Back to my home page

# How to build your own Quadcopter Autopilot / Flight Controller

*By Dr Gareth Owen (drgowen@gmail.com)*

**Contents**

*Fig 1: 3DR Quadcopter*

## Introduction

This article will walk you through building your own controller whilst teaching you the details of how it works. This information is hard to find, particarly for those of us who are not aerospace engineers! Personally, it took me six months because much of my time was spent bug hunting and tuning, but with this article you can achieve the same in anywhere from a few hours to a few days. I'll teach you the pitfalls so that you don't waste your time like I did.

The first shortcut is your choice of hardware. I chose to build my own from scratch at a stage when I knew nothing of RC or how to fly - this was a mistake. I thought that I would save a few pennies by doing it myself but after lots of accidental short circuits, new microchips and sensors, I've spent a fortune! So do yourself a favour and buy the ArduPilot 2.5 control board, wire up your copter, learn RC, and how to fly, and then come back here. The board is essentially just an Arduino with some sensors connected which we will program in this article with our own software - by using it you have everything connected you'll need to get flying - you'll also be able to play with the excellent ArduCopter software.

*Fig 2: ArduPilot hardware*

The ArduPilot project is sponsored by 3D Robotics - this means that they build the hardware and sell it for a small profit, and then feed some of this profit back to the community. The hardware and software is entirely open source and anyone is free to copy it. You can buy the original from them direct, or identical copies from Hobbyking (named HKPilot) and RCTimer (named ArduFlyer).

In this article, I am going to assume you have the ArduPilot hardware which is essentially an Arduino with attached sensors. If you choose to ignore my advice and build your own hardware, or use the arduino board, then you'll need to replace the lower level code (the HAL library). I'm also going to assume you have a quadcopter in X configuration - although not a lot of work is required (just different motor mixing) to switch between +/X and octa/hexacopters, they won't be given it any substantial attention in the article. Ideally, you've already flown your quad with the ArduCopter code loaded and hence you should have your motors connected as fol ɛ direction shown.

*Fig 3: Propellor Configuration*

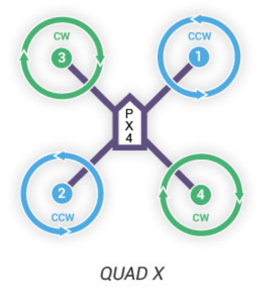


*Fig 4: ArduPilot Arduino IDE Setup*

I'n ıve some experience with the arduino - or atleast with C/C++. The arduino libraries are not particularly brilliant or well suited, so we'll be using some of the ArduPilot libraries which are superior. However, we'll be keeping their use to a minimum in favour of the DIY approach (which is why you're here after all). The first and main library that we're going to use is the ArduPilot Hardware Abstraction Layer (HAL) library. This library tries to hide some of the low level details about how you read and write to pins and some other things - the advantage is that the software can then be ported to new hardware by only changing the hardware abstraction layer. In the case of ArduPilot, there are two hardware platforms, APM and PX4, each of which have their own HAL library which allows the ArduPilot code to remain the same across both. If you later decide to run your code on the Raspberry Pi, you'll only need to change the HAL.

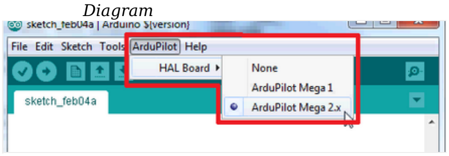
The HAL library is made up from several components:

- RCInput - for reading the RC Radio.
- RCOutput - for controlling the motors and other outputs.
- Scheduler - for running particular tasks at regular time intervals.
- Console - essentially provides access to the serial port.
- us drivers (small circuit board networks for connecting to sensors)
- erial Purpose Input/Output - allows raw access to the arduino pins, but in our case, mainly the LEDs

*Fig 5: Radio Signals*

**WHAT TO DOWNLOAD:** You'll need to download the ArduPilot version of the Arduino IDE. Also grab the libraries which should be placed in your sketches folder. Also make sure you select your board type from the Arduino menu like so:

