Technical documentation
Tricopter with stabilized camera

Version 1.0

Author: Karl-Johan Barsk
Date: December 12, 2011

Status

Reviewed	Karl-Johan Barsk	111206
Approved	Fredrik Lindsten	111208

Course name:	Control Project	E-mail:	tsrt10.tricopter@gmail.com
Project group:	Triforce	Document responsible:	Karl-Johan Barsk
Course code:	TSRT10	Author's E-mail:	karba878@student.liu.se
Project:	Tricopter	Document name:	Technical documentation tricopter.pdf

Project Identity

Group E-mail:	tsrt10.tricopter@gmail.com
Homepage:	http://www.isy.liu.se/edu/projekt/tsrt10/2011/trikopter/
Orderer:	Fredrik Lindsten, Linköping University **Phone:** +46 13 - 28 13 65, **E-mail:** lindsten@isy.liu.se
Customer:	David Törnqvist, Linköping University **Phone:** +46 13 - 28 18 82 , **E-mail:** tornqvist@isy.liu.se
Course Responsible:	David Törnqvist, Linköping University **Phone:** +46 13 - 28 18 82, **E-mail:** tornqvist@isy.liu.se
Project Manager:	Josefin Kemppainen
Advisors:	Manon Kok, Linköping University **Phone:** +46 13 - 28 40 43 , **E-mail:** manon.kok@isy.liu.se

Group Members

Name	Responsibility	Phone	E-mail (@student.liu.se)
Josefin Kemppainen	Project Manager	070 - 866 56 75	joske208
Karl-Johan Barsk	Documents	070 - 788 95 48	karba878
Joakim Hallqvist	Firmware	070 - 571 37 64	joaha738
Patrik Johansson	Hardware	070 - 299 47 48	patjo855
Rasmus Jönsson	Software	070 - 999 30 19	rasjo160
Johan Larsson	Tests	070 - 747 87 78	johla342
Mattis Lorentzon	Information	070 - 592 32 66	matlo622
Björn Rödseth	Designer	070 - 274 52 20	bjoro826

Document History

Version	Date	Changes made	Sign	Reviewer
0.1	111130	First draft.	KJB	Karl-Johan Barsk
1.0	111208	Second draft.	KJB	Karl-Johan Barsk

Contents

1 Introduction

During the course of the project the group has constructed a surveillance system made up of a UAV (Unmanned Aerial Vehicle) with a mounted camera. The user is able to specify a flying route along with a specific point of interest for surveillance and the UAV will then perform the surveillance mission. This will be referred to as the autonomous flight mode. During the flight a video stream from the camera will be broadcasted to a ground station with the optional addition to record it.

Additional to the autonomous flight mode there will be a manual flight mode. The manual mode is for manual control of the tricopter with two optional safety features to prevent accidents. This will allow for unexperienced pilots to try out the equipment without risking damage to it. These are:

1. A virtual box with pre-specified boundaries. If this feature is enabled and the user breaches the boundaries, an auto-pilot will take control of the tricopter and fly it back to the center of the virtual box.
2. Easy-control, which when enabled translates the user control signals to direction commands in a fixed coordinate system instead of directly forwarding the signals to the rotors.

For more information on the safety features, see section 4.3.

2 Definitions

APM	ArduPilot Mega
ArduCopter	Open source control system for multicopters
ArduPilot	The autopilot of the tricopter
ArduPilot board	The ArduPilot and IMUcopter together as one unit
AV	Audio video
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESC	Electronic Speed Controller
GPS	Global Positioning System
GUI	Graphical User Interface
IMU	Inertial Measurement Unit
IMUcamera	IMU-module with an IMU and a processor mounted on the Gimbal
IMUcopter	An IMU-shield connected to the ArduPilot
IMU-shield	Sensor module with accelerometers, gyroscopes, magnetometers and a barometer
I^2C	Inter-Integrated Circuit
RC	Radio Control
UAV	Unmanned Aerial Vehicle
XBee	Wireless modem

2.1 Signal definitions

Tricopter heading: The heading of the tricoper in degrees from the magnetometer to compensate for the drift in the raw gyro on the GPS and the IMUcopter.

Tricopter orientation: The roll, pitch and yaw of the tricopter in degrees.

Target location: The latitude and longitude of the target in degrees and the altitude in dm.

GPS data: The tricopter's position specified in latitude and longitude (degrees), the altitude[1] in dm and the speed of the tricopter in cm/s.

2.2 Orientation

If the tricopter is seen from above and the arm with the tail pan servo is pointing south, then the arm pointing north-east will be called right and the one pointing north-west will be called left, see figure 1.

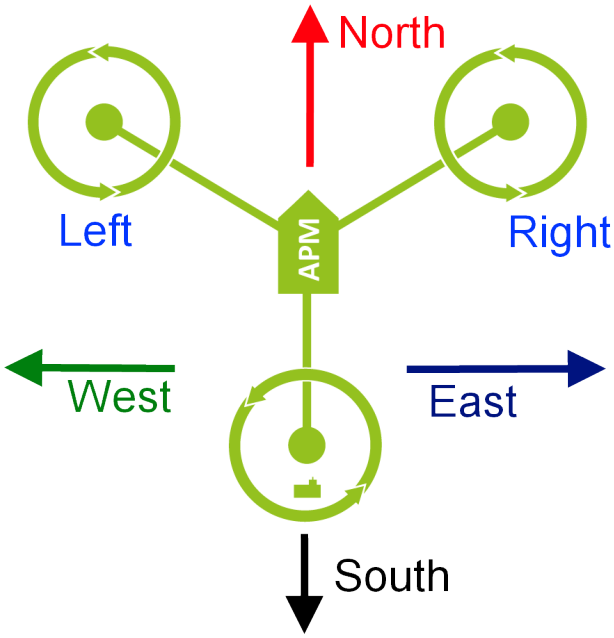


Figure 1: Right/Left orientation for the system

3 System overview

The UAV consists of a tricopter with an ArduCopter platform. ArduCopter is based on the open source autopilot ArduPilot and is one of the most sophisticated IMU-based autopilots on the market. It provides, among other things, full UAV functionality with scripted waypoints and manual RC control.

The tricopter consists of a Ground Station, a Flight unit, a Surveillance unit, a Sensor unit and a Communication unit. The relation of these subsystems can be seen in figure 2.

For more information about the ArduPilot, see [1].

[1] Due to the fact that a barometer and sonar sensor will be used to decide the altitude this information will be redundant.

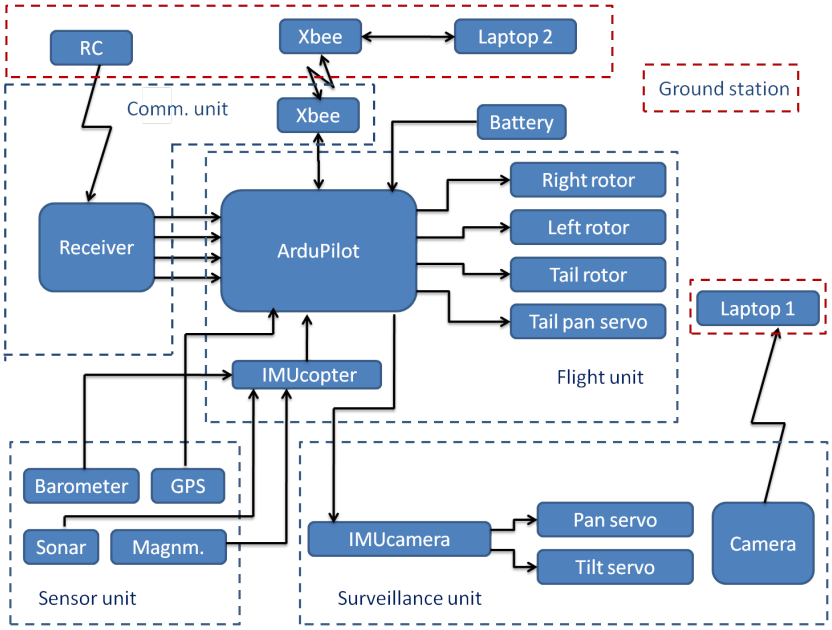


Figure 2: Block diagram for the system. For further descriptions of the subsystems, see section 8 for the Ground station, section 4 for the Flight unit, section 5 for the Surveillance unit, section 6 for the Sensor unit and section 7 for the Communication unit.

3.1 I²C

In the interface sections of the Flight unit, the Surveillance unit and the Sensor unit (sections 4.2, 5.2 and 6.2 respectively) the I²C bus is mentioned. On this bus, the ArduPilot will be acting as master. It is written in C using the ARDUINO Wire library.

I²C uses two bidirectional lines named Serial Data Line (SDA) and Serial Clock (SCL) with pull-up resistors. As the name suggests, the former is used for sending the data and the latter for the clock.

Since the bus is the same as the one used by the barometer, section 6.1.4, the given barometer code on the ArduPilot initializes the bus and sets the ArduPilot to master. This is done in the file `APM_BMP085.cpp`.

The protocol on the bus, namely the communication between the ArduPilot and the IMUcamera, can be seen in table 1 below.

Table 1: I²C protocol for communication between ArduPilot and IMUcamera.

Type	**byte 1: packet size**	**byte 2: flag**	**byte 3 to end: packet**
Current location	12	1	Lat, lng, alt
Camera target	12	2	Lat, lng, alt
Tricopter heading	12	3	Roll, pitch, yaw
Gimbal servo angles	2	4	Pan, tilt

The fourth package - servo angles to the gimbal - is for testing purposes only.

The implementation can be seen in appendix C. There the variable `busy_bus` is used to make sure that the bus is not overrun with transmissions. Each package will be sent

according to its send rate, which is equal to number of runs through the `loop()` (the main function). A (constant) variable, aptly named `send_rate_offset`, is used to further handle possible conflicts on the bus by offsetting the transmissions a number of runs.

4 Flight unit

The purpose of the flight unit is to fly the tricopter according to the commands received from the Ground station. It manages the sensor data from the Sensor unit, see section 6 and controls the Gimbal, see section 5.1.1. It consists of three rotors, one tail pan servo, IMUcopter and the ArduPilot chipset. Note that though the GPS is mounted on the ArduPilot chipset, it is considered a part of the Sensor unit, see section 6. See figure 3 for the outline of the unit.

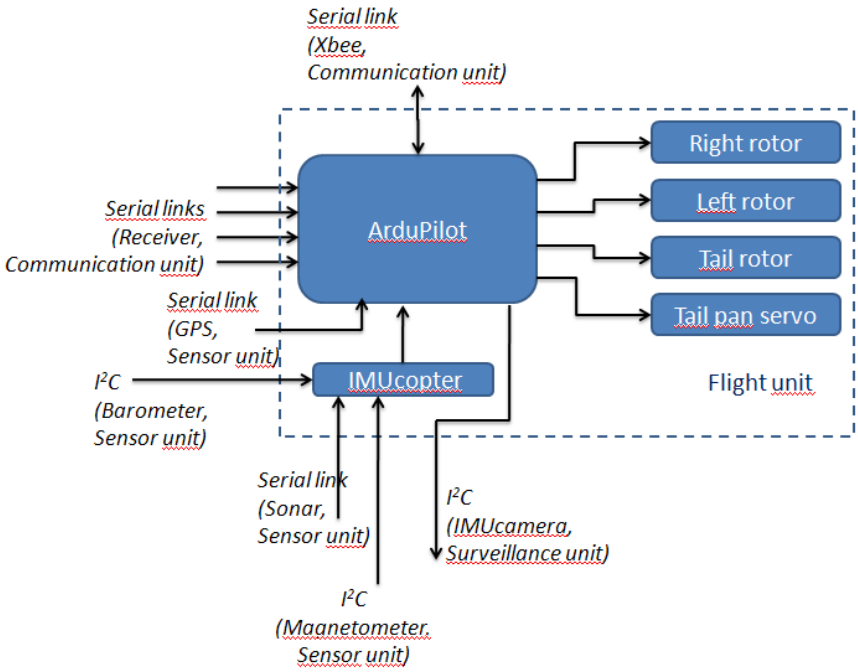


Figure 3: Block diagram for the Flight unit with ingoing and outgoing signals.

4.1 Hardware

The Flight unit consists of:

- ArduPilot Mega - Arduino Compatible UAV Controller w/ ATMega2560
- Three rotors and a tail pan servo
- EM-406/uBlox/MTK Adapter Cable 5 cm
- ArduPilot Mega IMU Shield/OilPan Rev-H (With Pins) [IMUcopter]

The ArduPilot Mega, which is an IMU-based open source autopilot, is used to control the tricopter by sending signals to both the servo on the tail and to the three ESCs which control the three rotors. The main board, which is designed with an ATMega2560 micro controller, is placed as close as possible to the tricopter's center of mass. The IMUCopter is mounted on the ArduPilot and with help of the Sensor unit, see section 6, the ArduPilot Mega is a fully functional autopilot for a UAV.

The ArduPilot IMU on the Flight unit, called IMUcopter, is used to take measurements of the acceleration and the angular velocity of the tricopter for the ArduPilot Mega. The processor on ArduPilot Mega is used to process the measurements from IMUcopter as well as measurements from the barometer, magnetometer, sonar and GPS, which are part of the sensor unit. The IMUcopter consists of the regular functionality of an IMU, which has a triple axis accelerometer and a triple axis gyro.

The ArduPilot Mega and IMUcopter will be considered as one device as described in section 4.2.1.

4.2 Interface

This section describes the communication of the Flight unit.

Signals in

- GPS data, see definition in section 2.1, from the Sensor unit via the serial port on the ArduPilot.
- Tricopter heading, see definition in section 2.1, from the magnetometer in Sensor unit via the I^2C bus and serial ports.
- Altitude from the sonar sensor, see section 6.1.1, in the Sensor unit via the port marked pitot tube on IMUcopter.
- Altitude from the barometer, see section 6.1.4, in the Sensor unit via the I^2C bus and serial ports.
- Control signals from the Communication unit.
- Route/target coordinates from the Communication unit.

Signals out

- Updated UAV flight data, i.e. tricopter orientation and GPS data according to section 2.1, to the Communication unit.
- The Flight unit provides the Surveillance unit with heading and GPS position data via an I^2C bus.

4.2.1 Internal communication

The communication between IMUcopter and ArduPilot is serial and has not been modified further. Henceforth, IMUcopter and ArduPilot will be considered as one device in terms of communication with other components.

