AeroQuad Flight Software

Developer’s Guide

# Introduction

The AeroQuad Flight Software uses a mixture of C/C++ to accommodate multiple hardware options and algorithms within the Arduino platform. The motivation behind using a mixture of C/C++ is to find the right balance between flexibility and commonality within the restrictive programming space and processing speed of a microcontroller environment. This also allows the end user to have a methodical way to add new hardware capabilities with minimal impact to the existing flight algorithms. As a result upgrading to new improved hardware sensors can be done easily and a clear defined method for users to customize and improve the flight software itself can be achieved. The purpose of this guide is to document the software architecture implemented, provide a description of what each component does, and to give a guideline on how to contribute new features for the future.

The software architecture documented here is considered a work in progress. With the possibility of larger and faster processors available in the future and with the general understanding and experience in using Object Oriented programming in the AeroQuad community, a more comprehensive C++ architecture can be developed. This current architecture is considered a learning step in this direction.

The AeroQuad Flight Software is provided as an open source project written within the Arduino development environment. Where possible it uses the Arduino libraries, but may rely on low level ATmega microcontroller programming for optimization of certain functions.

# Software Architecture

Each of the main functions depicted in dark blue in Figure 1 are kept in Arduino sketches (.pde files). Class definitions are placed in header files (.h) and are depicted in light blue. Header files will typically have the main class (for example Gyro) at the top of the header file and all available subclasses (such as Gyro:Gyro\_AeroQuad\_v2.0) will be listed underneath it. The following sections will describe each main function and any supporting header files needed.

## AeroQuad.pde

The AeroQuad.pde sketch contains the setup and main loop of the flight software. It’s primary responsibility is to maintain the timing that each of the main functions are to execute at.

## SerialCom.pde

This sketch receives external serial commands and responds to telemetry requests.