*Diagram*

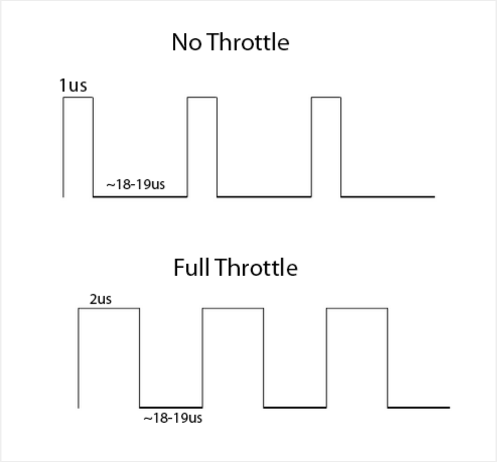


Our flight controller is going to have to read in the radio inputs (pilot commands), measure our current attitude (yaw/pitch/roll), change the motor speeds to orientate the quad in the desired way. So let's start out by reading the radio.

Back to top

## Reading the Radio Inputs

RC Radios have several outputs, one for each channel/stick/switch/knob. Each radio output transmits a pulse at 50Hz with the width of the pulse determining where the stick is on the RC transmitter. Typically, the pulse is between 1000us and 2000us long with a 18000us to 19000us pause before the next - so a throttle of 0 would produce a pulse of 1000us and full throttle would be 2000us. Sadly, most radios are not this precise so we normally have to measure the min/max pulse widths for each stick (which we'll do in a minute).



The ArduPilot HAL library does the dirty work of measuring these pulse widths for us. If you were coding this yourself, you'd have to use pin interrupts and the timer to measure them - arduino's AnalogRead isn't suitable because it holds (blocks) the processor whilst it is measuring which stops us from doing anything else. It's not hard to implement an interrupt measurer, it can be programmed in an hour or so but as it's fairly mundane we won't.

Here's some sample code for measuring the channel 'values' using the APM HAL library. The channel values are just a measure in microseconds of the pulse width.

```
#include <AP_Common.h>
#include <AP_Math.h>
#include <AP_Param.h>
#include <AP_Progmem.h>
#include <AP_ADC.h>
#include <AP_InertialSensor.h>

#include <AP_HAL.h>
#include <AP_HAL_AVR.h>

const AP_HAL::HAL& hal = AP_HAL_AVR_APM2;  // Hardware abstraction layer

void setup()
{

}

void loop()
{
  uint16_t channels[8];  // array for raw channel values

  // Read RC channels and store in channels array
  hal.rcin->read(channels, 8);

  // Copy from channels array to something human readable - array entry 0 = input 1, etc.
  uint16_t rcthr, rcyaw, rcpit, rcroll;   // Variables to store rc input
  rcthr = channels[2];
  rcyaw = channels[3];
  rcpit = channels[1];
  rcroll = channels[0];

  hal.console->printf_P(
            PSTR("individual read THR %d YAW %d PIT %d ROLL %d\r\n"),
            rcthr, rcyaw, rcpit, rcroll);

  hal.scheduler->delay(50);  //Wait 50ms
}

AP_HAL_MAIN();    // special macro that replace's one of Arduino's to setup the code (e.g. ensure loop() is called in a loop).
```

Create a new sketch and upload the code to the ardupilot hardware. Use the serial monitor and write down the minimum and maximum values for each channel (whilst moving the sticks to their extremes).

Now let's scale the stick values so that they represent something meaningful. We're going to use a function called map, which takes a number between one range and places it in another - e.g., if we had a value of 50, which was between 0-100, and we wanted to scale it to be between 0 and 500, the map function would return 250.

The map function (copied from Arduino library) should be pasted into your code after the #include and defines:

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

It is used as:

```
result = map(VALUE, FROM_MIN, FROM_MAX, TO_MIN, TO_MAX).
```

It makes sense for the throttle to remain untouched, no doubt you've calibrated your ESCs with the existing throttle values (if you followed my advice about flying first) so let's not play with it. Pitch and roll should be scaled to be between -45 degrees and +45 degrees, whilst yaw might scale to +-150 degrees.