4.3 Firmware

The firmware on the flight unit is run on the ArduPilot board which is responsible for:

- Processing all sensor data from IMUcopter.
- Processing the flight commands.
- Sending control signals to the tricopter's rotors and servo.
- Sending data to IMUcamera.

4.3.1 ArduPilot

The ArduPilot is a complete control system running on an Arduino base. It receives commands from the ground station and uses the information from the IMUcopter to stabilize the tricopter while performing these commands. It is also responsible for the autonomous flight mode. Its output signals are control signals for the rotors, tail pan servo and flight information to the ground station.

The board is delivered with open source firmware that fuses sensor information from the IMUcopter to get estimates of the tricopter's position, velocity and orientation.

Most of the ArduPilot's functionality already exists in the firmware but some things have been added:

- Functionality to send target coordinates to the IMUcamera over I^2C bus.
- Functionality to send position and orientation estimates to the IMUcamera over I^2C bus.
- Functionality for virtual box feature in manual mode.

Autonomous mode
Performing autonomous flight is a matter of translating route information to control signals for the rotors and tail pan servo. The route information consists of predefined waypoints that the tricopter should pass through. Using sensor information to estimate position, velocity and orientation, it is possible to adjust the control signals to steer the tricopter in the desired direction. The sensor information is hardware filtered on the IMUcopter and then fused on the ArduPilot to get the estimates.

The autonomous flight mode will be performed using already existing functionality in the ArduPilot firmware.

Autonomous landing: In the given code there is an implemented command for autonomous landing. By setting a land command in the APM MISSION PLANNER the tricopter will perform landing. At three meter altitude the tricopter will hold the current GPS position in longitudinal and lateral direction and then descend. When the tricopter is either 40 cm above the ground or has the speed 0 m/s the engines will be turned off and the tricopter will fall freely from this point. This command has been tested in a simulation environment and it works but it does not perform a very smooth landing. Therefore this command has not been implemented in the final product because of the risk of damaging the components.

Figure 4 shows an example of how a route specification looks like when the landing command land in the simulation. Primarily the tricopter will fly autonomously to

waypoint 1 and then further to waypoint 2. The third waypoint is not an actual waypoint because the command has been changed to a landing command. Therefore the next thing to do is to land at the location of waypoint 2 and the simulated tricopter will not fly to the third waypoint as seen in the figure. The location of waypoint 3 (the landing command) can be chosen arbitrarily.

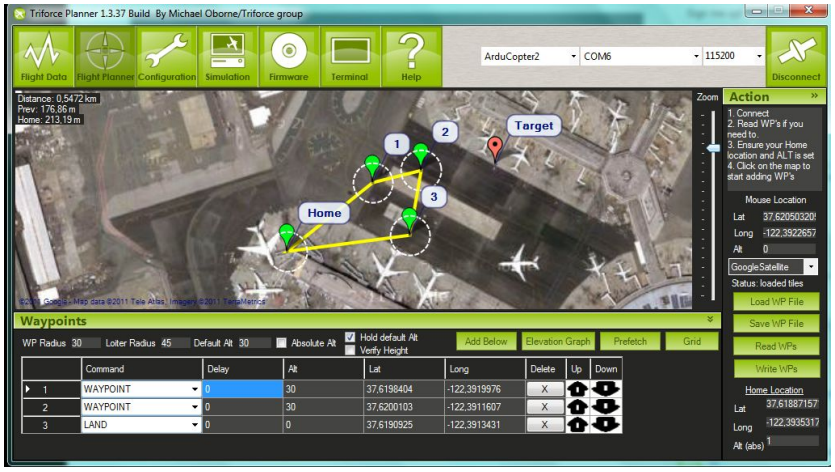


Figure 4: Exampel of a route with autonomous landing.

Manual mode

In manual mode the user controls the tricopter using an RC controller. How the control signals are interpreted depends on the feature that has been enabled.

Virtual box feature: The virtual box feature is used to confine the flight space for the tricopter. The box default size is 20 x 20 x 20 [m], but can be altered. The box cannot be too small due to inaccuracy in the GPS, thus a box size of the default one, $\pm$ 1 or 2 meters is recommended. The center point of the box is determined by the tricopter's position when the feature is activated. To prevent the tricopter from crashing because the box is placed too close to the ground, the box is automatically elevated to a safe height. This height is by default 3 meters. The box is oriented so that the sides are parallel to the longitude and latitude lines.

To calculate the boundaries B_i, the following equations have been used. The $longitude$ and $latitude$ are in degrees $*10^7$ and $size_{box}$, $altitude_{tricopter}$, $safeHeight$ and r_{earth} in meters.

$$
\begin{aligned}
B_N &= latitude + \frac{size_{box}}{2}\frac{180 \cdot 10^7}{\pi}\frac{1}{r_{earth}} \\
B_S &= latitude - \frac{size_{box}}{2}\frac{180 \cdot 10^7}{\pi}\frac{1}{r_{earth}} \\
B_W &= longitude - \frac{size_{box}}{2}\frac{180 \cdot 10^7}{\pi}\frac{1}{r_{earth}} \\
B_E &= longitude + \frac{size_{box}}{2}\frac{180 \cdot 10^7}{\pi}\frac{1}{r_{earth}} \\
B_T &= \begin{cases} altitude_{tricopter} + \frac{size_{box}}{2} & \text{if } altitude_{tricopter} \geq safeHeight + \frac{size_{box}}{2} \\ safeHeight + size_{box} & \text{else} \end{cases} \\
B_B &= \begin{cases} altitude_{tricopter} - \frac{size_{box}}{2} & \text{if } altitude_{tricopter} \geq safeHeight + \frac{size_{box}}{2} \\ safeHeight & \text{else} \end{cases}
\end{aligned} \tag{1}
$$

The subscripts i, B_i, refers to **N**orth, **S**outh, **W**est, **E**ast, **T**op and **B**ottom respectively.

When a boundary is broken the tricopter will fly back autonomously to the middle in a proximity of 5 x 5 x 5 [m]. The control of the tricopter will be given back to the user when it is back in the middle.

To calculate the boundaries of the middle of the box the equations (1) above will be used but the $size_{box}$ is replaced by the wanted $size_{middle}$, which is 5 meters by default.

This mode is activated by switching AUX1 on the RC controller to the lower level, see section 8.2.

The code for the implementation can be seen in appendix D.

Easy control feature: When this feature is activated a fixed coordinate system is used which was created and aligned to the tricopter's orientation when it was armed. As long as this feature is active all control commands are interpreted as desired flight directions in this coordinate system and will be converted to rotor commands to follow them.

This mode is activated by switching AUX2 on the RC controller to the lower level, see section 8.2.

5 Surveillance unit

The function of the Surveillance unit is to calculate the orientation of the Gimbal and then control the camera. The purpose of the camera and the Gimbal is covered in the introduction, see section 1.

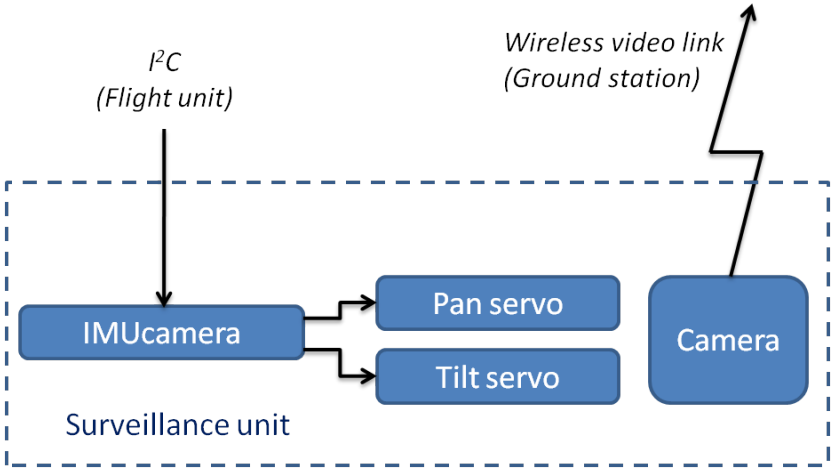


Figure 5: Block diagram for the Surveillande unit.

5.1 Hardware

The Surveillance unit consists of:

- Camera
- Gimbal
- ArduImu+V2 [IMUcamera]

5.1.1 Gimbal

The Gimbal is the device on which the camera is mounted. It consists of two servos, one to perform a panning movement and one to perform a tilting movement. This makes it possible to rotate the camera relative to the UAV so that it focuses on the target coordinates.

5.1.2 IMUCamera

IMUcamera is connected to ArduPilot via the I^2C -bus (section 3.1) from which it receives the target's location and the tricopter's position and orientation. With this information IMUcamera calculates the desired angles for the gimbal servos, see section 5.3.2.

Note that this unit is used as s processor only, since the sensor are not used. See section 5.4.

5.2 Interface

The interface between the surveillance unit and the other units, the Ground station and Flight unit, will be presented below.

Signals in:

- Tricopter orientation and target location, see definition in section 2.1, from the ArduPilot in the Flight unit. The signals are transmitted via the I^2C bus. The IMUcamera will have address 2 on the bus.

Signals out:

- Video to Laptop 1 in the Ground station via the Video link.
- Reference signal from IMUcamera to the gimbal servos.

5.3 Firmware

The firmware for the unit is run on IMUcamera and is used to communicate with ArduPilot and to calculate the gimbal servo angles.

5.3.1 Angle calculation

The first thing IMUcamera does in order to calculate the desired camera angles is to calculate the distance between the tricopter and the target in the ground plane and the bearing relative to north, see equations (2)-(6). The longitudes and latitudes received from ArduPilot are given in degrees $x10^7$ and the altitudes are given in cm. The calculated bearing is given in degrees $x10^2$. The cartesian coordinates for the target, in a coordinate system with its center at the tricopter's position see figure 7a, are given by equations (7)-(9). These are in m.

$$\Delta_{lat} = tricopter_{lat} - target_{lat} \tag{2}$$

$$\Delta_{long} = \cos(\frac{target_{lat}}{10^7}) \cdot (tricopter_{long} - target_{long}) \tag{3}$$

$$\Delta_{alt} = tricopter_{alt} - target_{alt} \tag{4}$$

$$dist = \sqrt{\Delta_{lat}^2 + \Delta_{long}^2} \cdot \frac{\pi}{180} \frac{r_{earth}}{10^7} \tag{5}$$

$$bearing = 9000 + \arctan(\frac{-\Delta_{long}}{\Delta_{lat}}) \cdot \frac{18000}{\pi} \tag{6}$$

The factor 9000 in (6) is to turn the bearing 90° towards north.

$$x_E = dist \cdot \cos(bearing) \tag{7}$$

$$y_E = dist \cdot \sin(bearing) \tag{8}$$

$$z_E = \Delta_{alt}/100 \tag{9}$$

These equations are a good approximation for coordinates close to each other but are not completely accurate. In the equations for Δ_{long} we assume that both the target and the tricopter have the same latitude. A more graphical explanation of the equations (7) - (9) can be seen i figure 6.

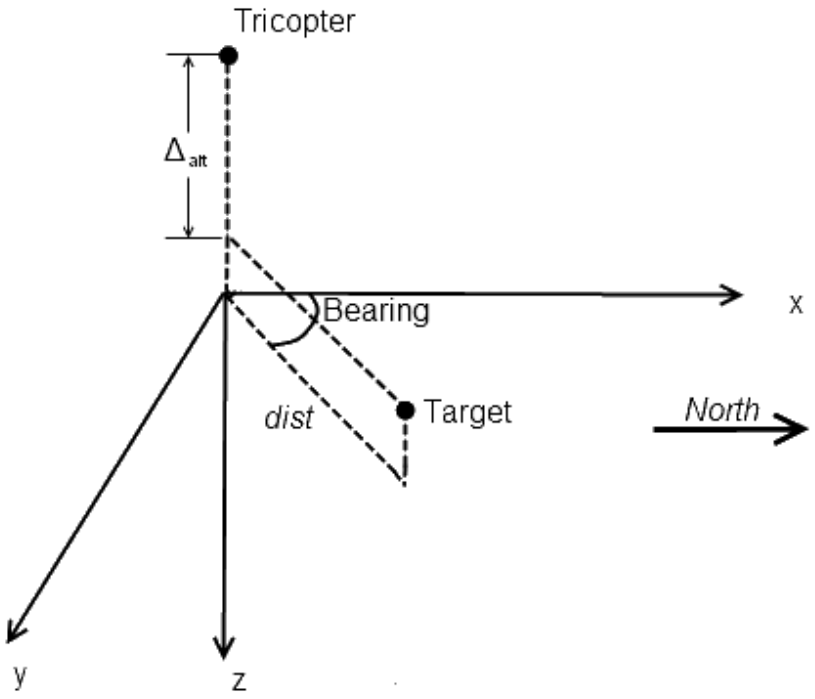


Figure 6: A graphical interpretation of the transformation from the tricopter's coordinate system to the earth's.

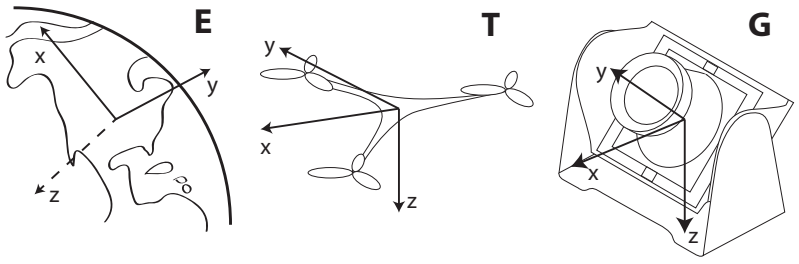
To compensate for the tricopter's heading, a three dimensional rotation matrix with the pitch (p), roll (r) and yaw (y) angles is used, see equation (10), to transfer the target to the tricopter's coordinate system, see figure 7b.

$$\begin{bmatrix} x_T \\ y_T \\ z_T \end{bmatrix} = \begin{bmatrix} \cos(r) & -\sin(r) & 0 \\ \sin(r) & \cos(r) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(p) & -\sin(p) \\ 0 & \sin(p) & \cos(p) \end{bmatrix} \begin{bmatrix} \cos(y) & 0 & \sin(y) \\ 0 & 1 & 0 \\ -\sin(y) & 0 & \cos(y) \end{bmatrix} \begin{bmatrix} x_E \\ y_E \\ z_E \end{bmatrix} \tag{10}$$

The desired gimbal angles relative to the tricopter are then given by equations (11)-(12).

$$angle_{tilt} = \arctan(\frac{z_T}{dist}) \tag{11}$$

$$angle_{pan} = \arctan(\frac{x_T}{y_T}) \tag{12}$$



(a) Earth coordinate system (E). (b) Tricopter coordinate system (T). (c) Gimbal coordinate system (G).

