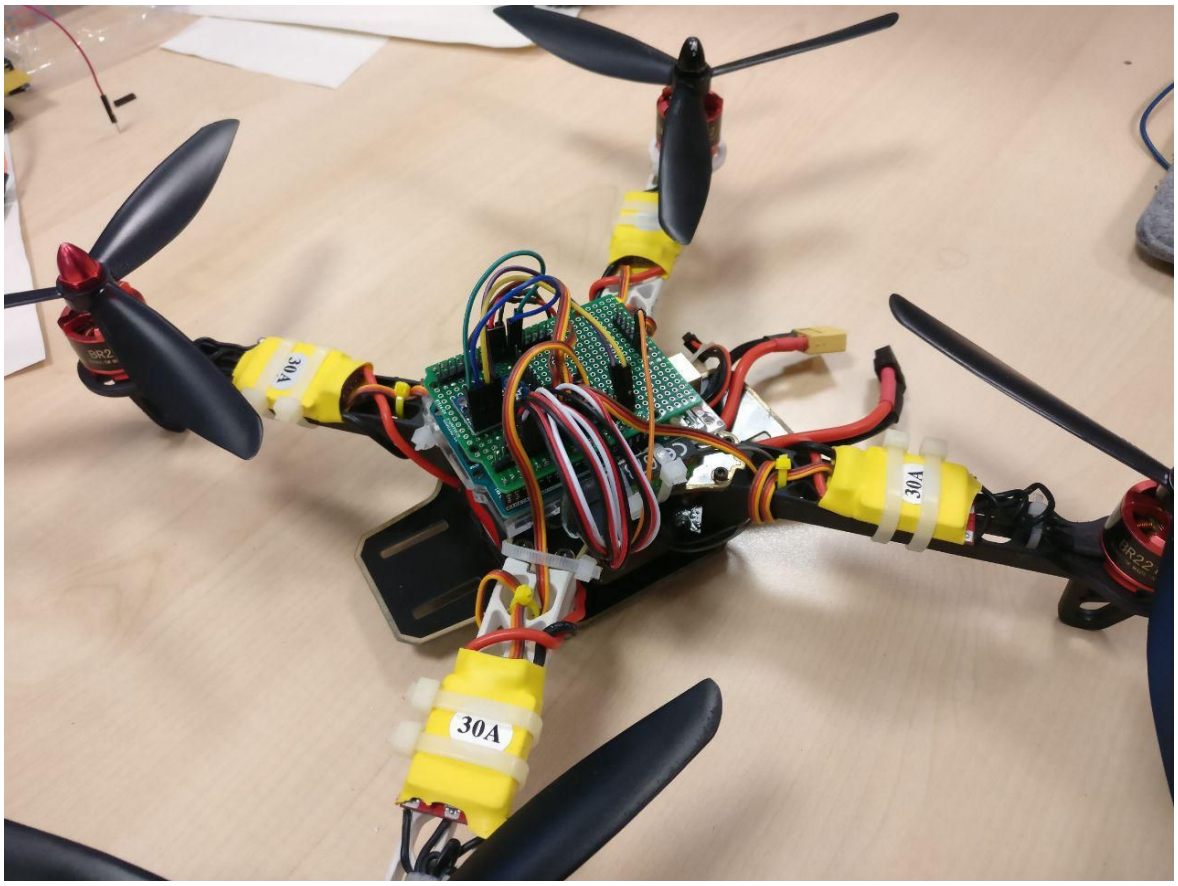
Scuola Superiore Sant' Anna

Arduino flight controller

An Arduino flight controller based on PID controller and MPU-6050 IMU



Author: Edoardo Cittadini

Email: edo.citta@gmail.com

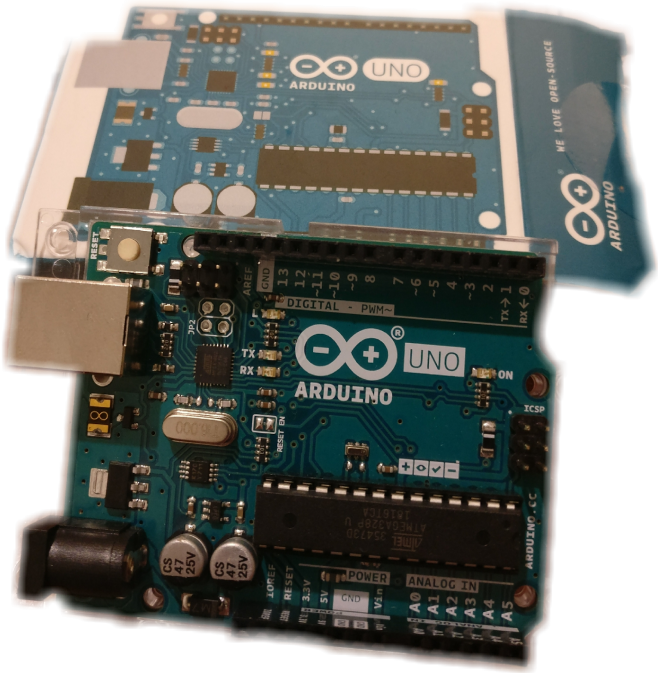
External refs: http://www.augc.it/

Index

Introduction

The project propose an implementation of a flight controller based on Arduino, in particular in all the shiel based on ATMega328p following the traces of the most important, popular and used software in the industry in oreder to guarantee the same standards, settings and ways of acting on the quadrocopter controller.

This allows the expert user to act in a familiar way with the software and at the same time allows beginners to approach the basic configurator in the same way as more advanced controllers.



The automatic position correction of the multirotor is based on a PID algorithm that uses data taken from the inertial measurement unit (IMU) as input reference and compares them with an appropriately scaled and dimensioned radio input provided by the operator acting on the radio sticks.

The value processed by the PID controller is then normalized according to the specifications of the square wave (PWM) required as output and it is then saved in the pulse array to be written on the motors.