My code in loop() now looks like follows after I've substituted in the map function with the min/max values for each stick. We'll also change the variable types to long to support negative numbers.

```
long rcthr, rcyaw, rcpit, rcroll;   // Variables to store rc input
rcthr = channels[2];
rcyaw = map(channels[3], 1068, 1915, -150, 150);
rcpit = map(channels[1], 1077, 1915, -45, 45);
rcroll = map(channels[0], 1090, 1913, -45, 45);
```

Pitch should be negative when the stick is forward and roll/yaw should be negative when the stick is left. If this isn't the case then reverse them until they are correct.

You should now print the new values out and monitor them on the serial monitor. Ideally, they should be zero or very close when the sticks (except thr) are centred. Play with the min/max values until they are. There will be some jitter (waving about the true value) because the sticks on your transmitter are analog but it should be of the order +-1 or +-2 degrees. Once you've got your quad flying, you might consider returning here to introduce an averaging filter.

Ensure that pitch forward, roll left, and yaw left are negative numbers - if they're not, put a minus sign before the map. Also ensure that the throttle increases in value as you raise the throttle.

Back to top

## Controlling the motors

Motors are controlled through the Electronic Speed Controllers (ESCs). They work on pulse widths between approximately 1000us and 2000us like the RC radio receiver - sending a pulse of 1000us typically means off, and a pulse of 2000us means fully on. The ESCs expect to receive the pulse at 50Hz normally, but most off the shelf ESCs average the last 5-10 values and then send the average to the motors. Whilst this can work on a quad, it behaves much better if we minimise the effect of this averaging filter to give near instantaneous response. Hence, the APM HAL library sends the pulse at 490Hz, meaning that the 5-10 pulses which are averaged occur very quickly largely negating the filter's effect.

In setup(), let's enable the outputs:

```
hal.rcout->set_freq(0xF, 490);
hal.rcout->enable_mask(0xFF);
```

After your includes, let's define a mapping of output number to motor name - this mapping is the same as the ArduCopter uses but the numbering starts from zero rather than one.

```
#define MOTOR_FL   2    // Front left
#define MOTOR_FR   0    // Front right
#define MOTOR_BL   1    // back left
#define MOTOR_BR   3    // back right
```

In your loop, after reading the radio inputs, let's send the radio throttle straight to one of the motors:

```
hal.rcout->write(MOTOR_FR, rcthr);
```

You can now program your quad and try it, WITHOUT propellors. Slowly raise the throttle and the front right motor should spin up. By repeating the last line for the remaining three motors, all your motors would spin up although the quad will just crash if you have propellors on because we have to do stablisation - slight differences between the motors, props, ESCs, etc mean that slightly unequal force is applied at each motor so it'll never remain level.

** Comment out the write line before proceeding for safety reasons **

Back to top

## Determining Orientation

Next, we need to determine which orientation, or attitude as it's known, the quad copter is in. We can then use this, along with the pilot's commands to vary the motor speed. There are two sensors used for determining orientation, accelerometers and gyroscopes. Accelerometers measure acceleration in each direction (gravity is an acceleration force so it gives us a direction to ground) and gyroscopes measure angular velocity (e.g. rotation speed around each axis); however, accelerometers are very sensitive to vibrations and aren't particularly quick whilst gyroscopes are quick and vibration resistant but tend do drift (e.g. show constant rotation of 1/2 degrees/sec when stationary). So, we use a sensor fusion algorithm to fuse the two together and get the best of both worlds - the scope of such an algorithm is outside the scope of this article, typically a Kalman filter is used, or in the case of ArduPilot, a Direct Cosine Matrix (DCM). I've provided the DCM link for interest if you have a maths background - for the rest of us we don't need to know the details.