Figure 7: Coordinate systems.

The implementation of this can be seen in appendix B.

5.3.2 Servo input

The servos used for this project are pulse-width-modulated and the servo control code available in Arduino is used to generate the pulses given a specific angle. The servo code takes a value between 0 and 180 and generates a pulse between 0.5 and 2.5 ms. It is important to disable interrupts from the bus code during the pulse generation otherwise the pulse may stay high for too long. An offset was added so if both calculated angles are zero the camera will be pointing straight ahead. To prevent the servos from taking damage, some limitations on the servo output were implemented to correspond to the restrictions of the gimbal.

If you look at figure 8 you can suspect that the transfer from the tilt servo angle to the actual tilt of the camera is not linear because of the joints shaded in the figure. Because of this, a non-linear transfer function had to be calculated.

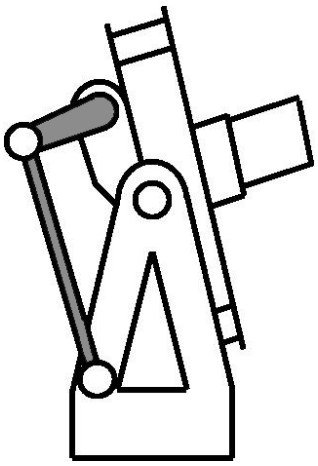


Figure 8: Gimbal sketch.

To see the relationship between the servo and gimbal angle, see figure 9. As seen by the transfer function the maximum difference between the two angles is around 5°.

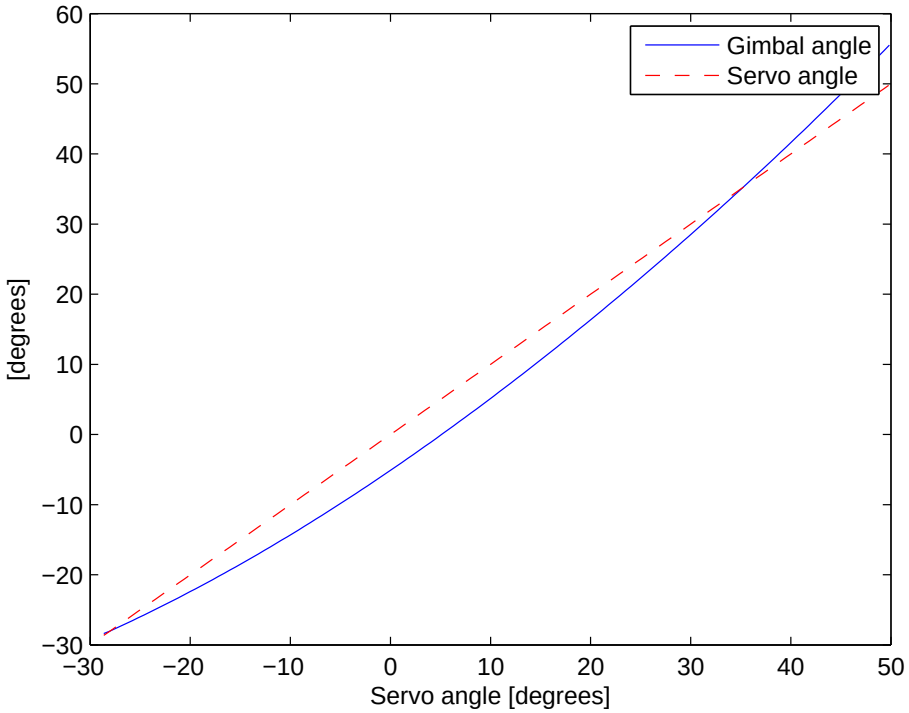


Figure 9: Gimbal transfer function. The red line represents the servo angle and the blue line the gimbal angle.

For the implementation of the gimbal transfer function, see appendix B.

5.4 Complications

Before the work on the tricopter started, the plan was to use the sensors on the IMUcamera to control and correct for eventual reference errors, e.g. drift errors. But after some testing it was concluded that the readings from the IMUcamera were so inaccurate that it was impossible to use them for the project's purpose. On the other hand, the gimbal servos were accurate enough for the IMUcamera's sensors not to be needed.

6 Sensor unit

To be able to fulfill the requirements specified by the project, additional sensors needed to be placed on the tricopter. These sensors will be described in this section. See figure 10 for the outline of the unit.

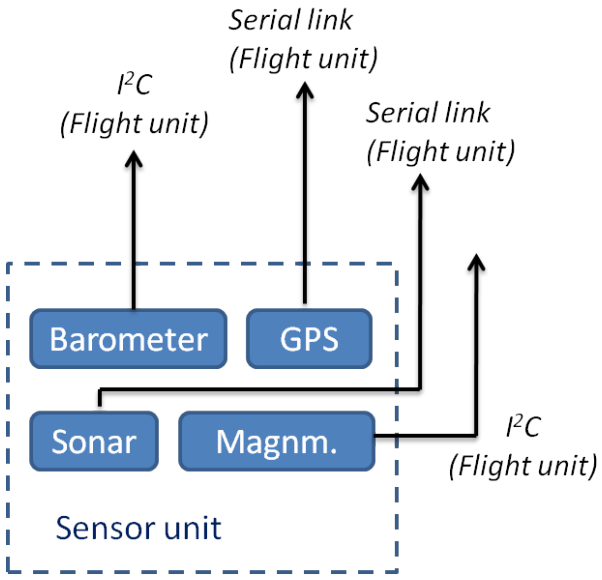


Figure 10: Block diagram for the Sensor unit with ingoing and outgoing signals.

6.1 Hardware

The Sensor unit consists of:

- MB1200 XL-MaxSonar-EZ0 High Performance Ultrasonic Range Finder
- GS407 U-Blox5 GPS 4Hz (New Antenna & Free uBlox Adapter Basic)
- HMC5883L - Triple Axis Magnetometer
- A built-in barometer

6.1.1 Sonar

Since autonomous landing was a secondary requirement in this project, a sonar sensor was mounted on the tricopter for accurate altitude determination when the tricopter is close to ground level.

If the built-in barometer (section 6.1.4) gets a height reading below eight meters the ArduPilot will start using a combined height measurement from both the sonar and the barometer. The AutoPilot uses the sonar data to create a scale variable, varying between 0 and 1, which decreases the closer the tricopter gets to the ground. Based on this it calculates its current height as presented in equation 13.

$$current\ height = scale\ variable * barometer\ readings + (1 - scale\ variable) * sonar\ readings \tag{13}$$

This sensor is placed on the main frame of the tricopter, facing the ground, with at least a distance of eight centimetres from the body to prevent the sonar sensor from picking up electrical disturbances.

6.1.2 GPS

To determine the position of the tricopter and enable waypoint navigation, a GPS module is mounted on the tricopter. This GPS module is connected to the ArduPilot board through the GPS port.

6.1.3 Magnetometer

Since the tricopter is able to hover, no heading from GPS to compensate for IMU yaw drift. To compensate for this, a magnetometer is used the magnetic field to provide a heading of the tricopter for the Flight unit and the Surveillance unit. Since the magnetic field's declination varies depending on location, this has to be accounted for in the ArduPilot. This declination which can easily be obtained online at [9], can then be set in the APM Planner, see the User manual [6].

The magnetometer is mounted on IMUcopter via an I^2C cable.

6.1.4 Barometer

The barometer is used to determine the altitude of the tricopter, which is done by measuring the air pressure. When the tricopter gets armed, the ArduPilot saves its current barometer reading, then uses that as a reference value during flight to estimate the tricopter's current altitude.

The barometer is physically mounted on IMUcopter.

6.2 Interface

This section describes the communication of the Sensor unit.

Signals in

This unit has no signals in.

Signals out

- GPS data serially, see definition in section 2.1, to the Flight unit via the serial port on the ArduPilot board.
- Magnetometer acquired signals to the Flight unit via an I^2C bus with address 0x1E on the ArduPilot board.
- Sonar sensor acquired signals to the Flight unit via the port marked pitot tube on the ArduPilot board.
- Barometer acquired signals to the Flight unit via the I^2C bus with address 0x77 on the ArduPilot board.

6.3 Complications

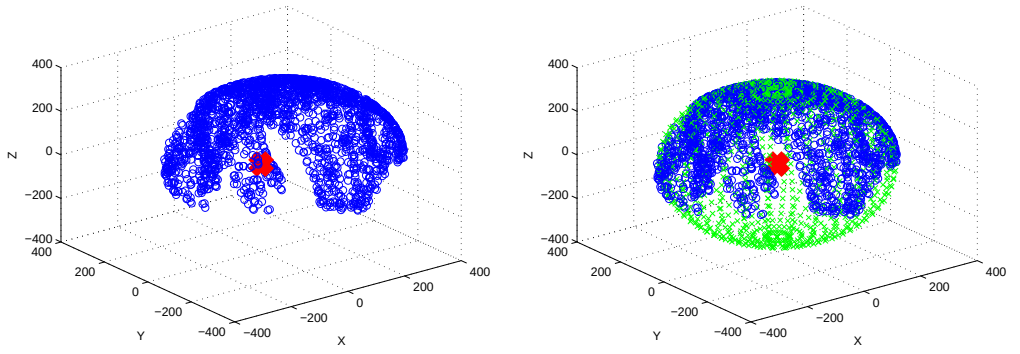
During the course of the project, a few critical issues arose. These will be listed below.

6.3.1 Magnetometer

While performing tests on the magnetometer, large offsets were measured regarding the orientation, along all axis, of the tricopter. Initially it was suspected that the electronics mounted on the tricopter were the cause of the bad compass readings. However, after further tests and investigations, it was conclusive that the offset calibration performed by the Arducopter was not well suited for a location with a relatively large vertical component of the earth's magnetic field, which Sweden is affected by.

A few data collection sessions revealed that the offsets in the magnetometer reading were stationary and therefore could be compensated for with stationary offset values. To find these offsets the collected magnetometer readings were plotted up in MATLAB. The earths magnetic field is measured as three orthogonal vector components which correspond to the three axis in the plotted figures. The length of these components combined should be constant since the measured field is approximately stationary. Therefore the plotted values should resemble a sphere centered around the origin. Raw magnetometer data revealed a sphere that was centered around an offset point from the origin. The location of this point is the offset needed to fix the incorrect magnetometer readings.

The plotted magnetometer readings can be seen in figure 11.



(a) Raw magnetometer data.

(b) Corrected magnetometer data plotted with a fitted sphere.

Figure 11: Magnetometer calibration.

6.3.2 GPS

At the end of the project, a rather unexpected error occured that caused the ArduPilot to not get GPS lock. Given the shortage of time to further inverstigate this issue, a conclusion could not be made. However, suspicions pointed to, that this issue was due to the communication between the ArduPilot and IMUcamera over the I^2C -bus, which interrupted the communication between the ArduPilot and the GPS device. Although this error has not been resolved, it can easily be worked around by first disconnecting the camera, starting the tricopter and waiting for the ArduPilot to get GPS lock. After this is done, reconnect the camera and reset the ArduPilot.

7 Communication unit

This section covers the wireless communication between the tricopter and the ground station. Figure 12 below is an overview of the communication unit.

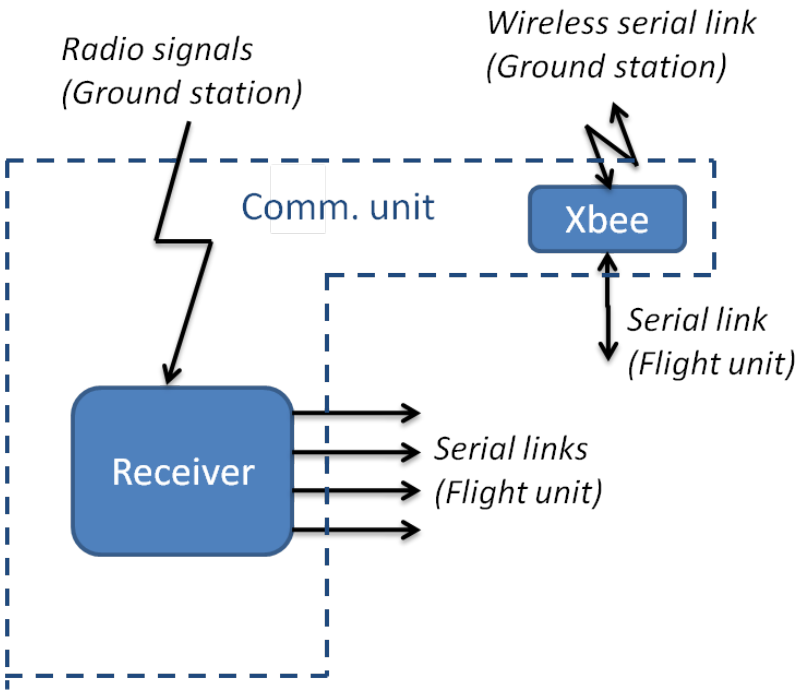


Figure 12: Block diagram for the Communication unit with ingoing and outgoing signals.

7.1 Hardware

The Communication unit consists of:

- XBee Pro 900 Wire Ant
- XBee Pro 900 RP-SMA Ant
- Multiplex Royal 9 evo
- AR7000 DSM2 7-Channel Receiver

7.1.1 XBee

The XBee will be connected to the ArduPilot and communicate while airborne with the XBee on the ground, connected to Laptop 2 which is part of the Ground station. It will be used for updating parameters, tracking sensor outputs, setting flight paths and target coordinates.

The first thing to do to get the XBee to work is to load the latest firmware using a Digi software called X-CTU [8] and set the correct baud rate. APM-planner will connect to the XBee on the baud rate 57600. **This is done with the unit installed on the product.**

The XBee is a very sensitive unit and must be used carefully. **The XBee must be connected to the ArduPilot board after the board is supplied with power from the battery. Then it must be disconnected before the battery power is broken.** It is **very important** to follow these steps otherwise the XBee will be reset and the unbricking procedure described on the ArduCopter web page [7] must be performed.