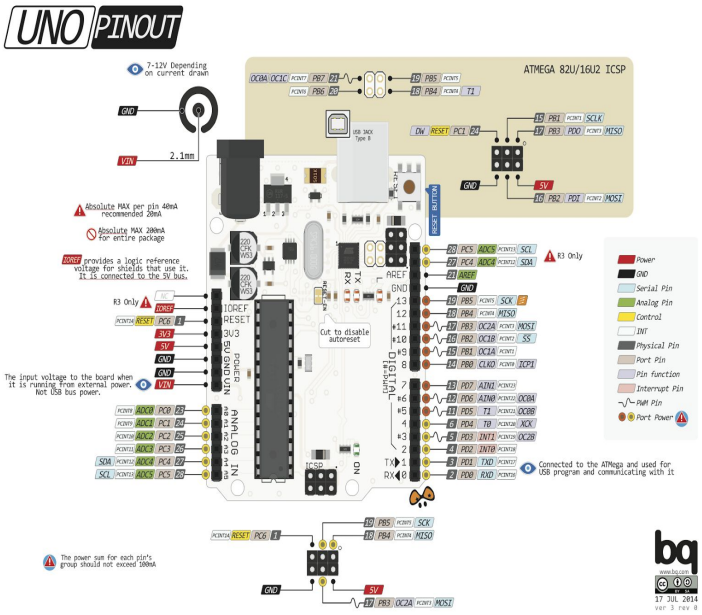
These values are then interpreted as the pulse width that represents the time in which the digital signal must be high with respect to the overall duration of the signal itself.

Radio AFHDS 2A

The radio receiver used is a 6-channel parallel radio receiver based on the AFHDS 2A protocol (Automatic Frequency Hopping Digital System).

Detecting PWM pulses using the pulseIn () function is simple but it has the very severe issue of busy wait until the end of the pulse, which is absolutely not possible in a flight controller.

Therefore, a lower level management is needed to increase efficiency and this mechanism is provided by the possibility of triggering some pins based on the change in the input value detected.



To do this it is first necessary to enable, as shown in the datasheet, the bit in the Pin Change Interrupt Control Register to enable scanning of the PCMSK2 mask and that bit is the one represented by PCIE2.

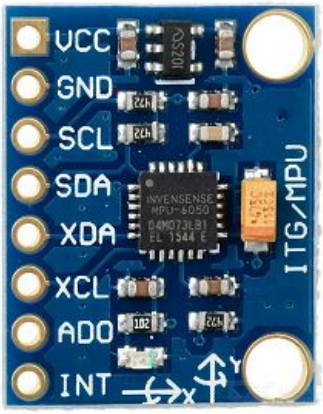
After this it's necessary within PCMSK2 to enable all the bits corresponding to the pins on which you want to have the pin change interrupt that are PCINT20, PCINT21, PCINT22, PCINT23 corresponding as can be seen from the figure to pins 4, 5, 6 e 7.