Thankfully, the MPU6050 sensor chip containing the accelerometer and gyroscopes has a built in Digital Motion Processing unit (aka sensor fusion) that we can use. It will fuse the values together and present us with the result in quaternions. quaternions are a different way of representing orientation (as opposed to euler angles: yaw pitch roll) that has some advantages - if you've programmed 3d graphics you'll already be familiar with them. To make things easier, we tend to convert quaternions into Euler angles and work with them instead.

Here's the code to use the MPU6050 sensor with sensor fusion.

In setup():

```
// Disable barometer to stop it corrupting bus
hal.gpio->pinMode(40, GPIO_OUTPUT);
hal.gpio->write(40, 1);

// Initialise MPU6050 sensor
ins.init(AP_InertialSensor::COLD_START,
                 AP_InertialSensor::RATE_100HZ,
                 NULL);

// Initialise MPU6050's internal sensor fusion (aka DigitalMotionProcessing)
hal.scheduler->suspend_timer_procs();  // stop bus collisions
ins.dmp_init();
ins.push_gyro_offsets_to_dmp();
hal.scheduler->resume_timer_procs();
```

Now let's read the sensor. At the beginning of loop() add this line, which will force a wait until there is new sensor data. There's no point in changing motor speeds unless we know something new.

```
while (ins.num_samples_available() == 0);
```

** Remove the 50ms delay from the loop, no longer needed **

Now let's get the yaw/pitch/roll from the sensor and convert them from radians to degrees:

```
ins.update();
ins.quaternion.to_euler(&roll, &pitch, &yaw);
roll = ToDeg(roll) ;
pitch = ToDeg(pitch) ;
yaw = ToDeg(yaw) ;
```

Now let's print it out to the serial console:

```
hal.console->printf_P(
          PSTR("P:%4.1f  R:%4.1f Y:%4.1f\n"),
                          pitch,
                          roll,
                          yaw);
```

You need to put a rate throttle on this print statement, e.g. ensure it's only printed once every 20 times around the loop (hint: use a counter). Otherwise the serial line will get flooded.

Move your copter around and ensure the right values are changing!

Back to top

## Acrobatic / Rate mode control

Acrobatic/rate mode is where the sticks on your transmitter tell the quad to rotate at a particular rate (e.g. 50deg/sec), and when you return the sticks to center the quad stops rotating. This is as opposed to stablise mode where returning the sticks to center will level the quadcopter. It's a mode that takes practice to learn how to fly in but we are required to implement this mode first because the stablise controllers operate on top of the rate controllers.

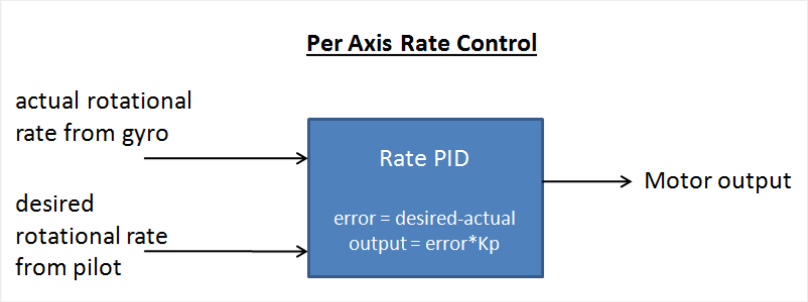
So, our aim is for each of the pilot's sticks to dictate a rate of rotation and for the quad to try to achieve that rate of rotation. So if the pilot is saying rotate 50deg/sec forward on the pitch axis, and we're currently not rotating, then we need to speed up the rear motors and slow down the front ones. The question is, by how much do we speed them up/slow them down? To decide this, you need to understand Proportional Integral Derivative (PID) controllers which we are going to make extensive use of. Whilst somewhat of a dark art, the principles are fairly straight forward. Let's assume our quadcopter is not rotating on the pitch axis at the moment, so actual = 0, and let's further assume the pilot wants the quad to rotate at 15deg/sec, so desired = 15. Now we can say that the error between what we want, and what we've got is:

```
error = desired - actual = 15 - 0 = 15
```