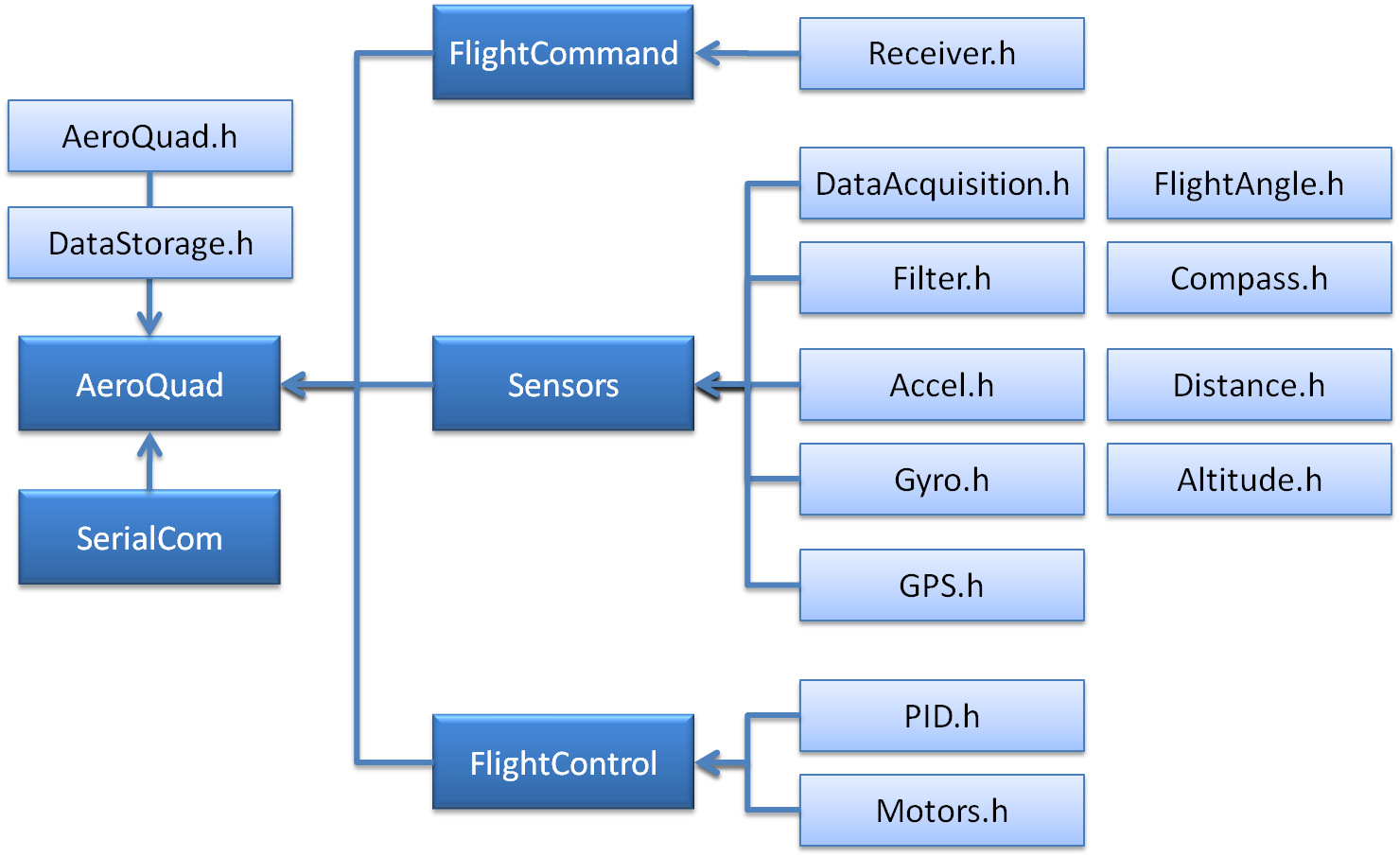


Figure - Software Architecture

## FlightCommand.pde

FlightCommand.pde is responsible for decoding transmitter stick combinations and for setting up AeroQuad modes such as motor arming/disarming and Acro/Stable flight modes. This function relies on Receiver.h for receiving radio controlled signals from the pilot. Future classes are planned to receive pilot commands over wireless link from a laptop or mobile device.

## Sensors.pde

The Sensors.pde sketch is responsible for taking on-board sensor measurements and calculating flight attitude. The following header files are used by Sensors.pde:

* DataAcquisition.h – container for certain hardware configurations where a common sensor measurement call is needed. Examples of this are I2C communication for Wii sensors, SPI ADC communication for the APM and APM IMU Sensor.
* Filter.h – contains a simple low pass filter to remove noise from sensor measurements
* Accel.h – defines how to measure accelerometer data and convert to engineering units
* Gyro.h – defines how to measure gyro data and convert to engineering units
* GPS.h – defines how to communicate with a GPS and decode NMEA strings
* FlightAngle.h – contains multiple algorithms to calculate multicopter vehicle attitude
* Compass.h – defines how to measure magnetometer or similar sensor data
* Distance.h – defines how to measure sonar or IR data
* Altitude.h – defines how to measure barometer or similar sensor data

## FlightControl.pde

This sketch combines sensor measurements and transmitter commands into motor commands for the defined flight configuration (X, +, etc.). The following header files are used by FlightControl.pde

* PID.h – contains the PID implementation used for control of the multicopter
* Motors.h – defines which motor control method is used (for example PWM or I2C).

# Class Definitions

This section will describe each class defined in the AeroQuad Flight Software. Any new subclasses must conform to the methods (or function calls) defined in the main class. The header files are listed in alphabetical order below. Each section will list the available methods or function calls used by the flight software. If any new commands are required for a class, please submit requests to the AeroQuad Google Code issue tracking list at: <http://add.me.later.com>

Please note: Only header files that defines a class used within the flight software are listed below. There are several header files found in the top level architecture that contain functions not organized as a class. This was done since they typically just contain a single function call.

## Accel.h

This class defines how the flight software interacts with an accelerometer. The following function calls must be re-defined in each subclass:

* initialize() – initializes how the microcontroller reads measurements from the accelerometer
* measure() – performs sensor measurement from the accelerometer
* getFlightData(axis) – returns modified raw sensor data for use in control algorithms

The following function calls are common to all accelerometer objects:

* \_initialize(roll channel, pitch channel, Z axis channel) – reads accelerometer calibration data (used to center measurements around zero) and where applicable assigns A/D channels to each axis
* getRaw(axis) – returns the A/D value centered around zero.
* getData(axis) – returns the A/D value centered around zero with smoothing applied
* invert(axis) – execute once to invert the accelerometer axis
* getZero(axis) – returns the raw A/D value that defines zero output from the accelerometer
* setZero(axis) – stores a new A/D value that defines zero output from the accelerometer
* getScaleFactor() – returns the scale factor used to convert A/D measurements to G
* getSmoothFactor(factor) – gets the smoothing factor used for low pass filtering (values between 0 and 1, with 1 defining no filtering used)
* setSmoothFactor(factor) – sets the smoothing factor used for low pass filtering of sensor data
* angleRad(axis) – returns the angle in radians that is calculated for the desired axis
* angleDeg(axis) – returns the angle in degress that is calculated for the desired axis

## Altitude.h

Not yet implemented in the v2.0 flight software.