Hence, the recommended starting procedure is

1. Supply the tricopter with power from the battery.
2. Wait for the initialization to finish, after the status LEDs on the ArduCopter stop flashing rapidly.
3. Supply the XBee module on the tricopter with power, i.e. the outer of the two switches, see figure 13.
4. Wait two seconds.
5. Turn on the RX/TX switch, i.e. the inner one, see figure 13.
6. You are good to go, i.e. connect through APM Planner.

and the corresponding shutdown procedure is

1. Turn of the RX/TX switch, see figure 13.
2. Turn of the XBee power, see figure 13.
3. Shut down the tricopter.

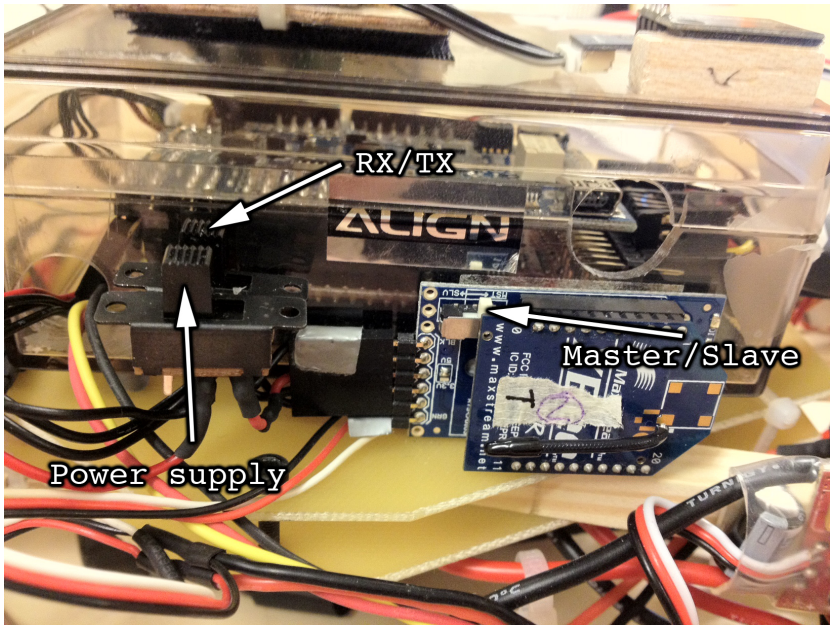


Figure 13: The three switches related to the XBee module on the tricopter.

The switch on the XBee module should be in master mode at all time, see figure 13

7.2 Interface

The interface between the Communication unit and the other units, Ground station and Flight unit, will be presented below.

Signals in:

- Control signals from the RC control in the Ground station.
- Updated route/target coordinates over XBee from the Ground station.

Signals out:

- Control signals to the ArduPilot in the Flight unit from the RC.
- Updated route/target coordinates to the ArduPilot in the Flight unit over XBee.

8 Ground station

The Ground station consists of two computers and one Radio Controller (RC). The Ground station is primarily used to control the tricopter from the ground, either with the RC (in manual mode) or the autonomous mode. It will also receive the orientation of the tricopter from the Communication unit, see section 7. One of the computers is dedicated to receiving the analogue video signal from the camera, named Laptop 1. The other is equipped with the software APM Mission Planner and communicates with the tricopter via XBee, see section 7.1.1. This computer is named Laptop 2.

See figure 14 for the outline of the unit.

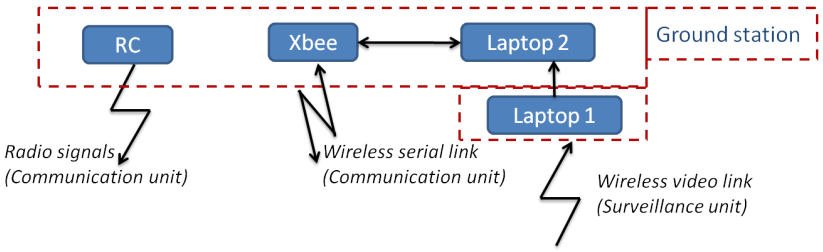


Figure 14: Block diagram for the Ground station with ingoing and outgoing signals.

8.1 Hardware

The Ground station consists of:

- iTheater glasses
- Wireless AV receiver
- Two laptops
- RC
- XBee module

- Dazzle*Tv video converter

The wireless AV receiver will receive analogue video signals from the wireless video link at the tricopter. To watch this video, iTheater glasses or a video converter connected to a laptop can be used. By using the video converter and a composite video cable, Laptop 2 can play back the video signal from Laptop 1.

For laptop to tricopter communication, the Xbee module will be used connected to Laptop 2 through an USB-port. It will send and receive signals to and from the Xbee on the tricopter. For more information on the XBee, see section 7.1.1.

8.2 Radio controller

The RC-controller that was used during the project was a Royal 9 Evo, as seen in figure 15. It is communicating with a AR7000 DSM2 7-Channel RC-receiver on the back arm on the tricopter. It uses a 2.4 GHz band frequency and a DSM2 modulation. Figure 15 shows which stick is controlling which command, note that some flight modes may change the function of the stick or disable it.

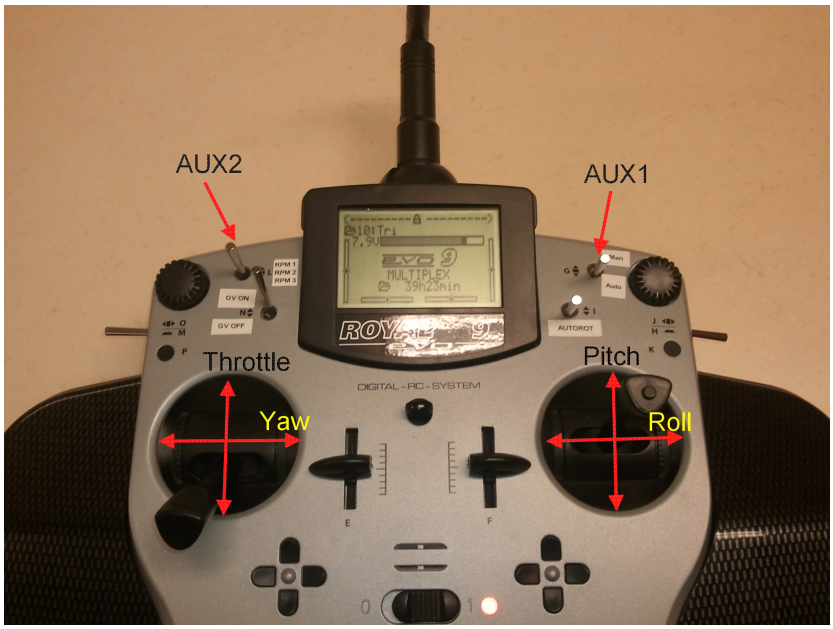


Figure 15: Radio controller and its different control sticks

AUX1 controls

- **Upper level:** Autonomous mode
- **Middle level:** Manual mode
- **Lower level:** Virtual box

AUX2 controls

- **Upper level:** Not specified

- **Middle level:** Not specified
- **Lower level:** Easy control

8.3 Interface

The interface between the Ground station and the other units, namely the Communication unit, and the Surveillance unit, is described here.

Signals in

- Analogue video signal from the video link in the Surveillance unit.
- Tricopter heading, orientation and position via XBee/USB.
- Waypoint coordinates (longitude, latitude, altitude) via XBee/USB.
- Target coordinates (longitude, latitude, altitude) via XBee/USB.
- PI parameters via XBee/USB.
- Size of the virtual box via XBee/USB.
- Other parameters e.g flight modes via XBee/USB.

Signals out

- Control signals to the Communication unit via the RC.
- Waypoint coordinates (longitude, latitude, altitude) via XBee/USB.
- Target coordinates (longitude, latitude, altitude) via XBee/USB.
- PI parameters via XBee/USB.
- Size of the virtual box via XBee/USB.
- Flight modes via XBee/USB.

8.4 Software

For the required functionality of the project, there was a need to create software for handling transmission of data and parameters, playback of video, and also for logging of the data. There already existed an open source program with a graphic interface for simulation and programming of already existed an open source program with a graphical interface for simulation and programming of the ArduCopter chipset. This program is called APM Mission Planner, [2], and was modified to fit our specific goals and purposes, while it maintained a user-friendly environment. The software is executed on Laptop 2.

8.4.1 Functionality

The functionality of the software can be divided into and listed in two parts, one with required functionality and one with optional. The required functionalities are linked to the contents of the requirement specification, [5], representing priority 1 requirements of the ground station. The optional functionalities correspond to priority 2 requirements.

Table 2: Required functionality

Functionality	**Description**
Waypoints via XBee.	Send waypoints of the flight route over the XBee.
Target point via XBee.	Send the position of the target over the XBee.
Tricopter position and orientation via XBee	Receive data about the tricopter's position and orientation over the XBee.
Playback of video	Display the video feed in VLC player on computer.
Size of the box	Set the size of the Virtual Box offline.

Table 3: Optional functionality

Functionality	**Description**
Size of the Box (wireless)	Send and receive parameters that determine the size of the Virtual Box via XBee.
Flight mode	Display or change the mode of the tricopter. Autonomous or manual mode.
Virtual Box/Easy control features	Display or change which features are active.

The attributes, required functionality, presented above, were first implemented as functions in an application that can be executed from a terminal window. When all the functionality worked, the next step was to modify the APM Mission Planner to handle the functions, which removed the need for the external application.

8.4.2 Console program

The purpose of the console program is to have a simple software that fulfils the priority 1 requirements for the ground station. The console program has the following functions:

- connect() - Connect the laptop to the tricopter, with XBee or USB-cable
- get_orientation() - Receive the orientation of the tricopter
- get_location() - Receive the position of the tricopter
- set_box_size() - Send the size of the virtual box to the tricopter
- set_target() - Send the target coordinates to the tricopter
- get_target() - Receive the target coordinates of the tricopter
- set_waypoint() - Send the coordinate of a specific waypoint to the tricopter
- add_waypoint() - Send the coordinate of a new waypoint to the tricopter

8.4.3 APM Mission Planner

The APM Mission Planner v1.0.66 is an off-the-shelf software with much of the functionality required for the project already at hand, such as updating the firmware, setting controller parameters and as the name suggests, planning waypoints for a flight route.

The software was modified to fit the requirements such as setting and displaying the target location, set the box size and display the video feed. The interesting parts of the off-the-shelf software are the "Flight Data" and "Flight Planner" tab. If there is a need to move a waypoint, left-click on the waypoint and drag it to the desired location while holding down the left mouse button.

More about how to use the modified version of the APM Mission Planner V1.0.66, is described in the User manual, [6].

The modifications that were made to APM Mission Planner can be found in appendix F, except graphical modification which were made with Microsoft Visual Studio's designer. The designer automatically generates code, so it is hard to track the exact changes. The generated code is located in the designer files e.g. `FlightData.Designer.cs`. Among these changes were the exchange of the map representation of the multicopter, from a quadcopter to a tricopter, see figure 18

Modifications in Flight Planner tab

The Flight Planner tab, see figure 16, is used for defining an intended flight route by using waypoints and for defining the target that the camera should lock on. The off-the-shelf software was only able to define the route as waypoints and the project also required a waypoint for the target that the camera should look at. The target waypoint was implemented in "Flight planner" as a red marker so the user can differ the target from route waypoint, which is green, see figure 16.

The coordinates for the target can not be written with "Flight planner" because there were some problems with the change of layout, so the reading and writing functionality to the tricopter is implemented in "Flight data".

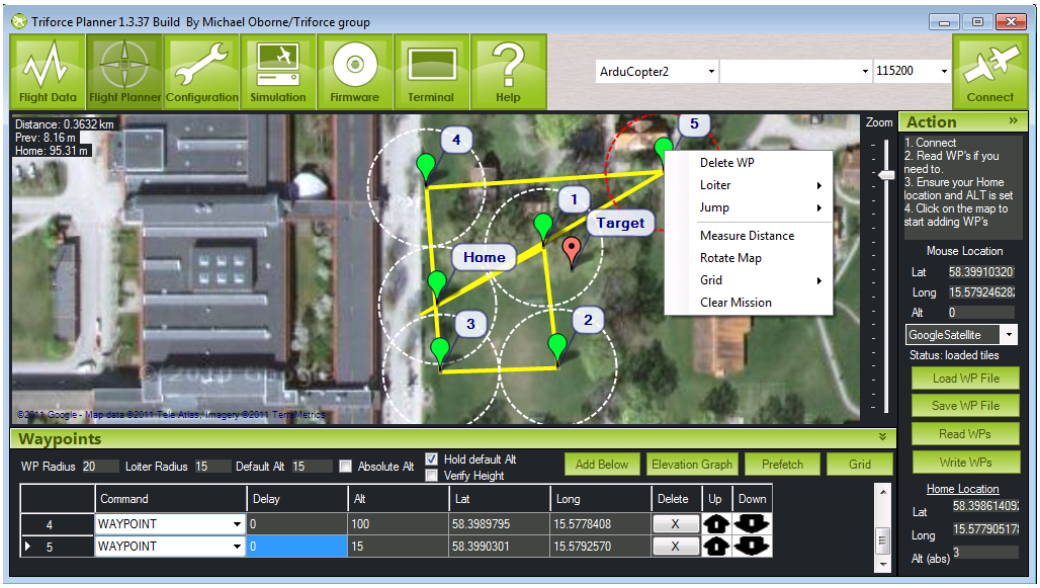


Figure 16: The modified Flight Planner tab.

Modifications in Flight Data tab

The Flight Data tab is used for displaying real-time data from the tricopter, see figure 18.

In the off-the-shelf software there were some functionalities that had to be implemented to fulfil the requirements of the software. These functionalities were displaying the location of the target and setting the virtual box size.

Sending the target coordinates, was implemented in "Flight data" instead of "Flight planner" because of problems with editing the layout. That and setting the box size functionality were implemented in the tab "Set Box & Target", which was included in the parameter box, see figure 18.

When there was some time over, the video feed box was implemented in "Flight data". The user can use this box to display the video feedback from the tricopter, if there is a video receiver connected to the laptop and the set up has been done according to the "User manual", [6]. In the non-modified version of APM Mission Planner it is possible to display the video broadcast in the attitude window, but because the camera is not fixed in the tricopter it was decided to split up the attitude and video into two separate windows. It was also done due to the desire to be able to record the video feedback and the attitude indicator separately. Because of the implementation of the window for the video broadcast, the parameter box was moved to the left.

The intended flight trajectory and the position of the tricopter are presented on a map. Representation of a target was implemented in the same way as in "Flight planner". A red marker shows the coordinates of the target. The real-time position indicator for the multicopter in "Flight data" was changed from a quad copter to a tricopter.

The difference between the modified and the non-modified APM Mission Planner, can be seen in figures 18 17, respectively.