Now given our error, we multiply it by a constant, Kp, to produce the number which we will use to slow down or speed up the motors. So, we can say the motors change as follows:

```
frontMotors = throttle - error*Kp
rearMotors = throttle + error*Kp
```

As the motors speed up the quad will start to rotate, and the error will decrease, causing the difference between the back/rear motor speeds to decrease. This is desirable, as having a difference in motor speeds will accelerate the quad, and having no difference will cause it to hold level (in a perfect world). Believe it or not, this is all we really need for rate mode, to apply this principle to each of the axes (yaw, pitch, roll) and using the gyros to tell us what rate we're rotating at (actual). The question you're probably asking is, what should I set Kp to? Well, that's a matter for experimentation - I've set some values that work well with my 450mm quadcopter - stick with these until you've got this coded.



If you've been studying PIDs before, you'll know there are actually two other parts to a PID: integral and derivative. Integral (Ki is the tuning parameter) essentially compensates for a constant error, sometimes the Kp term might not provide enough response to get all the way if the quad is unbalanced, or there's some wind. Derivative we're going to ignore for now.

Let's get started, define the following PID array and constants globally:

```
PID pids[6];
#define PID_PITCH_RATE 0
#define PID_ROLL_RATE 1
#define PID_PITCH_STAB 2
#define PID_ROLL_STAB 3
#define PID_YAW_RATE 4
#define PID_YAW_STAB 5
```

Now initialise the PIDs with sensible values (you might need to come back and adjust these later) in the setup() function.

```
pids[PID_PITCH_RATE].kP(0.7);
//  pids[PID_PITCH_RATE].kI(1);
pids[PID_PITCH_RATE].imax(50);

pids[PID_ROLL_RATE].kP(0.7);
//  pids[PID_ROLL_RATE].kI(1);
pids[PID_ROLL_RATE].imax(50);

pids[PID_YAW_RATE].kP(2.5);
//  pids[PID_YAW_RATE].kI(1);
pids[PID_YAW_RATE].imax(50);

pids[PID_PITCH_STAB].kP(4.5);
pids[PID_ROLL_STAB].kP(4.5);
pids[PID_YAW_STAB].kP(10);
```

Leave the I-terms uncommented for now until we can get it flying OK, as they may make it difficult to identify problems in the code.

Ask the gyros for rotational velocity data for each axis.

```
Vector3f gyro = ins.get_gyro();
```

Gyro data is in radians/sec, gyro.x = roll, gyro.y = pitch, gyro.z = yaw. So let's convert these to degrees and store them:

```
float gyroPitch = ToDeg(gyro.y), gyroRoll = ToDeg(gyro.x), gyroYaw = ToDeg(gyro.z);
```

Next, we're going to perform the ACRO stablisation. We're only going to do this if the throttle is above the minimum point (approx 100pts above, mine is at 1170, where minimum is 1070) otherwise the propellors will spin when the throttle is zero and the quad is not-level.

```
if(rcthr > 1170) {   // *** MINIMUM THROTTLE TO DO CORRECTIONS MAKE THIS 20pts ABOVE YOUR MIN THR STICK ***/
        long pitch_output =   pids[PID_PITCH_RATE].get_pid(gyroPitch - rcpit, 1);
        long roll_output =   pids[PID_ROLL_RATE].get_pid(gyroRoll - rcroll, 1);
        long yaw_output =   pids[PID_YAW_RATE].get_pid(gyroYaw - rcyaw, 1);

        hal.rcout->write(MOTOR_FL, rcthr - roll_output - pitch_output);
        hal.rcout->write(MOTOR_BL, rcthr - roll_output + pitch_output);
        hal.rcout->write(MOTOR_FR, rcthr + roll_output - pitch_output);
        hal.rcout->write(MOTOR_BR, rcthr + roll_output + pitch_output);
} else {  // MOTORS OFF
        hal.rcout->write(MOTOR_FL, 1000);
        hal.rcout->write(MOTOR_BL, 1000);
        hal.rcout->write(MOTOR_FR, 1000);
        hal.rcout->write(MOTOR_BR, 1000);

        for(int i=0; i<6; i++) // reset PID integrals whilst on the ground
                pids[i].reset_I();
}
```

