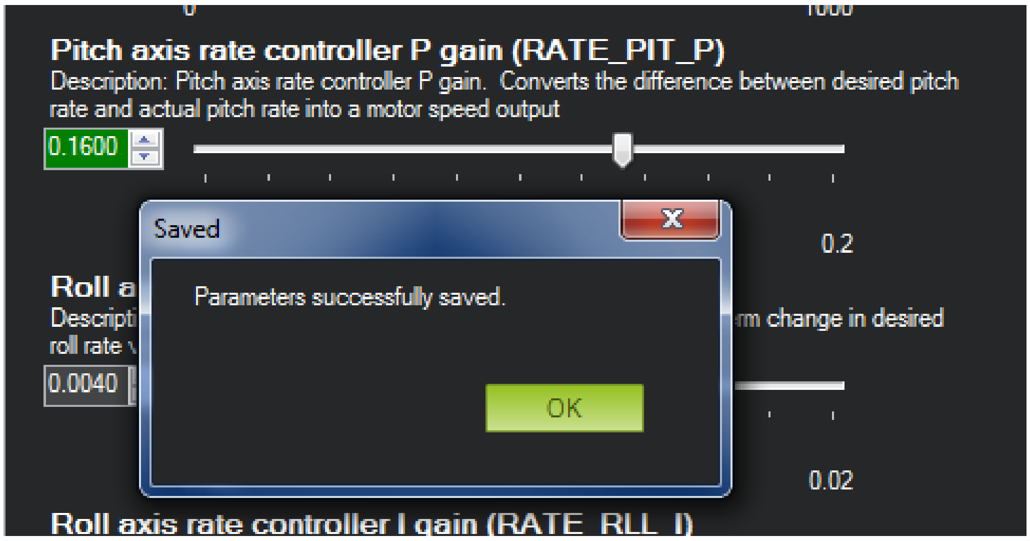
0 1000

Pitch axis rate controller P gain (RATE_PIT_P)
Description: Pitch axis rate controller P gain. Converts the difference between desired pitch rate and actual pitch rate into a motor speed output
0.1500

0.08 0.2

Figure 2-8: Parameters displays in Mission Planner

After changing the value, user needs to click **write** it into autopilot system. A message to define this new value will then be sent to autopilot and if the procedure is successful, Mission Planner will display a notification as shown in **Figure 2-9**.



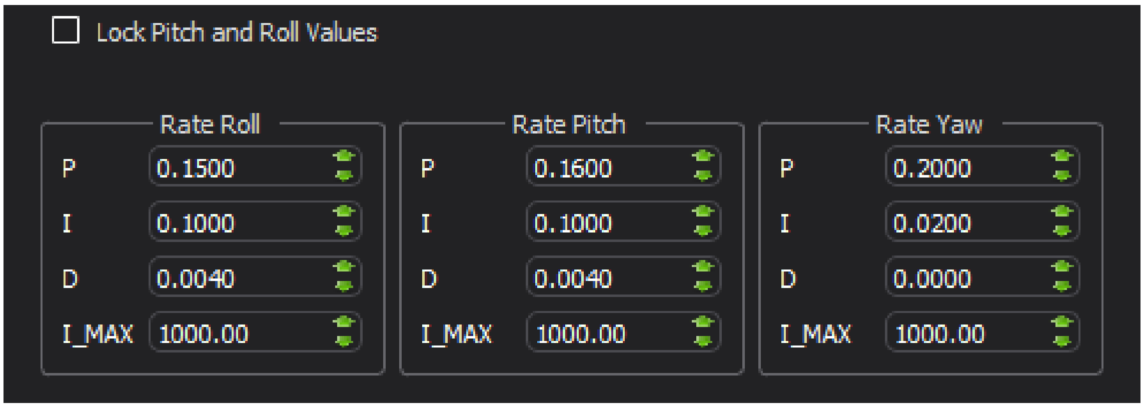
Pitch axis rate controller P gain (RATE_PIT_P)
Description: Pitch axis rate controller P gain. Converts the difference between desired pitch rate and actual pitch rate into a motor speed output
0.1600

Saved
Parameters successfully saved.
OK

0.2
0.0040
0.02
Roll axis rate controller I gain (RATE_RLL_I)

Figure 2-9: Notify in Mission Planner after Ack message received

This value is then checked again by loading it using both Mission Planner and APM Planner 2.0. In both case, the value of RATE_PIT_P is 0.16, which is the confirmation for the test.



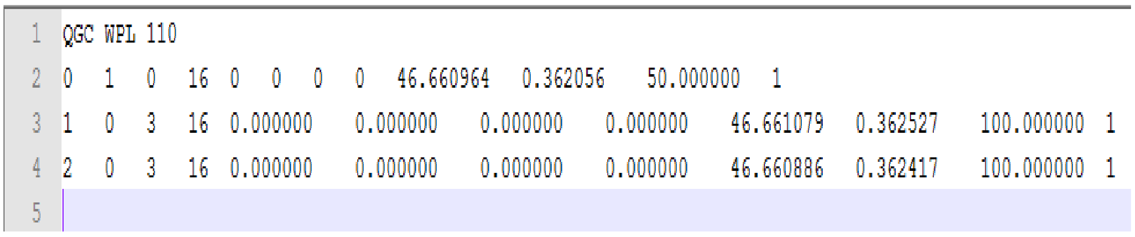
☐ Lock Pitch and Roll Values

	Rate Roll	Rate Pitch	Rate Yaw
P	0.1500	0.1600	0.2000
I	0.1000	0.1000	0.0200
D	0.0040	0.0040	0.0000
I_MAX	1000.00	1000.00	1000.00