The resulting setup routine is:

```
void interrupt_registers_startup(){
   PCICR |= (1 << PCIE2);
   PCMSK2 |= (1 << PCINT20);
   PCMSK2 |= (1 << PCINT21);
   PCMSK2 |= (1 << PCINT22);
   PCMSK2 |= (1 << PCINT23);
}
```

IMU - MPU 6050

IMU is the short name for Inertial Measurement Unit and it is the hardware that allows, through the use of accelerometers, gyroscopes and magnetometers to measure angular velocities and accelerations along the three axes and therefore allows to know the position of the quadcopter on which it is installed whenever it is interrogated by transforming the raw data into angular values (both accelerations and angular velocities).



The most used IMUs are those belonging to the 6 series; MPU-6000 and MPU-6050.

The MPU-6000 interfaces with the MCU via SPI (Serial Peripheral Interface); it is faster than MPU-6050 and it is also more sensitive to external interferences.

For this reason it is usually found in the commercial flight controllers already built-in as an integrated circuit in the same board of the microcontroller.

MPU-6050 is used in this project for 3 main reasons:

1. It is undoubtedly the most widely used inertial sensor in Arduino
2. It costs very little and this allows to lower the total budget for the project and consequently allows a larger number of users to carry it out
3. It communicates via I2C (Inter Integrated Circuits) therefore it requires only the two SCL and SDA lines plus power supply and ground and it is also more resistant to external interference so for a DIY flight controller this is a great advantage

I2C serial protocol is greatly implemented and abstracted by the Arduino library <Wire.h> that allows to interface in a very intuitive way with devices that use this serial protocol and in the case of this project as usual with Arduino boards that only have one I2C interface we are going to use pin A4 and A5.

This type of sensor provides 16-bit data but is composed of 8-bit registers so, in order to obtain a single value, two contiguous registers must be read and combined with the left shift operator provided by the programming language (<<).

Furthermore, if sensor architecture is analyzed, it can be seen, as shown in the screenshot taken from the datasheet, that all the data registers are contiguous and this is due to the fact that the sensor is able to return all the raw data in one shot starting from a given address with an operation called register burst.

We will use burst reading because it minimize sensor accesses and allows us to be more efficient and therefore save CPU time which is essential with low-performance MCUs like the ATMega328p.

To be more precise we will skip the accelerometer and temperature data that are not involved in the rate/acro mode implementation.

InvenSense

MPU-6000/MPU-6050 Register Map and Descriptions

Document Number: RM-MPU-6000A-00
Revision: 4.2
Release Date: 08/19/2013

Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3B	59	ACCEL_XOUT_H	R	ACCEL_XOUT[15:8]							
3C	60	ACCEL_XOUT_L	R	ACCEL_XOUT[7:0]							
3D	61	ACCEL_YOUT_H	R	ACCEL_YOUT[15:8]							
3E	62	ACCEL_YOUT_L	R	ACCEL_YOUT[7:0]							
3F	63	ACCEL_ZOUT_H	R	ACCEL_ZOUT[15:8]							
40	64	ACCEL_ZOUT_L	R	ACCEL_ZOUT[7:0]							
41	65	TEMP_OUT_H	R	TEMP_OUT[15:8]							
42	66	TEMP_OUT_L	R	TEMP_OUT[7:0]							
43	67	GYRO_XOUT_H	R	GYRO_XOUT[15:8]							
44	68	GYRO_XOUT_L	R	GYRO_XOUT[7:0]							
45	69	GYRO_YOUT_H	R	GYRO_YOUT[15:8]							
46	70	GYRO_YOUT_L	R	GYRO_YOUT[7:0]							
47	71	GYRO_ZOUT_H	R	GYRO_ZOUT[15:8]							
48	72	GYRO_ZOUT_L	R	GYRO_ZOUT[7:0]							
49	73	EXT_SENS_DATA_00	R	EXT_SENS_DATA_00[7:0]							

A good habit when using sensors that need to return precise data is to provide and perform a calibration routine during setup; this is mandatory because all the sensors are intrinsically different and come out with small differences that must be smoothed out to make operations the same on all the platforms that will run the software.