## Compass.h

Not yet implemented in the v2.0 flight software.

## Distance.h

Not yet implemented in the v2.0 flight software.

## FlightAngle.h

This class defines how to calculate vehicle attitude. The following function calls must be re-defined in each subclass:

* initialize() – initializes the starting values required to calculate attitude
* calculate() – returns the vehicle attitude
* getGyroAngle(axis) – returns the estimated angle calculated from gyro data (verify if can be deleted)

The following function calls are common to all accelerometer objects:

* getData(axis) – returns the calculated angle for the requested axis
* getType() – returns the angle estimation algorithm used data (verify if can be deleted)

## GPS.h

Under Construction

## Gyro.h

This class defines how the flight software interacts with a gyro. The following function calls must be re-defined in each subclass:

* initialize() – initializes how the microcontroller reads measurements from the gyro
* measure() – performs sensor measurement from the gyro
* autoZero() – measures the A/D value that corresponds to a zero angular rate
* getFlightData(axis) – returns modified raw sensor data for use in control algorithms

The following function calls are common to all gyro objects:

* \_initialize(roll channel, pitch channel, Z axis channel) – reads gyro calibration data (used to center measurements around zero) and where applicable assigns A/D channels to each axis
* getRaw(axis) – returns the A/D value centered around zero.
* getData(axis) – returns the A/D value centered around zero with smoothing applied
* invert(axis) – execute once to invert the accelerometer axis
* setData(axis) – sets gyroData[axis] (verify if can be deleted)
* getZero(axis) – returns the raw A/D value that defines zero output from the gyro
* setZero(axis) – stores a new A/D value that defines zero output from the gyro
* getScaleFactor() – returns the scale factor used to convert A/D measurements to G
* getSmoothFactor(factor) – gets the smoothing factor used for low pass filtering (values between 0 and 1, with 1 defining no filtering used)
* setSmoothFactor(factor) – sets the smoothing factor used for low pass filtering of sensor data
* rateDegPerSec(axis) – returns the angular rate in degrees/sec that is calculated for the desired axis
* rateRadPerSec(axis) – returns the angular rate in radians/sec that is calculated for the desired axis

## Motors.h

This class defines the method to use for motor control. The following function calls must be re-defined in each subclass:

* initialize() – initializes motor control
* write() – commands each motor to values set by setMotorCommand(motor, value)
* commandAllMotors(value) – command all motors to the same value

The following function calls are common to all motor objects:

* setRemoteCommand(motor, value)
* getRemoteCommand(motor)
* getMotorSlope()
* getMotorOffset()
* setMinCommand(motor, value)
* getMinCommand(motor)
* setMaxCommand(motor, value)
* getMaxCommand(motor)
* setMotorAxisCommand(motor, value)
* getMotorAxisCommand(motor)
* setMotorCommand(motor, value)
* getMotorCommand(motor)
* setThrottle(value)
* getThrottle()

## Receiver.h

This class defines the method to use for decoding R/C receiver data. The following function calls must be re-defined in each subclass:

* initialize() – initializes receiver decode
* read() – reads R/C receiver data for all channels

The following function calls are common to all motor objects:

* \_initialize()
* getRaw(channel)
* getData(channel)
* getZero(channel)
* setZero(channel, value)
* getSmoothFactor(channel)
* setSmoothFactor(channel)
* getXmitFactor()
* setXmitFactor(value)
* getTransmitterSlope(channel)
* setTransmitterSlope(channel, value)
* getTransmitterOffset(channel)
* setTransmitterOffset(channel, value)
* getAngle(channel)