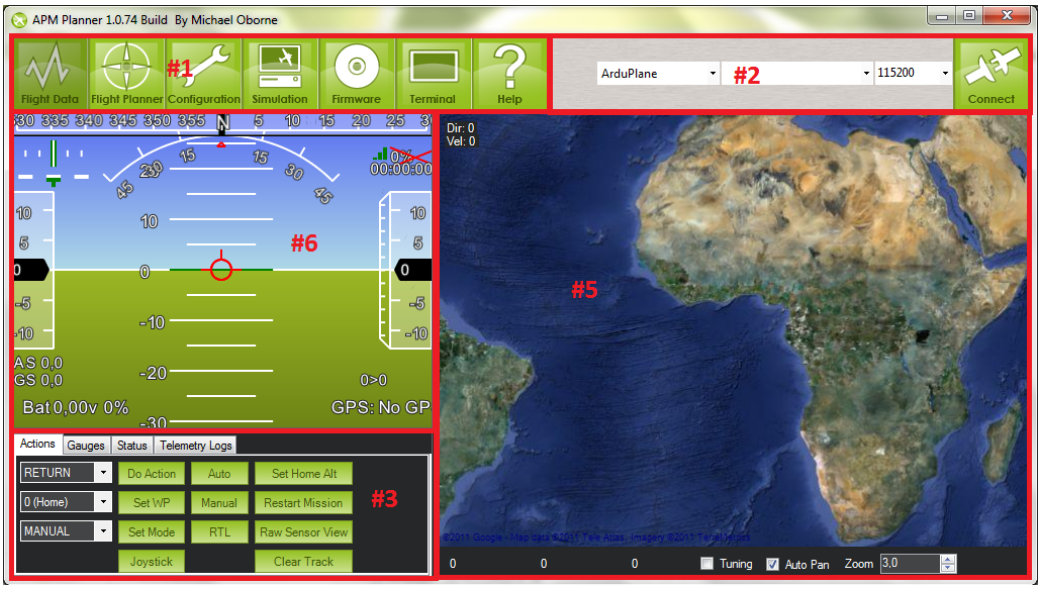


Figure 17: The non-modified Flight Data tab.

Figure 18: The modified Flight Data tab.

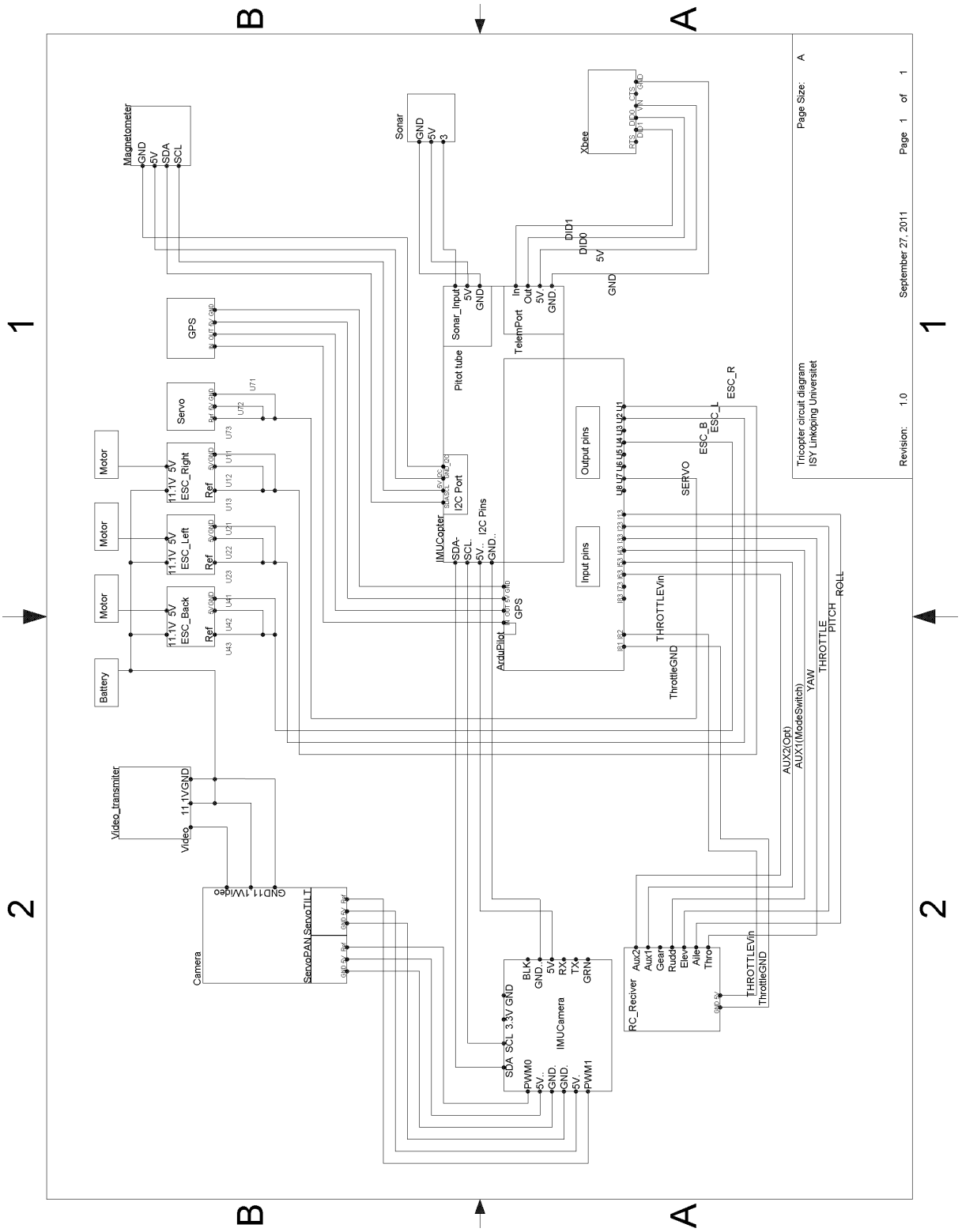
The numbering in the figures above:

#1 Menu bar
#2 Connection bar
#3 Parameter box
#4 Video window
#5 Map window
#6 Attitude window

Modifications to firmware related to APM Mission Planner

To be able to send and receive target position and virtual box size between the tricopter and APM Mission Planner, and also for the sent values to be saved to the EEPROM on the ArduPilot, some modifications hade to be made to the firmware. These can be found in appendix E.

A Wiring diagram



B Surveillance unit code

In `arduimu.pde`:

```
//
// Gimbal code
//

#include <Wire.h>       // I2C library
#include <Waypoints.h> // Used for location struct
#include <Servo.h>      // Servo library

//
// Macros
//
#define ToRad(x) ((x)*0.01745329252)
#define ToDeg(x) ((x)*57.2957795131)

//
// I2C
//
#define CURRENT_LOCATION      1          // Package ID
#define TARGET_LOCATION       2          // --- || ---
#define TRICOPTER_HEADING     3          // --- || ---
#define GIMBAL_SERVO_ANGLES   4          // --- || ---

#define I2C_MAX_MESSAGE_LENGTH 32     // Max message length in bytes

//
// Gimbal
//
#define OFFSET_TILT    215.0f          // Tilt offset to level gimbal ↩
    (degrees)
#define OFFSET_PAN      80.0f          // Pan offset to level gimbal  ↩
    (degrees)
#define OFFSET_THETA    24.0f          // Theta offset to level tilt  ↩
    (degrees)
#define MAX_TILT        30             // Maximum tilt servo output   ↩
    (degrees, w.o offset)
#define MIN_TILT       -65             // Minimum tilt servo output   ↩
    (degrees, w.o offset)
#define MAX_PAN         90             // Maximum pan servo output    ↩
    (degrees, w.o offset)
#define MIN_PAN        -90             // Minimum pan servo output    ↩
    (degrees, w.o offset)

//
// Current angles
//
static float angle_tilt = 0.0f;       // Gimbal tilt angle (w.o ↩
    offset)
static float angle_pan  = 0.0f;       // Gimbal pan angle  (w.o ↩
    offset)

//
// Servo
```

```
//
Servo servo_tilt;                       // Tilt servo object
Servo servo_pan;                        // Pan servo object

//
// Locations and orientation
//
Waypoints::WP target;                   // Target location in    (lng,↩
    lat,alt)
Waypoints::WP tricopter_pos;            // Tricopter location in (lng,↩
    lat,alt)

long tricopter_roll  = 0;               // Tricopter roll angle  (↩
    degrees*100)
long tricopter_pitch = 0;               // Tricopter pitch angle (↩
    degrees*100)
long tricopter_yaw   = 0;               // Tricopter yaw angle   (↩
    degrees*100)

static bool new_orientation = false; // New orientation data has ↩
    been received?
static bool target_in_sight = false; // Target is in sight?

//
// I2C received data
//
byte received_data[I2C_MAX_MESSAGE_LENGTH];

//
// Setup - Run once at startup
//
void setup() {

   //
   // Gimbal servo setup
   //
   servo_tilt.attach(10); // analog pin 1
   servo_tilt.setMaximumPulse(2450);
   servo_tilt.setMinimumPulse(450);

   servo_pan.attach(9); // analog pin 0
   servo_pan.setMaximumPulse(2450);
   servo_pan.setMinimumPulse(450);

   //
   // Set angles to zero at start
   //
   angle_tilt = 0;
   angle_pan  = 0;
   setServoAngles();

   //
   // Reset locations (soccer field outside B-house)
   //
   target.lat = 583982430;
   target.lng = 155789512;
   target.alt = 0;
```

```
    tricopter_pos.lat = 583979029;
    tricopter_pos.lng = 155789512;
    tricopter_pos.alt = 0;

    //
    // Initialize I2C
    //
    for(uint8_t i=0; i<I2C_MAX_MESSAGE_LENGTH; i++)
    received_data[i] = 0;
    Wire.begin(0x02); // Address 2 on the bus
    Wire.onReceive(receiveEvent);
    Wire.onRequest(requestEvent);
}

//
// Main loop
//
void loop() {

    //
    // Update angles when new orientation has been received
    //
    if(new_orientation) {
        cli();
        calculateAngles();
        setServoAngles();
        new_orientation = false;
        sei();
    }

    //
    // Refresh servos ~ every 20 ms (disable interrupts!)
    //
    cli();
    Servo::refresh();
    sei();

}

//
// I2C - receive event
//
void receiveEvent(int howMany) {

    // At least two bytes sent
    int8_t payload = Wire.receive();
    int8_t flag    = Wire.receive();

    // Get all data
    while(Wire.available()) {
        int i = payload - Wire.available();
        byte data_byte = Wire.receive();
        received_data[i] = data_byte;
    }

    // Interpret message
```

```
   switch(flag) {

      // New tricopter location
      case CURRENT_LOCATION:
         tricopter_pos.lat    = (long) convertToint32_t(received_data↩
             );
         tricopter_pos.lng    = (long) convertToint32_t(received_data↩
             +4);
         tricopter_pos.alt    = (long) convertToint32_t(received_data↩
             +8);
         break;

      // New target location
      case TARGET_LOCATION:
         target.lat           = (long) convertToint32_t(received_data↩
             );
         target.lng           = (long) convertToint32_t(received_data↩
             +4);
         target.alt           = (long) convertToint32_t(received_data↩
             +8);
         break;

      // New tricopter heading
      case TRICOPTER_HEADING:
         tricopter_roll       = (long) convertToint32_t(received_data↩
             );
         tricopter_pitch      = (long) convertToint32_t(received_data↩
             +4);
         tricopter_yaw        = (long) convertToint32_t(received_data↩
             +8);
         new_orientation      = true;
         break;

      // New requested gimbal angles
      case GIMBAL_SERVO_ANGLES:
         angle_pan            = float(int8_t(received_data[0]));
         angle_tilt           = float(int8_t(received_data[1]));
         setServoAngles();
         break;
   }
}

//
// I2C - request event
//
void requestEvent() {

   // Only one type of request - Target in sight?
   if(target_in_sight)
      Wire.send(1);
   else
      Wire.send(0);
}

//
// Convert 4 bytes to int32
//
```

```
int32_t convertToint32_t(byte* data) {
   return int32_t((uint32_t(data[3]) << 4*6) + (uint32_t(data[2]) <<↩
        4*4) + (uint32_t(data[1]) << 4*2) + uint32_t(data[0]));
}

//
// Clamp servo angles to [-180, 180]
//
void clampServoAngles() {
   // Get right angle interval
   while(angle_pan  < -180.0f) angle_pan  += 360.0f;
   while(angle_pan  >  180.0f) angle_pan  -= 360.0f;
   while(angle_tilt < -180.0f) angle_tilt += 360.0f;
   while(angle_tilt >  180.0f) angle_tilt -= 360.0f;

   // Clamp servo angles
   if(angle_pan  < MIN_PAN)  angle_pan  = MIN_PAN;
   if(angle_pan  > MAX_PAN)  angle_pan  = MAX_PAN;
    if(angle_tilt < MIN_TILT) angle_tilt = MIN_TILT;
    if(angle_tilt > MAX_TILT) angle_tilt = MAX_TILT;
}

//
// Set out desired angles to servos
//
void setServoAngles() {

   // Clamp angles to valid interval
   clampServoAngles();

   //
   // Compensate for non-linearity in tilt angle output.
   // See technical documentation for more details.
   //

   // Parameters
   #define GIMBAL_A   12 // (mm)
   #define GIMBAL_B   12
   #define GIMBAL_C   27
   #define GIMBAL_D   22
   #define GIMBAL_F   6
   #define GIMBAL_G   11
   #define GIMBAL_PHI 0.49934672 // atan(F/G)
   #ifndef PI
      #define PI  3.1415927
   #endif

   // Calculate theta and add offset for leveling tilt
   float theta    = ToRad(angle_tilt + OFFSET_THETA);

   // Calculate alpha from theta and parameters
   float psi      = PI - GIMBAL_PHI - theta;
   float i        = sqrt(GIMBAL_B*GIMBAL_B + GIMBAL_D*GIMBAL_D - 2*↩
       GIMBAL_B*GIMBAL_D*cos(psi));
   float alpha_1  = acos((i*i + GIMBAL_A*GIMBAL_A - GIMBAL_C*↩
       GIMBAL_C)/(2*GIMBAL_A*i));
   float alpha_2  = acos((i*i + GIMBAL_B*GIMBAL_B - GIMBAL_D*↩
```

```
        GIMBAL_D)/(2*GIMBAL_B*i));
    float alpha     = alpha_1 + alpha_2;

    // Get servo angle from alpha
    float tilt_out = OFFSET_TILT - ToDeg(alpha);

    // Write servo angle
    servo_tilt.writef(tilt_out);

    // Set pan angle (no non-linearity)
    servo_pan.writef(OFFSET_PAN - angle_pan);

    /* // Debug
    Serial.print(" Tilt in: "); Serial.print(angle_tilt);
    Serial.print(" Theta: "); Serial.print(ToDeg(theta));
    Serial.print(" ( "); Serial.print(theta); Serial.print(" ) ");
    Serial.print(" Alpha: "); Serial.print(ToDeg(alpha));
    Serial.print(" ( "); Serial.print(alpha); Serial.print(" ) ");
    Serial.print(" Tilt out: "); Serial.print(tilt_out);
    */
}

//
// Get distance in ground plane between to locations
//
long get_distance(Waypoints::WP *loc1, Waypoints::WP *loc2) {

    // Valid input?
    if(loc1->lat == 0 || loc1->lng == 0)
        return -1;
    if(loc2->lat == 0 || loc2->lng == 0)
        return -1;

    // Latitude of loc2 in radians
    float rads = (abs(loc2->lat) / 10000000) * 0.0174532925;

    // Get longitude scalings from latitude
    float _scaleLongDown = cos(rads);
    float _scaleLongUp   = 1.0f/cos(rads);

    // Get angle differences
    float dlat  = (float)(loc2->lat - loc1->lat);
    float dlong = ((float)(loc2->lng - loc1->lng)) * _scaleLongDown;

    // Get distance from angle difference
    return sqrt(sq(dlat) + sq(dlong)) * .01113195;
}

//
// Get bearing from loc1 to loc2
//
long get_bearing(Waypoints::WP *loc1, Waypoints::WP *loc2) {

    // Valid input?
    if(loc1->lat == 0 || loc1->lng == 0)
        return -1;
    if(loc2->lat == 0 || loc2->lng == 0)
```

```
      return -1;

   // Latitude of loc2 in radians
   float rads = (abs(loc2->lat) / 10000000) * 0.0174532925;

   // Get longitude scalings from latitude
   float _scaleLongDown = cos(rads);
   float _scaleLongUp   = 1.0f / cos(rads);

   // Get longitude difference
   long off_x = loc2->lng - loc1->lng;

   // Get latitude difference
   long off_y = (loc2->lat - loc1->lat) * _scaleLongUp;

   // Get bearing from differences
   long bearing = 9000 + atan2(-off_y, off_x) * 5729.57795;

   // Wrap bearing if necessary
   if(bearing < 0)
      bearing += 36000;
   return bearing;
}

//
// Calculate gimbal angles
//
void calculateAngles() {

   // Get tricopter orientation in radians
   float roll_rad  = ToRad(tricopter_roll /100.0f);
   float pitch_rad = ToRad(tricopter_pitch/100.0f);
   float yaw_rad   = ToRad(tricopter_yaw  /100.0f);

   // Calculate cos and sin values
   float cos_roll  = cos(-roll_rad);
   float sin_roll  = sin(-roll_rad);
   float cos_pitch = cos(-pitch_rad);
   float sin_pitch = sin(-pitch_rad);
   float cos_yaw   = cos(-yaw_rad);
   float sin_yaw   = sin(-yaw_rad);

   // Calculate distance and bearing from tricopter to target
   float dist    = (float)get_distance(&tricopter_pos, &target);
   float bearing = (float)ToRad(get_bearing(&tricopter_pos, &target)↩
       /100.0f);

   // Get relative coordinates in world system
   float x = dist*cos(bearing);
   float y = dist*sin(bearing);
   float z = -(target.alt - tricopter_pos.alt)/100.0f; // cm to m

   // Rotate relative coordinates according to tricopter orientation
   float y_p = x*cos_pitch*cos_yaw - y*cos_pitch*sin_yaw + z*↩
       sin_pitch;
    float x_p = x*(cos_yaw*sin_pitch*sin_roll+cos_roll*sin_yaw) + y↩
        *(cos_roll*cos_yaw-sin_pitch*sin_roll*sin_yaw) - z*cos_pitch↩
```

```
        *sin_roll;
    float z_p = -(x*(sin_roll*sin_yaw-cos_roll*cos_yaw*sin_pitch) + ↩
        y*(cos_yaw*sin_roll+cos_roll*sin_pitch*sin_yaw) + z*↩
        cos_pitch*cos_roll);

   // Get resulting angles
   angle_tilt = ToDeg(atan2(z_p, abs(dist)));
   angle_pan  = ToDeg(atan2(x_p, y_p));

   // Target in sight?
   if(angle_tilt > MAX_TILT ||
      angle_tilt < MIN_TILT ||
      angle_pan  > MAX_PAN  ||
      angle_pan  < MIN_PAN) {
         target_in_sight = false;
   }
   else{
      target_in_sight = true;
   }

   /* // Debug
   Serial.print(" !!! ");
   Serial.print(" Tilt: ");Serial.print(angle_tilt);
   Serial.print(" Pan: ");Serial.print(angle_pan);
   Serial.print(" Dist: ");Serial.print(dist);
   Serial.print(" Bearing: ");Serial.print(bearing);
   Serial.print(" Tri.lng: ");Serial.print(tricopter_pos.lng);
   Serial.print(" Tri.lat: ");Serial.print(tricopter_pos.lat);
   Serial.print(" Tar.lng: ");Serial.print(target.lng);
   Serial.print(" Tar.lat: ");Serial.print(target.lat);
   Serial.print(" x: ");Serial.print(x);
   Serial.print(" y: ");Serial.print(y);
   Serial.print(" z: ");Serial.print(z);*/

}
```

