**Quadcopter Source Code From Scratch**

This is one of a [series of posts](http://www.benripley.com/category/diy/quadcopter/) chronicling my attempts at implementing a Multi-rotor aircraft flight control system.

Before starting this project, I knew almost noting about multi-rotor aircraft and how they work. In an effort to really understand the mechanics of the flight control system, I have decided to design and implement the software myself using an [Arduino](http://www.benripley.com/diy/arduino/getting-started-with-arduino/" \o "Getting Started With Arduino) rather than purchasing an off-the-shelf flight controller.

In [What is a Quadcopter](http://www.benripley.com/diy/quadcopter/what-is-a-quadcopter/), I covered the physics of quadcopter flight. Here, we’ll dive into a deeper understanding of how to implement this control with an Arduino.

**Controlling The Motors**

To start with, each propellor is controlled by a digital output on the Arduino by generating a modulated signal that rapidly switches the power to the motor on and off. Using the Arduino servo library writeMicroseconds function makes this quite simple. We just provide a number between 1000 and 2000 indicating the time we wish the motor to remain on (1000: Off, 2000: Full Power).

|  |
| --- |
| // Set motor output power to 50%  Servo motor0;  motor0.attach(PIN\_MOTOR0);  int half\_power = 1500;  motor0.writeMicroseconds(halfPower); |

**Reading The Sensor Values from the IMU**

Now that we can control the Motors, we need to be able to use the sensor values to determine our orientation and make sure we stay level. The IMU outputs a buffer of 12 bytes over the I2C bus on each update event. The buffer contains float values for the angles; Roll, Pitch, and Yaw as calculated by the IUM. Each float is 4 bytes. We need to convert the byte array back into a float type. For this we use [Unions](http://en.wikipedia.org/wiki/Union_%28computer_science%29#C.2FC.2B.2B). For our purposes you can think of a union as a data structure with multiple data members (float and byte[]) where each data member begins at the same location in memory. I’ve included the code below for one of the three angles.

|  |
| --- |
| float angleX; // Our Roll angle in degrees (-180 to +180)    // Read 12 bytes into data[] from I2C  wire.requestFrom(ADDR\_SLAVE\_I2C, PACKET\_SIZE);  byte data[PACKET\_SIZE);  int i = 0;  while(Wire.available())  {    data[i] = Wire.read();    i++;  }    // Use a union to convert between byte[] (b) and float (fval)  union ROL\_tag {byte b[4]; float fval;} ROL\_Union;    // Copy the appropriate bytes to our Roll union  ROL\_Union.b[0] = data[0];  ROL\_Union.b[1] = data[1];  ROL\_Union.b[2] = data[2];  ROL\_Union.b[3] = data[3];    // Copy the float value from the union to our angle variable  if(isnan(ROL\_Union.fval) != 1)  {    angleX = ROL\_Union.fval;  } |

There are four signals we need to read from the receiver; Roll, Pitch, Yaw, and Throttle. The receiver produces these [PWM](http://en.wikipedia.org/wiki/Pulse-width_modulation) signals from the joystick values specified by the operator and we read each of these signals on a digital input on the Arduino.

PWM works by varying the width of the on signal (read Duty Cycle) within a fixed signal frequency or period of time. So what we are really looking for is the length of time the signal remains high for each cycle. You can read more about different ways to read a PWM signal with an Arduino [here](http://www.benripley.com/diy/arduino/three-ways-to-read-a-pwm-signal-with-arduino/). I am using Pin Change Interrupts to trigger a specified Interrupt Service Routine (ISR) on RISING and FALLING signal edges.

|  |
| --- |
| #include  #define MY\_PIN 5 // we could choose any pin    volatile int pwm\_value = 0;  volatile int prev\_time = 0;  uint8\_t latest\_interrupted\_pin;    void setup() {   pinMode(MY\_PIN, INPUT); digitalWrite(MY\_PIN, HIGH);   PCintPort::attachInterrupt(MY\_PIN, &rising, RISING);  }    void rising()  {   latest\_interrupted\_pin=PCintPort::arduinoPin;   PCintPort::attachInterrupt(latest\_interrupted\_pin, &falling, FALLING);   prev\_time = micros();  }    void falling() {   latest\_interrupted\_pin=PCintPort::arduinoPin;   PCintPort::attachInterrupt(latest\_interrupted\_pin, &rising, RISING);   pwm\_value = micros()-prev\_time; // now we have the latest PWM value  } |

**Putting It All Together**

Now that we have the basics values in place, we can plug them into a PID Controller and adjust the motors accordingly. We will use one PID controller for each axis (pitch, roll and yaw). The setpoint is the receiver values set by the operator via the remote control. Here, the operator is specifying the desired angle of the quadcopter. The error becomes the difference between this desired angle and the angle actually measured by the IMU. The PID controller varies the output by applying an algorithm using the measured error over time to achieve a minimum output error. For more on how the PID controller works, read [here](http://www.benripley.com/diy/quadcopter/multi-rotor-control-system-part-ii/).

|  |
| --- |
| // Project the received throttle signal into a valid motor speed range  throttle=map(throttle\_rx,THROTTLE\_RMIN,THROTTLE\_RMAX,MOTOR\_ZERO\_LEVEL,MOTOR\_MAX\_LEVEL);    // Update our setpoints from the receiver. I also do some basic DSP here to  // allow for noise and sensitivity on the RX controls...  setpoint\_update();    // Update the measured angle values in the PID Controller.  pid\_update();    // Calculate PID output  pid\_compute();    // Apply our Roll and Pitch signals to the throttle value to calculate the  // new motor output values. Yaw control disabled during stabilization testing...  m0 = throttle + pid\_pitch\_out ; //+ pid\_yaw\_out;  m1 = throttle + pid\_roll\_out ;  //- pid\_yaw\_out;  m2 = throttle - pid\_pitch\_out ; //+ pid\_yaw\_out;  m3 = throttle - pid\_roll\_out ;  //- pid\_yaw\_out;    // Write the new values to the motor  update\_motors(m0, m1, m2, m3); |

**Source Code**

I hope that made sense. Take a look at the full source code [here](https://github.com/benripley/Arduino-Quadcopter).

**Flight Testing**

Here is a video of my first flight stabilization test. For testing, I’ve fixed one axis for testing so only one axis of rotation is active at a time.

This flight is only using the throttle. The Roll, Pitch, and Yaw signals on the receiver are all set to zero. The quadcopter therefore, should remain level. As you can see, the quadcopter is reading the sensor values and compensating for the measured error. However, it is severely overcompensating and also it is applying the compensation a little bit to slowly. It looks like I need to tune the PID values before trying a real flight.