# Customizing Code

Under Construction

## How to Add a New Class

class Accel {  
public:  
 float accelScaleFactor;  
 float smoothFactor;  
 int accelChannel[3];  
 int accelZero[3];  
 int accelData[3];  
 int accelADC[3];  
 int sign[3];  
 byte rollChannel, pitchChannel, zAxisChannel;

Accel(void) {  
 sign[ROLL] = 1;  
 sign[PITCH] = 1;  
 sign[YAW] = 1;  
 }

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // The following function calls must be defined in any new subclasses  
 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

virtual void initialize(void) {  
 this->\_initialize(rollChannel, pitchChannel, zAxisChannel);  
 smoothFactor = readFloat(ACCSMOOTH\_ADR);  
 }  
 virtual void measure(void);  
 virtual void calibrate(void);  
 virtual const int getFlightData(byte);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // The following functions are common between all Gyro subclasses  
 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void \_initialize(byte rollChannel, byte pitchChannel, byte zAxisChannel) {

accelChannel[ROLL] = rollChannel;

accelChannel[PITCH] = pitchChannel;

accelChannel[ZAXIS] = zAxisChannel;

accelZero[ROLL] = readFloat(LEVELROLLCAL\_ADR);

accelZero[PITCH] = readFloat(LEVELPITCHCAL\_ADR);

accelZero[ZAXIS] = readFloat(LEVELZCAL\_ADR);

}

const int getRaw(byte axis) {

return accelADC[axis] \* sign[axis];

}

const int getData(byte axis) {

return accelData[axis] \* sign[axis];

}

const int invert(byte axis) {

sign[axis] = -sign[axis];

return sign[axis];

}

const int getZero(byte axis) {

return accelZero[axis];

}

void setZero(byte axis, int value) {

accelZero[axis] = value;

}

const float getScaleFactor(void) {

return accelScaleFactor;

}

const float getSmoothFactor() {

return smoothFactor;

}

void setSmoothFactor(float value) {

smoothFactor = value;

}

const float angleRad(byte axis) {

if (axis == PITCH) return arctan2(accelData[PITCH] \* sign[PITCH], sqrt((long(accelData[ROLL]) \* accelData[ROLL]) + (long(accelData[ZAXIS]) \* accelData[ZAXIS])));

if (axis == ROLL) return arctan2(accelData[ROLL] \* sign[ROLL], sqrt((long(accelData[PITCH]) \* accelData[PITCH]) + (long(accelData[ZAXIS]) \* accelData[ZAXIS])));

}

const float angleDeg(byte axis) {

return degrees(angleRad(axis));

}

};

## How to Add a New Sub Class

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\* AeroQuad v1 Accelerometer \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#if defined(AeroQuad\_v1) || defined(AeroQuadMega\_v1) || defined(Multipilot) || defined(MultipilotI2C)

class Accel\_AeroQuad\_v1 : public Accel {

private:

int findZero[FINDZERO];

public:

Accel\_AeroQuad\_v1() : Accel(){

// Accelerometer Values

// If BMA180 setup for 1G

// Page 27 of datasheet = 0.00013g/LSB

accelScaleFactor = 0.00013;

}

void initialize(void) {

// rollChannel = 1

// pitchChannel = 0

// zAxisChannel = 2

this->\_initialize(1, 0, 2);

smoothFactor = readFloat(ACCSMOOTH\_ADR);

}

void measure(void) {

for (axis = ROLL; axis < LASTAXIS; axis++) {

accelADC[axis] = analogRead(accelChannel[axis]) - accelZero[axis];

accelData[axis] = smooth(accelADC[axis], accelData[axis], smoothFactor);

}

}

const int getFlightData(byte axis) {

return getRaw(axis);

}

// Allows user to zero accelerometers on command

void calibrate(void) {

for (byte calAxis = ROLL; calAxis < ZAXIS; calAxis++) {

for (int i=0; i<FINDZERO; i++)

findZero[i] = analogRead(accelChannel[calAxis]);

accelZero[calAxis] = findMode(findZero, FINDZERO);

}

accelZero[ZAXIS] = (accelZero[ROLL] + accelZero[PITCH]) / 2;

writeFloat(accelZero[ROLL], LEVELROLLCAL\_ADR);

writeFloat(accelZero[PITCH], LEVELPITCHCAL\_ADR);

writeFloat(accelZero[ZAXIS], LEVELZCAL\_ADR);

}

};

#endif