What it's needed to be known about this sensor before proceeding with the code is provided by the datasheet and in particular since we are going to use only the gyro we are interested in relatively few parameters:

1. MPU-6050 I2C slave address 0x68
2. MPU-6050 wake-up register 0x6B
3. Gyro configuration register 0x1B
4. Hw low pass filter register 0x1A
5. First gyro data address 0x43

We start reading the sensor by writing on the wake-up register the value 0x00 which allows the unit to wake up and reset

```
Wire.begin();

Wire.beginTransmission(0x68);

Wire.write(0x6B);

Wire.write(0x00);

Wire.endTransmission(true);
```

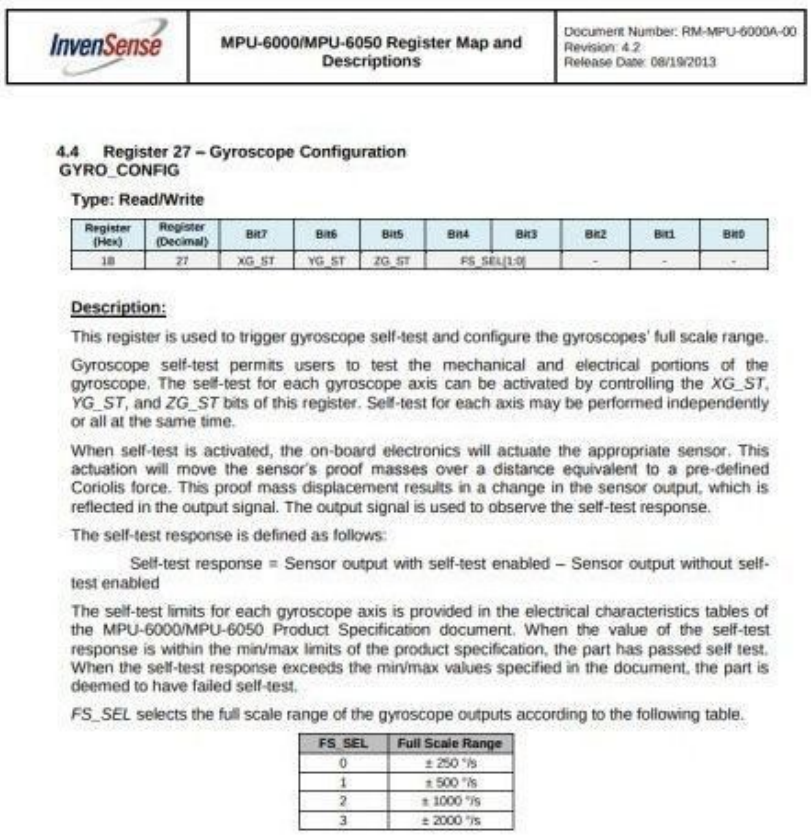
The lines above allow respectively to start the I2C communication, write the address of the register to be written, write the value to the register and close the communication.

Now we have to choose which will be the DPS scale that the gyro will use in the program; we use 500 dps (degrees per second) which is a good compromise between speed and accuracy, however the possible values are shown in the screenshot of the datasheet in the following page.

To do this we use the following lines:

```
Wire.beginTransmission(0x68);

Wire.write(0x1B);

Wire.write(0x08);

Wire.endTransmission();
```

We write the value 0x08 because as you can see in the figure on the right to get a resolution of 500 dps you need to write the value 1 in FS_SEL and if you consider the entire register 00001000 in hexadecimal is 0x08.



InvenSense | MPU-6000/MPU-6050 Register Map and Descriptions | Document Number: RM-MPU-6000A-00 Revision: 4.2 Release Date: 08/19/2013

4.4 Register 27 – Gyroscope Configuration
GYRO_CONFIG

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1B	27	XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]	-	-	-	-

Description:

This register is used to trigger gyroscope self-test and configure the gyroscopes' full scale range.

Gyroscope self-test permits users to test the mechanical and electrical portions of the gyroscope. The self-test for each gyroscope axis can be activated by controlling the *XG_ST*, *YG_ST*, and *ZG_ST* bits of this register. Self-test for each axis may be performed independently or all at the same time.