C I²C code

In `defines.h`:

```
//
// I2C parameters
#define IMU_CAMERA_ADRESS                       0x02
#define CURRENT_LOCATION_PACKET_SIZE            12
#define CURRENT_LOCATION_FLAG                   1
#define CAMERA_TARGET_LOCATION_PACKET_SIZE      12
#define CAMERA_TARGET_LOCATION_FLAG             2
#define TRICOPTER_HEADING_PACKET_SIZE           12
#define TRICOPTER_HEADING_FLAG                  3
#define GIMBAL_SERVO_ANGLES_PACKET_SIZE         2
#define GIMBAL_SERVO_ANGLES_FLAG                4

// A send rate of 0 equals no sending over the bus
// The send rate is sends per runs through 'loop()'
#define CURRENT_LOCATION_SEND_RATE          4000
#define CAMERA_TARGET_LOCATION_SEND_RATE    6000
#define TRICOPTER_HEADING_SEND_RATE         1000
#define GIMBAL_SERVO_ANGLES_SEND_RATE       0
//
```

In `ArduCopter.pde`:

```
//
// I2C variables
// The offset of the sent packages are send_rate_offset runs through↩
    'loop()'
static int busy_bus                         = 0;
static const int send_rate_offset           = 200;
static int update_current_location          = send_rate_offset*3;
static int update_camera_target_location    = send_rate_offset*2;
static int update_tricopter_heading         = send_rate_offset;
static int update_gimbal_servo_angles       = 0;
static int32_t angle_pan                    = 0;
static int32_t angle_tilt                   = 0;
//
```

In `loop()` in `ArduCopter.pde`:

```
// Sending tricopter location over I2C bus
   #if CURRENT_LOCATION_SEND_RATE != 0
      if(busy_bus == 0 && update_current_location == ↩
          CURRENT_LOCATION_SEND_RATE)
      {
         update_current_location = 0;
         busy_bus = send_rate_offset;

         Wire.beginTransmission(IMU_CAMERA_ADRESS);
         Wire.send(CURRENT_LOCATION_PACKET_SIZE);
         Wire.send(CURRENT_LOCATION_FLAG);
```

```
        Wire.send((uint8_t*)(makeCurrentLocationSendVector()), ↩
            CURRENT_LOCATION_PACKET_SIZE);
        Wire.endTransmission();
    }
    if(update_current_location < CURRENT_LOCATION_SEND_RATE)
        update_current_location++;
#endif
```

```
// Sending camera target location over I^2C bus
#if CAMERA_TARGET_LOCATION_SEND_RATE != 0
    if(busy_bus == 0 && new_camera_target && ↩
        update_camera_target_location == ↩
        CAMERA_TARGET_LOCATION_SEND_RATE)
    {
        update_camera_target_location = 0;
        busy_bus = send_rate_offset;

        Wire.beginTransmission(IMU_CAMERA_ADRESS);
        Wire.send(CAMERA_TARGET_LOCATION_PACKET_SIZE);
        Wire.send(CAMERA_TARGET_LOCATION_FLAG);
        Wire.send((uint8_t*)(makeCameraTargetLocationSendVector()),↩
            CAMERA_TARGET_LOCATION_PACKET_SIZE);
        Wire.endTransmission();
    }
    if(update_camera_target_location < ↩
        CAMERA_TARGET_LOCATION_SEND_RATE)
        update_camera_target_location++;
#endif
```

```
    // Sending tricopter heading over I2C bus
    #if TRICOPTER_HEADING_SEND_RATE != 0
        if(busy_bus == 0 && update_tricopter_heading == ↩
            TRICOPTER_HEADING_SEND_RATE)
        {
            update_tricopter_heading = 0;
            busy_bus = send_rate_offset;

            Wire.beginTransmission(IMU_CAMERA_ADRESS);
            Wire.send(TRICOPTER_HEADING_PACKET_SIZE);
            Wire.send(TRICOPTER_HEADING_FLAG);
            Wire.send((uint8_t*)(makeTricopterOrientationSendVector()),↩
                TRICOPTER_HEADING_PACKET_SIZE);
            Wire.endTransmission();
        }
        if(update_tricopter_heading < TRICOPTER_HEADING_SEND_RATE)
            update_tricopter_heading++;
    #endif
```

```
    // Sending gimbal servo angles over I2C bus
    #if GIMBAL_SERVO_ANGLES_SEND_RATE != 0
        if(busy_bus == 0 && update_gimbal_servo_angles == ↩
            GIMBAL_SERVO_ANGLES_SEND_RATE)
        {
```

```
            update_gimbal_servo_angles = 0;
            busy_bus = send_rate_offset;

            Wire.beginTransmission(IMU_CAMERA_ADRESS);
            Wire.send(GIMBAL_SERVO_ANGLES_PACKET_SIZE);
            Wire.send(GIMBAL_SERVO_ANGLES_FLAG);
            Wire.send(angle_pan);
            Wire.send(angle_tilt);
            Wire.endTransmission();
        }
        if(update_gimbal_servo_angles < GIMBAL_SERVO_ANGLES_SEND_RATE)
            update_gimbal_servo_angles++;
    #endif
```

```
    if(busy_bus > 0)
        busy_bus--;
```

```
// Making a vector with the current location to send over the I2C ↩
    bus
static long* makeCurrentLocationSendVector()
{
        long send_vector[3];
        send_vector[0] = current_loc.lat;
        send_vector[1] = current_loc.lng;
        send_vector[2] = current_loc.alt;

        return send_vector;
}
```

```
// Making a vector with the target location to send over the I2C bus
static long* makeCameraTargetLocationSendVector()
{
        long send_vector[3];
        send_vector[0] = camera_target.lat;
        send_vector[1] = camera_target.lng;
        send_vector[2] = camera_target.alt;

        return send_vector;
}
```

```
// Making a vector with the tricopter orientation to send over the ↩
    I2C bus
static long* makeTricopterOrientationSendVector()
{
        long send_vector[3];
        send_vector[0] = dcm.roll_sensor;
        send_vector[1] = dcm.pitch_sensor;
        send_vector[2] = dcm.yaw_sensor;

        return send_vector;
}
```

D Virtual box code

In `defines.h`:

```
//Virual box parameters
//-------------------------
#define BOX_SIZE                    20
// A box that is BOX_SIZE x BOX_SIZE x BOX_SIZE [m], default size
#define BOX_HEIGHT_ABOVE_GROUND     3
// The box height above the ground in [m]
#define MIDDLE_SIZE                 5
// The resolution of the middle in the box, in [m]

#define WEST                        1
#define EAST                        2
#define NORTH                       3
#define SOUTH                       4

#define PREVIOUSLY_BROKEN           1
#define PREVIOUSLY_UN_BROKEN        0
#define VIRTUAL_BOX_UPDATE_RATE     10000
#define LOITER_COUNTER_RATE         4
#define BOX_BOUNDARIES              10
#define MIDDLE_BOUNDARIES           11
//
```

In `ArduCopter.pde`:

```
//Virtual box parameters
static struct  Location middle_box;
static struct  Location start_loc;

static int virtual_box_counter = 0;
//Counter that controls how often we check if the boundaries are ↩
    broken.
static byte boundaries = PREVIOUSLY_UN_BROKEN;
//A variable that says if the boundaries previously been broken or ↩
    unbroken
static bool box_created = false;
//A bool that check if the box been created or not
static int32_t boundary_west;       //Boundaries for the box
static int32_t boundary_east;
static int32_t boundary_north;
static int32_t boundary_south;
static int32_t boundary_top;
static int32_t boundary_bottom;
static int32_t middle_boundary_west;
// Boundaries for the middle-box
static int32_t middle_boundary_east;
static int32_t middle_boundary_north;
static int32_t middle_boundary_south;
static int32_t middle_boundary_top;
static int32_t middle_boundary_bottom;
static int current_box_size = BOX_SIZE;
```

```
// Determines the size of the box

//
```

