Version 1.0
9/10/2010

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.

Download Software

The latest version of the Flight Software can be downloaded from:
http://code.google.com/p/aeroquad/downloads/list

If you wish to locate an older version of the software, follow the link above and in the Search pull down menu, select "deprecated downloads".

Software Support

The best place to get quick feedback or to discuss the flight software on-line is at the AeroQuad forums:
http://aeroquad.com/forum.php

There is a flight software specific board at:
http://aeroquad.com/forumdisplay.php?7-AeroQuad-Flight-Software

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.

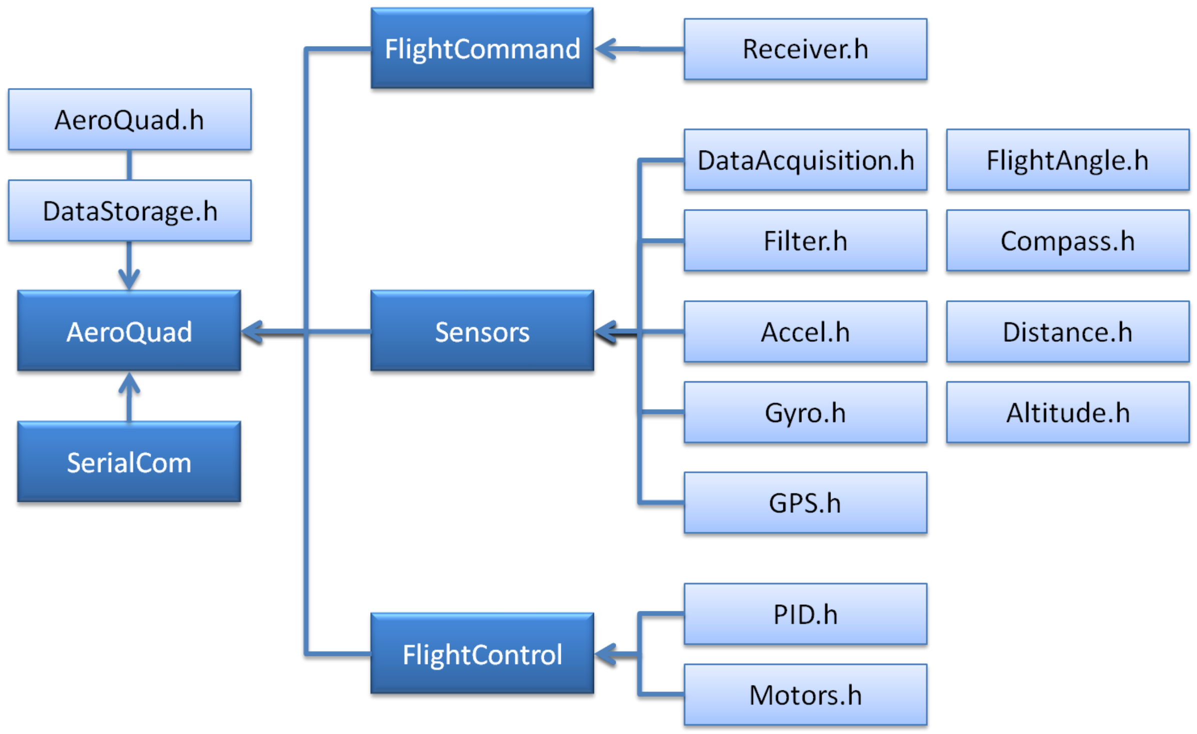

Figure 1- Software Architecture

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.

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 functions are required for a main class, please submit requests to the AeroQuad Google Code issue tracking list at: http://code.google.com/p/aeroquad/issues/list

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) – sets the value sent to motor over serial command
- getRemoteCommand(motor) – gets the value sent from a serial command
- getMotorSlope() – returns the slope (y=mx+b) of equation used to convert PWM to PPM duty cycle for motor command using analogWrite() (verify if can be deleted)
- getMotorOffset() – returns offset (y=mx+b) of equation used to convert PWM to PPM duty cycle for motor command using analogWrite() (verify if can be deleted)
- setMinCommand(motor, value) – sets the minimum allowable command to send to the motor
- getMinCommand(motor) – gets the minimum value that can be sent to the motor
- setMaxCommand(motor, value) – sets the maximum allowable command to send to the motor
- getMaxCommand(motor) – gets the maximum value that can be sent to the motor
- setMotorAxisCommand(axis, value) – sets the motor command to send for the specified axis. This is calculated from the PID for a specific axis.
- getMotorAxisCommand(motor) – gets the motor command for the specified axis
- setMotorCommand(motor, value) – commands the actual motor command to send to each motor. This is calculated using a combination of getMotorAxisCommand() values and the motor configuration that is setup (+, X, etc.)
- getMotorCommand(motor) – gets the value command for the specified motor
- setThrottle(value) – sets the throttle value used for the mixer table
- getThrottle() – gets the throttle value used for the mixer table