When self-test is activated, the on-board electronics will actuate the appropriate sensor. This actuation will move the sensor's proof masses over a distance equivalent to a pre-defined Coriolis force. This proof mass displacement results in a change in the sensor output, which is reflected in the output signal. The output signal is used to observe the self-test response.

The self-test response is defined as follows:

Self-test response = Sensor output with self-test enabled – Sensor output without self-test enabled

The self-test limits for each gyroscope axis is provided in the electrical characteristics tables of the MPU-6000/MPU-6050 Product Specification document. When the value of the self-test response is within the min/max limits of the product specification, the part has passed self test. When the self-test response exceeds the min/max values specified in the document, the part is deemed to have failed self-test.

FS_SEL selects the full scale range of the gyroscope outputs according to the following table.

FS_SEL	Full Scale Range
0	± 250 °/s
1	± 500 °/s
2	± 1000 °/s
3	± 2000 °/s

Reasoning with the same logic but with a different register you can set the low pass filter at 43Hz.

After the end of the registers setup it is possible to proceed with the calibration of the sensor.
The easiest way to do this is to take consecutive readings and calculate the average of the values found and subtract them as raw data error.

As you can see on the right when FS_SEL = 1 the divisor to get the value in deg / sec is 65.5 .

We now have all the elements to accurately retrive the angular velocities of the quadcopter among the 3 axes.

InvenSense | MPU-6000/MPU-6050 Product Specification | Document Number: PS-MPU-6000A-00 Revision: 3.4 Release Date: 08/19/2013

6 Electrical Characteristics

6.1 Gyroscope Specifications

VDD = 2.375V-3.46V, VLOGIC (MPU-6050 only) = 1.8V±5% or VDD, T_A = 25°C

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
GYROSCOPE SENSITIVITY						
Full-Scale Range	FS_SEL=0		±250		°/s	
	FS_SEL=1		±500		°/s	
	FS_SEL=2		±1000		°/s	
	FS_SEL=3		±2000		°/s	
Gyroscope ADC Word Length			16		bits	
Sensitivity Scale Factor	FS_SEL=0		131		LSB/(°/s)	
	FS_SEL=1		65.5		LSB/(°/s)	
	FS_SEL=2		32.8		LSB/(°/s)	
	FS_SEL=3		16.4		LSB/(°/s)	
Sensitivity Scale Factor Tolerance	25°C	-3		+3	%	
Sensitivity Scale Factor Variation Over Temperature			±2		%	
Nonlinearity	Best fit straight line; 25°C		0.2		%	
Cross-Axis Sensitivity			±2		%	
GYROSCOPE ZERO-RATE OUTPUT (ZRO)						
Initial ZRO Tolerance	25°C		±20		°/s	
ZRO Variation Over Temperature	-40°C to +85°C		±20		°/s	
Power-Supply Sensitivity (1-10Hz)	Sine wave, 100mVpp; VDD=2.5V		0.2		°/s	
Power-Supply Sensitivity (10 - 250Hz)	Sine wave, 100mVpp; VDD=2.5V		0.2		°/s	

Combining what has been explained in the previous pages, we get to write the following function that gives us exactly the data we need