Now raise the throttle about 20% and rotate your quad forward/back, and left/right in your hands and make sure the correct propellors speed up/slow down - if the quad is tilted forward, then the forward propellors should speed up and the rears slow down. If not, change the signs around on the motor outputs (e.g. if the pitch is wrong, swap the signs before the pitch, likewise with the roll).

You can test this fully if you choose, and tune your rate PIDs by fixing the quad on one axis with a piece of string and testing each of the axes in turn. It's a useful experience to get a better understanding of how the rate PID is working but not strictly necessary. Here's an example of mine with rate only PIDs - I command it to rotate at 50deg/second:

**Video: Rate PIDs only with quad fixed on one axis**

Now we need to add yaw support in. As you know, two motors spin in different directions to give us yaw control. So we need to speed up / slow down the two pairs to keep our yaw constant.

```
hal.rcout->write(MOTOR_FL, rcthr - roll_output - pitch_output - yaw_output);
hal.rcout->write(MOTOR_BL, rcthr - roll_output + pitch_output + yaw_output);
hal.rcout->write(MOTOR_FR, rcthr + roll_output - pitch_output + yaw_output);
hal.rcout->write(MOTOR_BR, rcthr + roll_output + pitch_output - yaw_output);
```

This is a bit more difficult to test. You need to raise the throttle so that it hovers a little. If the yaw signs are wrong then the quad will spin.

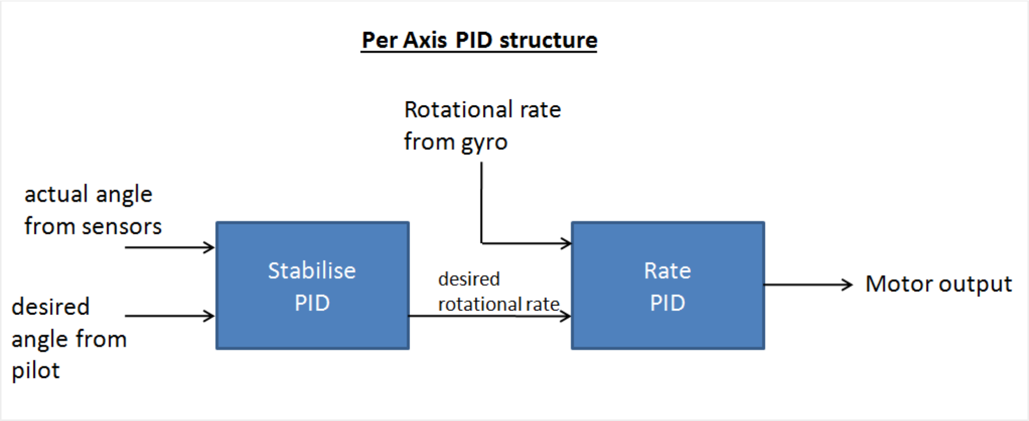
You should now be able to get your quad off the ground for a few seconds. If you're comfortable flying acro mode you will even be able to fly it - although bear in mind that this is pure acro mode, not that on ArduCopter where it performs auto-levelling for you.

If your quad flies floppy, or oscillates, then you need to adjust your rate Kp. Up if floppy or down if oscillating. If it's just going nuts, then you have the signs around the wrong way - try printing out PID outputs, and motor commands to debug whilst moving the quad around (without the battery connected).