In `loop()` in `ArduCopter.pde`:

```
if(box_created && virtual_box_counter == VIRTUAL_BOX_UPDATE_RATE)
   {
      virtual_box_counter = 0;

      #ifndef GO_TO_LOITER_BEFORE_GUIDED
         // If the boundaries not been broken but it breaks now, the we will go to guided and fly back to the middle of the box
         if ((boundaries == PREVIOUSLY_UN_BROKEN && boundaries_breaks()))
         {
            boundaries = PREVIOUSLY_BROKEN;        // Set the param to previously broken
            old_guided_WP = guided_WP;
            guided_WP = middle_box;             //  Set the waypoint to the middle of the box
            set_mode(GUIDED);                  // Return to middle
         }
      #else
         if ((boundaries == PREVIOUSLY_UN_BROKEN && boundaries_breaks()) || control_mode == LOITER)
         {
            set_mode(LOITER);

            if(loiter_counter == LOITER_COUNTER_RATE)
            {
               loiter_counter = 0;
               boundaries = PREVIOUSLY_BROKEN;
               old_guided_WP = guided_WP;
               guided_WP = middle_box;
               set_mode(GUIDED); //Return to middle
            }


            if(loiter_counter < LOITER_COUNTER_RATE)
               loiter_counter++;
         }
      #endif

      // The boundaries been broken and we arrive to the middle, then the control is given back to the user and the box-mode is turned on.
      if (boundaries == PREVIOUSLY_BROKEN && in_middle())
      {
         #ifdef VIRTUAL_BOX_PRINT
            Serial.println("***NOTE: In middle!");
         #endif
         boundaries = PREVIOUSLY_UN_BROKEN;
         guided_WP = old_guided_WP;
         set_mode(BOX);
```

```
        }
    }
    if (virtual_box_counter < VIRTUAL_BOX_UPDATE_RATE)
        virtual_box_counter++;
```

```
//Functions that controls if the boundaries breaks
static bool boundaries_breaks()
{
        if(current_loc.lat > boundary_north)
        {
            #ifdef VIRTUAL_BOX_PRINT
                Serial.println("boundary_north BROKEN!");
            #endif
            return true;
        }

        else if(current_loc.lat < boundary_south)
        {
            #ifdef VIRTUAL_BOX_PRINT
                Serial.println("boundary_south BROKEN!");
            #endif
            return true;
        }

        else if(current_loc.lng > boundary_east)
        {
            #ifdef VIRTUAL_BOX_PRINT
                Serial.println("boundary_east BROKEN!");
            #endif
            return true;
        }

        else if(current_loc.lng < boundary_west)
        {
            #ifdef VIRTUAL_BOX_PRINT
                Serial.println("boundary_west BROKEN!");
            #endif
            return true;
        }

        // 2D box in longitude and latitude
        #ifndef IGNORE_HIGHT
            else if(current_loc.alt < boundary_bottom)
            {
                #ifdef VIRTUAL_BOX_PRINT
                    Serial.println("boundary_bottom BROKEN!");
                #endif
                return true;
            }

            else if(current_loc.alt > boundary_top)
            {
                #ifdef VIRTUAL_BOX_PRINT
                    Serial.println("boundary_top BROKEN!");
                #endif
                return true;
```

```
        }
    #endif

    else
    {
        #ifdef VIRTUAL_BOX_PRINT
            Serial.println("boundary NOT BROKEN!");
            Serial.print("current_box_size: ");
            Serial.println(current_box_size);
        #endif
        return false;
    }
}
```

```
//Controls if the tricopter is back in the middle-box, either in 2D ↩
    or 3D.
static bool in_middle()
{
    #ifndef IGNORE_HEIGHT
        return(  (current_loc.lat < middle_boundary_north) &&
                 (current_loc.lat > middle_boundary_south)    &&
                 (current_loc.lng < middle_boundary_east)     &&
                 (current_loc.lng > middle_boundary_west)     &&
                 (current_loc.alt < middle_boundary_top)      &&
                 (current_loc.alt > middle_boundary_bottom)    );
    #else
        return(  (current_loc.lat < middle_boundary_north) &&
                 (current_loc.lat > middle_boundary_south)    &&
                 (current_loc.lng < middle_boundary_east)     &&
                 (current_loc.lng > middle_boundary_west)  );
    #endif
}
```

In system.pde:

```
// This function creates the virtual box and the middle - box ↩
    boundaries.
static void create_virtual_box()
{
    if(current_box_size == 0)
        current_box_size = BOX_SIZE;

    middle_box     = current_loc; //The middle in the box will be the↩
        current position
    start_loc      = current_loc; //The start_loc will be the current↩
        position

     boundary_west  = get_boundary(WEST, BOX_BOUNDARIES);
     boundary_east  = get_boundary(EAST, BOX_BOUNDARIES);
     boundary_north = get_boundary(NORTH, BOX_BOUNDARIES);
     boundary_south = get_boundary(SOUTH, BOX_BOUNDARIES);

     if(start_loc.alt < (current_box_size/2+BOX_HEIGHT_ABOVE_GROUND)↩
         *100)
     {
```

```
    boundary_top = (current_box_size/2+BOX_HEIGHT_ABOVE_GROUND)*100 +↩
        current_box_size*100/2;
    boundary_bottom = (current_box_size/2+BOX_HEIGHT_ABOVE_GROUND)↩
        *100 - current_box_size*100/2;
    middle_box.alt  = (boundary_top + boundary_bottom)/2;
    }

    else
    {
        boundary_top      = start_loc.alt + current_box_size*100/2;
        boundary_bottom   = start_loc.alt - current_box_size*100/2;
    }

    middle_boundary_west    = get_boundary(WEST, MIDDLE_BOUNDARIES);
    middle_boundary_east    = get_boundary(EAST, MIDDLE_BOUNDARIES);
    middle_boundary_north   = get_boundary(NORTH, MIDDLE_BOUNDARIES);
    middle_boundary_south   = get_boundary(SOUTH, MIDDLE_BOUNDARIES);
    middle_boundary_top     = middle_box.alt + MIDDLE_SIZE/2*100;
    middle_boundary_bottom  = middle_box.alt - MIDDLE_SIZE/2*100;
}
```

```
// Calculates the boundaries with aid of the start location.
static int32_t get_boundary(byte compass_direction, byte ↩
    boundary_type)
{
    if(start_loc.lat == 0 || start_loc.lng == 0)
        return -1;

    int32_t boundary              = 0;
    int boundary_size             = 0;
    int32_t start_loc_lat_abs     = start_loc.lat;

    if (start_loc_lat_abs < 0)
        start_loc_lat_abs = -start_loc_lat_abs;

    float rads_temp               = (start_loc_lat_abs/10000000.0) * ↩
        0.0174532925;
    float scaleLongDown_temp      = cos(rads_temp);
    float scaleLongUp_temp        = 1.0f / cos(rads_temp);
    float awesome_constant        = .01113195;
    float super_awesome_constant  = 1.0f/awesome_constant;

    if(boundary_type == BOX_BOUNDARIES)
    {
        boundary_size = current_box_size;
    }
    else
    {
        boundary_size = MIDDLE_SIZE;
    }

    switch (compass_direction)
    {
      case NORTH:
          boundary = (start_loc.lat*awesome_constant+boundary_size↩
              *0.5)*super_awesome_constant;
```

```
            break;
        case SOUTH:
            boundary = (start_loc.lat*awesome_constant-boundary_size↩
                *0.5)*super_awesome_constant;
            break;
        case WEST:
            boundary = start_loc.lng - int32_t((boundary_size*0.5)*(↩
                super_awesome_constant*scaleLongUp_temp));
            break;
        case EAST:
            boundary = start_loc.lng + int32_t((boundary_size*0.5)*(↩
                super_awesome_constant*scaleLongUp_temp));
            break;
    }
    return boundary;
}
```

In `set_mode()` in `system.pde`:

```
// A mode that creates a virtual box if the box-mode is turned on. ↩
    When the tricopter is inside the box the streering functionality↩
     is like stabilize.
   case BOX:
      if (! box_created)
      {
         #ifdef VIRTUAL_BOX_PRINT
            Serial.println("***NOTE: BOX CREATED (like a boss)");
         #endif
         create_virtual_box();
         box_created = true;
      }

      yaw_mode       = YAW_HOLD;
      roll_pitch_mode = ROLL_PITCH_STABLE;
      throttle_mode  = THROTTLE_MANUAL;
      reset_hold_I();
      break;
```

The following code has been added in every mode i `set_mode` in `system.pde` to remove the box when box mode is deactivated.

```
// If a virtual box has been created, un-make the box
        if (box_created)
        {
            box_created = false;
            boundaries = PREVIOUSLY_UN_BROKEN;
        }
```

E Firmware modifications to support sending/receiving virtual box size and target position

E.1 Target position

In `Parameters.h`:

```
...

k_param_camera_target_alt = 230,
k_param_camera_target_lat,
k_param_camera_target_lng,

...

AP_Int32 camera_target_alt;
AP_Int32 camera_target_lat;
AP_Int32 camera_target_lng;

...

camera_target_alt (0, k_param_camera_target_alt, PSTR("CAMERA_ALT"))↩
    ,
camera_target_lat (0, k_param_camera_target_lat, PSTR("CAMERA_LAT"))↩
    ,
camera_target_lng(0, k_param_camera_target_lng, PSTR("CAMERA_LNG")),
```

In `ArduCopter.pde`:

```
static struct  Location camera_target; // camera target waypoint.
static bool new_camera_target; // Flag to tell us if a new camera ↩
    target is received.

...

// If new target has been sent to tricopter, save it to eeprom and
// print to console that new target position has been received.
void update_camera_target()
{
   if (((g.camera_target_alt != camera_target.alt) ||
        (g.camera_target_lat != camera_target.lat) ||
        (g.camera_target_lng != camera_target.lng)) &&
       !new_camera_target)
   {
      g.camera_target_alt.save();
      camera_target.alt = g.camera_target_alt;
      g.camera_target_lat.save();
      camera_target.lat = g.camera_target_lat;
      g.camera_target_lng.save();
      camera_target.lng = g.camera_target_lng;
      new_camera_target = true;

   }
   else if (new_camera_target)
```

```
    {
        Serial.print("***NEW TARGET*** latitude: ");
        Serial.print((float)camera_target.lat/10000000);
        Serial.print(" longitude: ");
        Serial.print((float)camera_target.lng/10000000);
        Serial.print(" altitude: ");
        Serial.println((float)camera_target.alt/100);
        Serial3.print("***NEW TARGET*** latitude: ");
        Serial3.print((float)camera_target.lat/10000000);
        Serial3.print(" longitude: ");
        Serial3.print((float)camera_target.lng/10000000);
        Serial3.print(" altitude: ");
        Serial3.println((float)camera_target.alt/100);
        new_camera_target = false;
    }
}
```

At line 1248 in function `slow_loop()`:

```
// If target is not hardcoded, check if new has been received.
#ifndef HARD_CODED_CAMERA_TARGET
    update_camera_target();
#endif
```

E.2 Virtual box size

In `Parameters.h`:

```
...

k_param_box_size,

...

AP_Int16    box_size;

...

box_size(20, k_param_box_size, PSTR("BOX_SIZE")),
```

In `ArduCopter.pde`:

```
static bool box_size_changed = false; // Flag if new box size has ↩
    been received.

...

// If new box size has been sent to tricopter, save it to eeprom and
// print to console that new size has been changed.
void update_current_box_size()
{
    if ((g.box_size != current_box_size) && !box_size_changed)
    {
```

```
        current_box_size = g.box_size;
        box_size_changed = true;
    }
    else if (box_size_changed)
    {
        Serial.print("***NOTE: Box size changed!! New size ");
        Serial.println(current_box_size);
        Serial3.print("***NOTE: Box size changed!! New size ");
        Serial3.println(current_box_size);
        box_size_changed = false;
    }
}
```

At line 1240 in function `slow_loop()`:

```
// Check if new box size has been received and
// update the virtual box size if that is the case.
update_current_box_size();
if(box_size_changed) {
    update_virtual_box_size();
    box_size_changed = false;
}
```

F APM Mission Planner code

F.1 Box size functionality

In `Common.cs`:

```
// Modified to be consistent with changes made in the arducopter ↩
    code
// (simple is no longer a mode). Also, Box mode is added.
public enum ac2modes
{
    STABILIZE = 0,          // hold level position
    ACRO = 1,          // rate control
    //SIMPLE = 2,     //
    ALT_HOLD = 2,     // AUTO control //***changed***
    AUTO = 3,          // AUTO control //***changed***
    GUIDED = 4,        // AUTO control //***changed***
    LOITER = 5,        // Hold a single location //***changed***
    RTL = 6,              // AUTO control //***changed***
    CIRCLE = 7, //***changed***
    POSITION = 8, //***added***
    BOX = 9    //***added***
}
```

In `CurrentState.cs`, at line 270 in function `UpdateCurrentSettings(...)`:

```
// Modified to be consistent with changes made in the arducopter ↩
    code
// (simple is no longer a mode). Also, Box mode is added.
switch (sysstatus.mode)
{

...

    /*case (byte)102:
        mode = "Simple";
        break;*/
    case (byte)102: //***changed***
        mode = "Alt Hold";
        break;
    case (byte)103: //***changed***
        mode = "Auto";
        break;
    case (byte)104: //***changed***
        mode = "Guided";
        break;
    case (byte)105: //***changed***
        mode = "Loiter";
        break;
    case (byte)106: //***changed***
        mode = "RTL";
        break;
    case (byte)107: //***changed***
        mode = "Circle";
        break;
```

```
    case (byte)108: //***added***
        mode = "Position";
        break;
    case (byte)109: //***added***
        mode = "Box";
        break;
...

}
```

In `FlightData.cs`:

```
// Box size variables.
public static float BoxSize = 20;
public static float CurrentBoxSize = 20;

...

// Run when typing in the virtual box size text box.
private void textBox1_TextChanged(object sender, EventArgs e)
{
    try
    {
        if (sizeBox.Text != "")
        {
            if (int.Parse(sizeBox.Text) > 0)
            {
                BoxSize = float.Parse(sizeBox.Text);
            }
            else
            {
                MessageBox.Show("Box size can not be zero!");
                sizeBox.Text = "";
            }
        }
    }
    catch
    {
        MessageBox.Show("Invalid box size!!!");
        sizeBox.Text = "";
    }

}

...

// Update local virtual box size and display it.
public void UpdateBoxSize(float size)
{
    CurrentBoxSize = size;
    label2.Text = "Size: " + CurrentBoxSize;
}

...

// Run when clicking the "send" button for the virtual box size.
```

```
private void SendBoxSize(object sender, EventArgs e)
{
    if (comPort.BaseStream.IsOpen)
    {
        if (BoxSize < 0 || BoxSize > 255)
            Console.WriteLine("Invalid size");
        else if (MainV2.comPort.BaseStream.IsOpen)
        {
            try {
                    MainV2.comPort.setParam("BOX_SIZE", BoxSize);
                    MessageBox.Show("New box size set!");
                }
            catch (Exception ex) { Console.WriteLine(ex); }
        }
        try
        {
            MainV2.comPort.param = MainV2.comPort.getParamList();
            UpdateBoxSize((float)MainV2.comPort.param["BOX_SIZE"]);
        }
        catch (Exception ex) { Console.WriteLine(ex); }

    }
    else
        MessageBox.Show("Tricopter is not connected!");
}
```