Figure 2-10: Checking result with APM Planner 2.0

2.2.2 Flight plan design and writing/reading flight plan from GCS.

In general, there are various ways to create a flight plan for an UAV. Users can use supporting programs to create this plan or use some GCS supporting this application and can communicate with autopilot system such as Mission Planner or APM Planner 2.0. Moreover, flight plan is nothing but a *.txt file containing information about GPS longitude, latitude, altitude of the waypoint…Therefore, users can create this file with a simple text editor. **Figure 2-11** presents a flight plan created by Mission Planner.



```
QGC WPL 110
0   1   0   16  0   0   0   0   46.660964   0.362056    50.000000   1
1   0   3   16  0.000000    0.000000    0.000000    0.000000    46.661079   0.362527    100.000000  1
2   0   3   16  0.000000    0.000000    0.000000    0.000000    46.660886   0.362417    100.000000  1
```

Figure 2-11: A flight plan with three waypoints

After creating the plan, it must be uploaded into the controller board. In ArduPilot, as long as the board is fully supported by the developed firmware, writing a mission into autopilot system is nothing more than a communication sequence as shown earlier. **Figure 2-11** indicates this process.

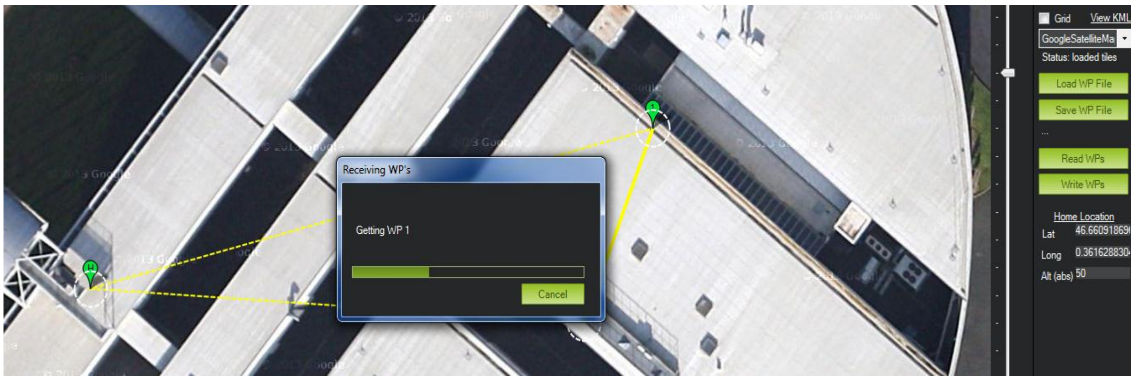


Figure 2-12: Reading flight plan

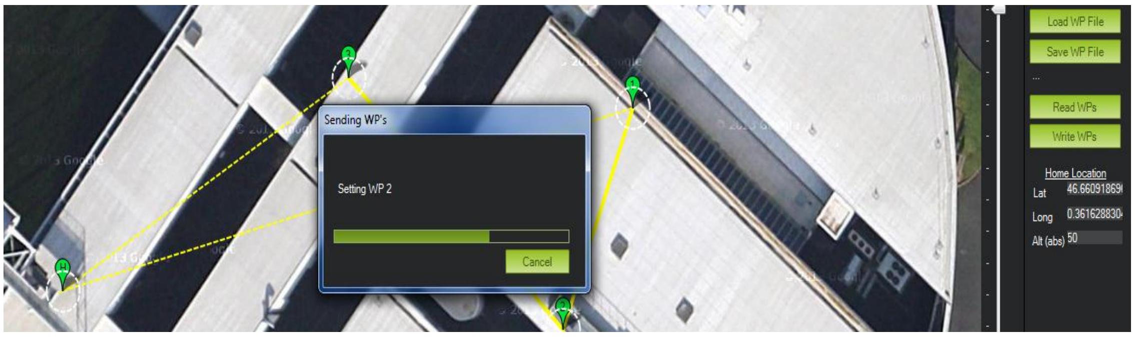


Figure 2-13: Writing flight plan

The two tests above are just some of the tests have been done to check the compatibility between ArduPilot, FlyMaple and Ground Control Station. The results from these tests led to the conclusion about each GCS in **table 4** above. From now on, in case there is no further explanation, the experiment results described in this report will belong to Mission Planner, the chosen GCS for real flight test of the Quadricopter in this project.

References

[1] Open-Source MAVLink Micro Air Vehicle Communication Protocol. [Online]. http://qgroundcontrol.org/mavlink/start

[2] MAVLink Onboard Integration Tutorial. [Online]. http://qgroundcontrol.org/dev/mavlink_onboard_integration_tutorial

[3] (2015, Mar) Cyclic redundancy check. [Online]. http://en.wikipedia.org/wiki/Cyclic_redundancy_check#Commonly_used_and_standardized_CRCs

[4] TCOM 370 Notes 99-9 Cyclic Codes and the CRC Code. [Online]. https://www.seas.upenn.edu/~kassam/tcom370/n99_9.pdf

[5] Field Reordering and CRC Extra Calculation. [Online]. http://qgroundcontrol.org/mavlink/crc_extra_calculation

[6] Waypoint Protocol. [Online]. http://qgroundcontrol.org/mavlink/waypoint_protocol

[7] QGroundControl - Home. [Online]. http://qgroundcontrol.org/start

[8] Mission Planner Overview. [Online]. http://planner.ardupilot.com/wiki/mission-planner-overview/

[9] APM Planner 2.0: Credits and Contributors. [Online]. http://planner2.ardupilot.com/home/credits-and-contributors/

[10] N.Đạt BÙI, "Embedded System for Quadricopter," Ho Chi Minh City University of Technology, Internship Report 2014.

[11] N.Dat BUI, "Ground Station, Documentations and Code explanations," Technical Report 2014.