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, scales it using y=mx+b, smooths it, and reduces it by the transmitter factor (defined from the Configurator) and centers it around zero (with the exception of throttle, gear and aux channels)

The following function calls are common to all motor objects:

- _initialize() – loads in all calibration values and all smoothing values for each receiver channel
- getRaw(channel) – returns the receiver value scaled by y=mx+b for the specified channel
- getData(channel) – returns the smoothed receiver value which is reduced by the transmitter factor defined from the Configurator for the specified channel
- getZero(channel) – returns the value that represents zero pilot input (verify if can be deleted)
- setZero(channel, value) – sets the value that represents zero pilot input (verify if can be deleted)
- getSmoothFactor(channel) – get the smooth value applied for the specified channel
- setSmoothFactor(channel) – sets the smoothing value applied for the specified channel
- getXmitFactor() – gets the value used to make pilot input less sensitive
- setXmitFactor(value) – sets the value to use to make the pilot input less sensitive
- getTransmitterSlope(channel) – gets the slope (y=mx+b) used to scale the receiver input to 1000-2000 microseconds (PWM)
- setTransmitterSlope(channel, value) – sets the slope used to scale receiver input. This is defined from transmitter calibration of the Configurator
- getTransmitterOffset(channel) – gets the offset (y=mx+b) used to scale the receiver input to 1000-2000 microseconds (PWM)
- setTransmitterOffset(channel, value) – sets the offset used to scale receiver input. This is defined from transmitter calibration of the Configurator
- getAngle(channel) – converts the roll/pitch/yaw stick inputs to represent +/-45 degrees

Customizing Code

If you wish to add new functionality for an existing class or to create a new class, use the following templates so that new objects can be added using a common method. If we use a similar method for adding new software functions, it will be easier for all contributing developers to read through the code and debug as necessary. Also, please place any new classes inside its own header file (.h) with the same name as the main class. Include all it's subclasses at the end of the header file.

What would be considered a class or a subclass? As an example, in general multicopters need to have a gyro for basic flight. You may want to have support for a gyro that has an analog output, or a gyro that uses I2C. So in this case we would create a class called gyro, then the subclasses could be called gyroAnalog and gyroI2C. The gyro class would contain function calls common to both the subclasses , like calculateRate(), while the subclasses would have the specific function calls for measuring analog input data or retrieving I2C data. Refer to the previous section (Class Definitions) for specific examples of classes and subclasses.

Class Example / Template

```
class exampleClass {
public:
  int exampleVariable;
  float exampleData[3];

  exampleClass(void) {
    // this is the constructor of the object and must have the same name
    // can be used to initialize any of the variables declared above
  }

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

  virtual void initialize(void);
  virtual void exampleFunction(int);
  virtual const int getExampleData(byte);

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

  void examplePublicFunction(byte axis, int value) {
    // insert common code here
  }

  const int getPublicData(byte axis) {
    return exampleData[axis];
  }
};
```

Sub Class Example / Template

```
class exampleSubClass : public exampleClass {
private:
  int exampleArray[3];  // only for use inside this subclass
  int examplePrivateData; // only for use inside this subclass
  void examplePrivateFunction(int functionVariable) {
    // it’s possible to declare functions just for this subclass
  }

public:
  exampleSubClass() : exampleClass(){
    // this is the constructor of the object and must have the same name
    // can be used to initialize any of the variables declared above
  }

  // ***********************************************************
  // Define all the virtual functions declared in the main class
  // ***********************************************************

  void initialize(void) {
    // insert code here
  }

  void exampleFunction(int someVariable) {
    // insert code here
    examplePrivateFunction(someVariable);
  }

  const int getExampleData(byte axis) {
    // insert code here
    return exampleArray[axis];
  }
};
```

How to Use an Object

After you create your classes and subclasses, they can be called in the code as follows:

```
#include “exampleHeader.h”
exampleSubClass objectName;
int data = 0;
int output;

objectName.initialize();
objectName.exampleFunction(data);
output = objectName.getExampleData(data);
```