Back to top

## Stablilised Control

Stabilised mode works similar to rate mode, except our code sits on top of the rate code as follows:



Now, the pilot's sticks dictate the angle that the quad should hold, not the rotational rate. So we can say, if the pilot's sticks are centred, and the quad is currently pitched at 20 degrees, then:

```
error = desiredAngle - actualAngle = 0 - 20 = -20
```

Now in this case, we're going to multiply error by a Kp such that the output is the angular rate to achieve. You'll notice from earlier, Kp for the stab controllers is set at 4.5. So, if we have an error of -20, then the output from the pid is -20*4.5 = -90 (the negative just indicates direction). This means the quad should try to achieve a rate of -90degrees per second to return it to level - we then just feed this into the rate controllers from earlier. As the quad starts to level, the error will decrease, the outputted target rate will decrease and so the quadcopter will initially return to level quickly and then slow down as it reaches level - this is what we want!

```
// our new stab pids
float pitch_stab_output = constrain(pids[PID_PITCH_STAB].get_pid((float)rcpit - pitch, 1), -250, 250);
float roll_stab_output = constrain(pids[PID_ROLL_STAB].get_pid((float)rcroll - roll, 1), -250, 250);
float yaw_stab_output = constrain(pids[PID_YAW_STAB].get_pid((float)rcyaw - yaw, 1), -360, 360);

// rate pids from earlier
long pitch_output =  (long) constrain(pids[PID_PITCH_RATE].get_pid(pitch_stab_output - gyroPitch, 1), - 500, 500);
long roll_output =  (long) constrain(pids[PID_ROLL_RATE].get_pid(roll_stab_output - gyroRoll, 1), -500, 500);
long yaw_output =  (long) constrain(pids[PID_YAW_RATE].get_pid(yaw_stab_output - gyroYaw, 1), -500, 500);
```

Now your quad should be able to hover, although it might be wobbly / oscillating. So, if it's not flying too great - now is the time to tune those PIDs, concentrating mainly on the rate ones (Kp in particular) - the stab ones _should_ be okay. Also turn on the rate I terms, and set them to ~1.0 for pitch/roll and nothing for yaw.

Notice that yaw isn't behaving as expected, the yaw is locked to your yaw stick - so when your yaw stick goes left 45degrees the quad rotates 45 degrees, when you return your stick to centre, the quad returns its yaw. This is how we've coded it at present, we could remove the yaw stablise controller and just let the yaw stick control yaw rate - but whilst it will work the yaw may drift and won't return to normal if a gust of wind catches the quad. So, when the pilot uses the yaw stick we feed this directly into the rate controller, when he lets go, we use the stab controller to lock the yaw where he left it.

As the yaw value goes from -180 to +180, we need a macro that will perform a wrap around when the yaw reaches -181, or +181. So define this near the top of your code:

```
#define wrap_180(x) (x < -180 ? x+360 : (x > 180 ? x - 360: x))
```

If you examine it carefully, if x is < -180, it adds +360, if it's > 180 then we add -360, otherwise we leave it alone.

Define this global or static variable:

```
float yaw_target = 0;
```

Now in the main loop, we need to feed the yaw stick to the rate controller if the pilot is using it, otherwise we use the stab controller to lock the yaw.

```
float yaw_stab_output = constrain(pids[PID_YAW_STAB].get_pid(wrap_180(yaw_target - yaw), 1), -360, 360);

if(abs(rcyaw) > 5) {  // if pilot commanding yaw
        yaw_stab_output = rcyaw;  // feed to rate controller (overwriting stab controller output)
        yaw_target = yaw;         // update yaw target
}
```

You'll also what to set your yaw target to be the direction that the quad is on the ground / throttle off - you can do this in the else part of the if.

That's it, now yaw should behave normally. Although if you pay attention, you might notice that the yaw drifts slowly over several tens of seconds. This may not bother you, the reason it is happening is because although your yaw stick is centred, the radio jitter means the quad doesn't always receive 0 - it hovers around that value causing the yaw to change. Additionally, the MPU6050's yaw sensor drifts over time (1-2deg/sec) - you'll need to use the compass to compensate for this drift (if you really care enough to fix it - most people don't notice).