F.2 Target functionality

In `FlightData.cs`:

```
// Target Vars
static public string tar_lat="";
static public string tar_lng = "";
// Current target
static float Clat = 58.39845f;
static float Clng = 15.57792f;
static float Calt = 1f;

...

// Run when typing in the target altitude text box.
private void TarAlt_TextChanged(object sender, EventArgs e)
{
    try
    {
        if (TarAlt.Text != "")
        {
            if (int.Parse(TarAlt.Text) >= 0){}
        }
    }
    catch
    {
        MessageBox.Show("Invalid altitude!!!");
        TarAlt.Text = "";
    }
```

```
}

...

// Update local target position and display it.
public void UpdateCurrentTarget(float lat, float lng, float alt)
{
    Clat = lat;
    Clng = lng;
    Calt = alt;
    Ctar_lat.Text = "Lat: " + Clat.ToString();
    Ctar_long.Text = "Long: " + Clng.ToString();
    Ctar_alt.Text = "Alt: " + Calt.ToString();
}

...

// Run when clicking the "send" button for the target position.
private void SendTarget(object sender, EventArgs e)
{
    if (comPort.BaseStream.IsOpen)
    {
        int templat = (int)(10000000*float.Parse(tar_lat));
        int templng = (int)(10000000*float.Parse(tar_lng));
        int tempalt = (int)(100*float.Parse(TarAlt.Text));


        if (tempalt < 0)
            MessageBox.Show("Bad altitude");
        else if (templat == 0 || templat >= 1800000000 || templat < ↩
            -1800000000)
            MessageBox.Show("Bad latitude");
        else if (templng == 0 || templat > 900000000 || templat < ↩
            -900000000)
            MessageBox.Show("Bad longitude");
        else if (MainV2.comPort.BaseStream.IsOpen)
        {
            try
            {
                MainV2.comPort.setParam("CAMERA_LAT", templat);
                MainV2.comPort.setParam("CAMERA_LNG", templng);
                MainV2.comPort.setParam("CAMERA_ALT", tempalt);
            }
            catch (Exception ex) { Console.WriteLine(ex); }
        }
        try
        {
            MainV2.comPort.param = MainV2.comPort.getParamList();
            UpdateCurrentTarget(((float)MainV2.comPort.param["↩
                CAMERA_LAT"]) / 10000000,
                                ((float)MainV2.comPort.param["↩
                                    CAMERA_LNG"]) / 10000000,
                                ((float)MainV2.comPort.param["↩
                                    CAMERA_ALT"]) / 100);

            float tempmsg1 = ((float)MainV2.comPort.param["↩
```

```
                CAMERA_LAT"]) / 10000000;
            float tempmsg2 = ((float)MainV2.comPort.param["↩
                CAMERA_LNG"]) / 10000000;
            float tempmsg3 = ((float)MainV2.comPort.param["↩
                CAMERA_ALT"]) / 100;

            string tempfinalmsg = "Lat: " + tempmsg1.ToString() + "↩
                long: " + tempmsg2.ToString() + "alt: " + tempmsg3.↩
                ToString();
            MessageBox.Show(tempfinalmsg);
        }
        catch (Exception ex) { Console.WriteLine(ex); }

    }
    else
        MessageBox.Show("Tricopter is not connected!");


}
```

At line 383 in `MainLoop()`:

```
// Updating lat/long displayed text.
TarLat.Text = "Lat: " + tar_lat;
TarLng.Text = "Long: " + tar_lng;
```

F.3 Box and target shared functionality

In `FlightData.cs`:

```
// Run when clicking the button to get current values for
// target position and virtual box size from the tricopter.
private void click_getBoxTar(object sender, EventArgs e)
{
    if (MainV2.comPort.BaseStream.IsOpen)
    {
        try
        {
            UpdateBoxSize((float)MainV2.comPort.param["BOX_SIZE"]);
            UpdateCurrentTarget(((float)MainV2.comPort.param["↩
                CAMERA_LAT"]) / 10000000,
                                ((float)MainV2.comPort.param["↩
                                    CAMERA_LNG"]) / 10000000,
                                ((float)MainV2.comPort.param["↩
                                    CAMERA_ALT"]) / 100);
        }
        catch (Exception ex) { Console.WriteLine(ex); }
    }
}
```

F.4 Modified map functionality

In `FlightData.cs` and `FlightPlanner.cs`:

```
//Modified function to support drawing target marker.
// Last parameter "isred" added to notify if the marker added should be red instead of green.
private void addpolygonmarker(string tag, double lng, double lat, int alt, bool isred = false)
{
    try
    {
        PointLatLng point = new PointLatLng(lat, lng);
        //***added code***
        // Red marker for the target
        GMapMarkerGoogleRed m2 = new GMapMarkerGoogleRed(point);
        m2.ToolTipMode = MarkerTooltipMode.Always;
        m2.ToolTipText = tag;
        m2.Tag = tag;
        //***added code end***
        GMapMarkerGoogleGreen m = new GMapMarkerGoogleGreen(point);
        m.ToolTipMode = MarkerTooltipMode.Always;
        m.ToolTipText = tag;
        m.Tag = tag;

        //ArdupilotMega.GMapMarkerRectWPRad mBorders = new ArdupilotMega.GMapMarkerRectWPRad(point, (int)float.Parse(TXT_WPRad.Text), MainMap);
        GMapMarkerRect mBorders = new GMapMarkerRect(point);
        {
            mBorders.InnerMarker = m;
            mBorders.wprad = (int)float.Parse(TXT_WPRad.Text);
            mBorders.MainMap = MainMap;
        }

        // Check if adding a green or red (target) marker.
        //***added code***
   if (isred)
            objects.Markers.Add(m2);
        else
        {
   //***added code end***
            objects.Markers.Add(m);
            objects.Markers.Add(mBorders);
        } //***added  bracket***
    }
    catch (Exception) { }
}
```

In `FlightPlanner.cs`:

```
// Added booleans.
static bool addtarget = false; // True if adding target and not WP.
static bool onTargetMarker = false; //True if mouse over the target marker and not WP marker.
```

```
...

void MainMap_OnMarkerLeave(GMapMarker item)
{
    if (!isMouseDown)
    {
        //***code added***
        // Test if targetmarker.
        if (item.Tag.ToString() == "Target")
        {
            onTargetMarker = false;
        }
        else
        //***added code end***
        if (item is GMapMarkerRect)
        {
            CurentRectMarker = null;

            GMapMarkerRect rc = item as GMapMarkerRect;
            rc.Pen.Color = Color.Blue;
            MainMap.Invalidate(false);
        }
    }
}

...

void MainMap_OnMarkerEnter(GMapMarker item)
{
    if (!isMouseDown)
    {
        //***code added***
        // Test if target marker.
        if (item.Tag.ToString() == "Target")
        {
            onTargetMarker = true;
        }
        else
        //***added code end***
        if (item is GMapMarkerRect)
        {
            GMapMarkerRect rc = item as GMapMarkerRect;
            rc.Pen.Color = Color.Red;
            MainMap.Invalidate(false);

            CurentRectMarker = rc;
        }
    }
}

...

void MainMap_MouseUp(object sender, MouseEventArgs e)
{

 ...
```

```
    if (isMouseDown) // mouse down on some other object and dragged to here.
    {
        if (e.Button == MouseButtons.Left)
        {
            isMouseDown = false;
        }
    //***code added***
        // Test if target marker
        if (onTargetMarker)
            onTargetMarker = false;
    //***added code end***

        if (!isMouseDraging)
        {
            if (CurentRectMarker != null)
            {
                // cant add WP in existing rect
            }
            //***code added***
            // Place target on the map if left-click while shift is held down.
            else if (addtarget && Control.ModifierKeys == Keys.Shift)
            {
                targetloc.lat = (int)(currentMarker.Position.Lat*10000000);
                targetloc.lng = (int)(currentMarker.Position.Lng*10000000);
                writeKML();
                FlightData.tar_lat = currentMarker.Position.Lat.ToString();
                FlightData.tar_lng = currentMarker.Position.Lng.ToString();
                addtarget = false;
            }
            //***added code end***
            else
            {
                callMe(currentMarker.Position.Lat, currentMarker.Position.Lng, 0);
            }
        }
        else

...

}

...

void MainMap_MouseDown(object sender, MouseEventArgs e)
{
    start = MainMap.FromLocalToLatLng(e.X, e.Y);
```

```
    if (e.Button == MouseButtons.Left && Control.ModifierKeys != ↩
        Keys.Alt)
    {

        isMouseDown = true;
        isMouseDraging = false;
        //***code added***
        // Place target on map if left-click while shift is held ↩
            down.
        if (Control.ModifierKeys == Keys.Shift)
        {
            addtarget = true;
        }
        /***added code end***/
        if (currentMarker.IsVisible)
        {
            currentMarker.Position = MainMap.FromLocalToLatLng(e.X, ↩
                e.Y);
        }
    }
}

...

// move current marker with left holding
void MainMap_MouseMove(object sender, MouseEventArgs e)
{

...

    //draging
    if (e.Button == MouseButtons.Left && isMouseDown)
    {
        isMouseDraging = true;
        //***code added***
        // Moving the target marker.
        if (onTargetMarker)
        {
            targetloc.lat = (int)(currentMarker.Position.Lat * ↩
                10000000);
            targetloc.lng = (int)(currentMarker.Position.Lng * ↩
                10000000);
            FlightData.tar_lat = currentMarker.Position.Lat.ToString↩
                ();
            FlightData.tar_lng = currentMarker.Position.Lng.ToString↩
                ();
            writeKML();
        }
        else
        //***added code end***
        if (CurentRectMarker == null) // left click pan

...

    }
}
```

At line 394 in function `FlightPlanner()`:

```
// Target initial position on map.
targetloc.lat = (int)(58.39845*10000000);
targetloc.lng = (int)(15.57792*10000000);
targetloc.alt = (int)(1*100);
```

At line 865 in function `WriteKML()`:

```
// Line added for drawing the target marker on the map.
addpolygonmarker("Target", (double)targetloc.lng / 10000000, (double↩
    )targetloc.lat / 10000000, 0, true);
```

In `FlightData.cs`, at line 401 in function `MainLoop()`:

```
// Draw target marker on the map.
addpolygonmarker("Target", (double)Clng, (double)Clat, (int)(Calt * ↩
    100), true);
```

F.5 Video functionality

In `FlightData.cs`:

```
// Booleans to show if recording is active and if recording video ↩
    window or attitude window.
static bool recording = false;
static bool recordcam = false;
// Video window
public static hud.HUD mycam = null;

...

// Signal that it is the attitude window we are starting to record.
private void hud_mouseover(object sender, EventArgs e)
{
    if (!recording)
        recordcam = false;
}

// Signal that it is the video window we are starting to record.
private void cam_mouseover(object sender, EventArgs e)
{
    if (!recording)
        recordcam = true;
}
```

In function `FlightData()`:

```
// hud2 is video window object in the graphic design.
mycam = hud2;
```

At line 262 in function `Mainloop()`:

```
//***code added***
// Condition added to decide if to record attitude window or video ↩
    window.
if (recordcam)
{
    hud2.streamjpgenable = true;
    aviwriter.avi_add(hud2.streamjpg.ToArray(), (uint)hud2.streamjpg↩
        .Length);
    aviwriter.avi_end(hud2.Width, hud2.Height, 10);
}
else//***added else cond(contains the original code though)***
{
    // Original code.
    hud1.streamjpgenable = true;
    // add a frame
    aviwriter.avi_add(hud1.streamjpg.ToArray(), (uint)hud1.streamjpg↩
        .Length);
    // write header - so even partial files will play
    aviwriter.avi_end(hud1.Width, hud1.Height, 10);
}
//***added code end***
```

At line 859 in function `cam_camimage(...)`:

```
// Display video in the appropriate window.
hud2.bgimage = camimage; //***modified***
```

At line 1229 in function `recordHudToAVIToolStripMenuItem_Click(...)`:

```
// Line added to show that recording is on.
recording = true;
```

At line 1248 in function `stopRecordToolStripMenuItem_Click(...)`:

```
// Line added to show that recording is off.
recording = false;
```

In `Configuration.cs`:

```
// Boolean to show if camera window is active.
public static bool is_camera_on = false;
```

At line 75 in function `Configuration_Load(...)`:

```
// This line was changed so that the configuration setting
// for hud overlay is applied to the video window and not
// the attitude window.
CHK_hudshow.Checked = GCSViews.FlightData.mycam.hudon;
```

At line 629 in function `BUT_videostart_Click(...)`:

```
// Line added to signal that the camera window is active.
is_camera_on = true;
```

At line 646 in function `BUT_videostop_Click(...)`:

```
// Line added to signal that the camera window is not active.
is_camera_on = false;
```

At line 665 in function `CHK_hudshow_CheckedChanged(...)`:

```
// This line was changed so that the configuration setting
// for hud overlay is applied to the video window and not
// the attitude window.
GCSViews.FlightData.mycam.hudon = CHK_hudshow.Checked;
```

In `HUD.cs`:

At line 624 in `doPaint()`:

```
// Original code has been commented/inactivated.
/*if (hudon == false)
    {
        return;
    }*/
}
// Added condition to only draw attitude("hud") overlay if it is ↩
    activated in "Configuration" tab.
if (hudon)
{

...

}
// Else draw an image.
else if (!ArdupilotMega.GCSViews.Configuration.is_camera_on)
{
    _bgimage = global::ArdupilotMega.Properties.Resources.↩
        camera_image;
}
```

References

[1] v/a, *arducopter – Arduino-based autopilot for mulitrotor craft, from quadcopters to traditional helis*. http://code.google.com/p/arducopter/, 2011-09-02.

[2] v/a, *ardupilot-mega – Official ArduPilot Mega repository*. http://code.google.com/p/ardupilot-mega/wiki/Mission, 2011-09-05.

[3] v/a, *FlightGear – sophisticated, professional, open-source flight simulation*. http://www.flightgear.org/, 2011-09-09.

[4] Atmel, *8-bit Atmel Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash*. http://www.atmel.com/dyn/resources/prod_documents/doc2549.pdf, may 2011.

[5] Barsk, Karl-Johan, *Requirement specification - Tricopter with stabilized camera version 1.1*. oct 2011.

[6] Barsk, Karl-Johan, *User manual - Tricopter with stabilized camera version 1.0* .

[7] v/a, *Quad Telemetry – XBee*. http://code.google.com/p/arducopter/wiki/Quad_TelemetryPage, 2011-11-14.

[8] v/a, *Knowledge Base Article - X-CTU (XCTU) software - Support - Digi International*. http://www.digi.com/support/kbase/kbaseresultdetl.jsp?kb=125, 2011-11-17.

[9] v/a, *NOAA- National Geophysical Data Center*. http://www.ngdc.noaa.gov/geomagmodels/Declination.jsp, 2011-12-06.