Back to top

## Final Product - video and full code

Congratulations - you've built your first flight controller for a multi-copter! You'll notice it's a lot more aggresive than the standard ArduCopter code - this is because ArduCopter has a lot of processing on pilot's inputs to make it easier to fly. Raise your throttle to ~80% and your quad will _rocket_ into the sky far faster than you could achieve on ArduCopter. Be warned - don't raise your throttle too close to 100% as that won't leave any room for the controller to change the motor speeds to get it level and it'll flip (you can implement an automatic throttle lowerer fairly easily).

Download the Completed Code - this should run straight away if your regular arducopter flies (after you've adjusted the radio max/mins) - you might also need to adjust the PIDs to get it stable.

Video of the final product in action.

Back to top

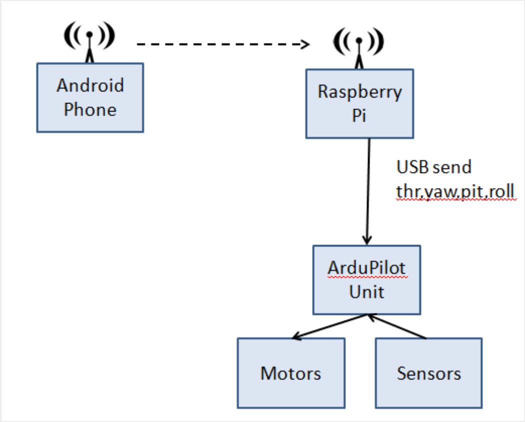
## Other ideas: Safety

- add a mechanism to arm/disarm the quadcopter.
- Ensure you've thought about what happens when there are bugs in your code - you don't want the throttle getting stuck on full! Investigate the watchdog timer.

Back to top

## Optional: Raspberry Pi

Your best bet here is to use the ArduPilot hardware as a sensor/control expansion board by connecting it to the Pi over the USB. You need to be very careful because the Pi runs Linux and as a result of this it is very difficult to do finely grained timing like controlling ESCs/reading radios. I learnt a hard lesson after choosing to do the low level control loop (PIDs) on the Pi - trying to be clever I decided to put a log write in the middle of the loop for debugging - the quad initially flied fine but then Linux decided to take 2seconds to write one log entry and the quad *almost* crashed into my car! Therefore, your best bet is to offload the time critical stuff to the ardupilot hardware and then run highlevel control on the Pi (e.g. navigation). You're then free to use a language like Python because millisecond precision isn't needed. The example I will give here is exactly that scenario.



Connect the ArduPilot to your Raspberry Pi over USB and modify the code in this article to accept THR, YAW, PIT, ROL over the serial port (sample provided below). You can then set your raspberry Pi up as a wifi access point and send your stick inputs over wireless from your phone (beware that Wifi has very short range ~30m).

Sample code

**Android App:** : Download app - sends thr, yaw, pitch, roll from pilot out on UDP port 7000 to 192.168.0.254 - you can change this in the app

**Raspberry Pi**: Download server - On the Pi, we run a python script that listens for the control packets from the Android app, and then sends them to the ArduPilot. Here I'm just implementing a simple relay, but you could easily do something more complex like navigation, control over 3G, etc.

**ArduPilot**: Download code - accepts thr, yaw, pitch and roll over the serial port rather than over the RC radio. A simple checksum is used to discard bad packets.

**Video**:

Back to top

## Optional: Autonomous Flight

Autonomous flight should now be fairly straightforward to implement. Some tips:

**GPS Navigation:** The ArduPilot provides libraries for parsing GPS data into latitude and longitude, you'll just need a PID to convert desired speed into pitch/roll, and another PID for converting distance to waypoint into desired speed. You can use the compass to work out direction to your waypoint, and then just translate that into the right amount of pitch and yaw.

**Altitude Hold**: You can sense altitude with the barometer that is build onto the ArduPilot board. You'll need two PIDs, one to calculated throttle alterations from desired ascent/descent rate and a second to calculate desired ascent/descent from distance to desired altitude.

Back to top