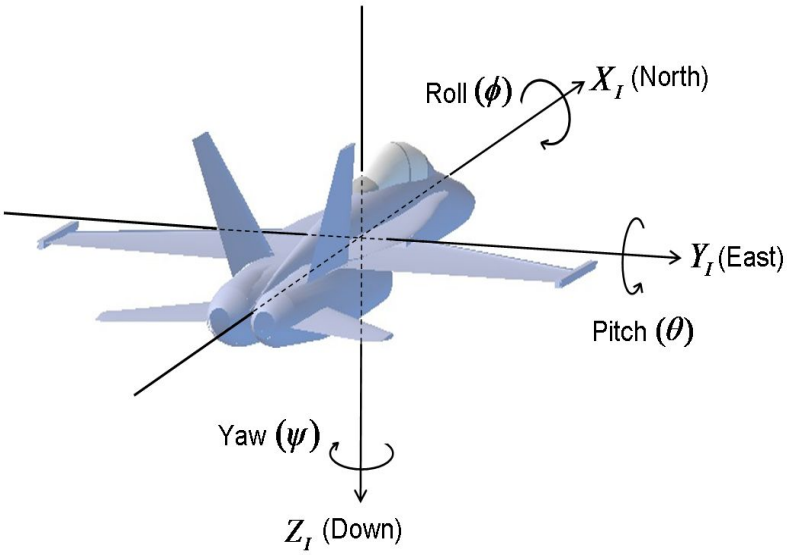
```
void read_imu(){

  Wire.beginTransmission(0x68);         //begin the I2C communication
  Wire.write(0x43);                     //sets the first gyro data as first address to start the burst request
  Wire.endTransmission(false);          //keeps alive the comunication
  Wire.requestFrom(0x68, 6, true);      // request of 6 registers in burst

  Gyr_rawX=Wire.read()<<8|Wire.read();  //left shift operator to recombine the 16-bit data sent in little-endian configuration
  Gyr_rawY=Wire.read()<<8|Wire.read();  //left shift operator to recombine the 16-bit data sent in little-endian configuration
  Gyr_rawZ=Wire.read()<<8|Wire.read();  //left shift operator to recombine the 16-bit data sent in little-endian configuration

  gyro_angle_x = (Gyr_rawX/65.5) - Gyro_raw_error_x;  //divide the 16-bit value by the raw data divisor and subtract the error obteined in calibration
  gyro_angle_y = (Gyr_rawY/65.5) - Gyro_raw_error_y;  //divide the 16-bit value by the raw data divisor and subtract the error obteined in calibration
  gyro_angle_z = (Gyr_rawZ/65.5) - Gyro_raw_error_z;  //divide the 16-bit value by the raw data divisor and subtract the error obteined in calibration

}
```

We now have both the gyro and the radiocontrol values available and therefore we are able to calculate the error between the actual quadcopter position and the desired one given by the radio stick commands.

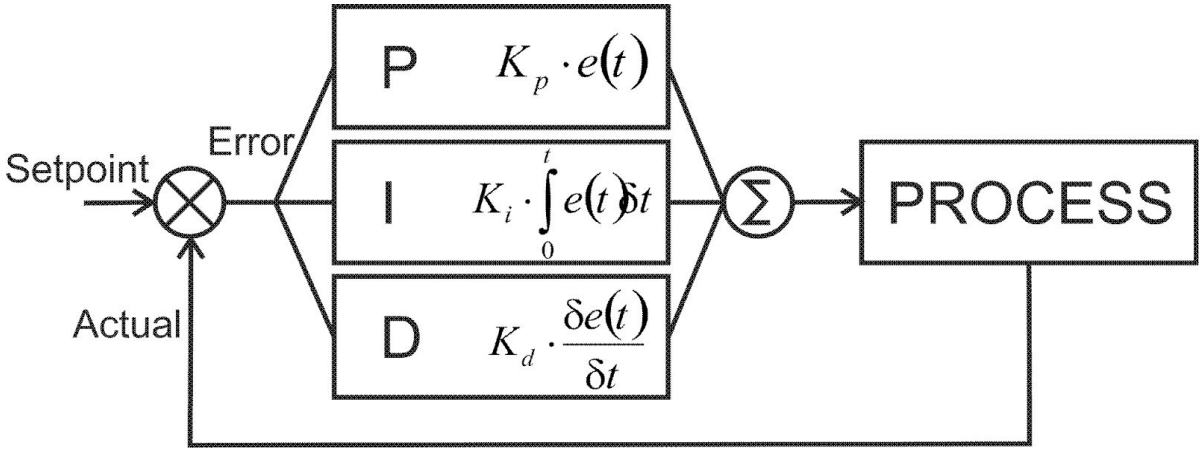
This error is used as the main parameter to be able to exploit, as will be explained in the next chapter, the PID controllers to stabilize the drone and make the flight experience precise with high performance (in acro mode).

Furthermore it is fundamental to establish a reference system for the dynamic system; in particular for the project we used the classical avionic reference system shown in the figure on the right.

Pitching upwards, rolling and yawing to the right yields a positive value from the gyroscope.



PID Controller



In control theory there are two main kinds of controllers:

- open loop controllers
- feedback loop controllers

Feedback systems differ from open loop systems because at each loop the system output is reported as input and is added / subtracted to the new reference passed as input as shown in the figure above.

The PID controller is one of the most used mechanisms in industrial applications to control a system in closed loop, and in particular it is a negative feedback loop controller.

The positive feedback is such defined because its operation leads the system to amplify in a positive way, thus adding a corrective value to the state of the system itself. This mechanism is unstable because often using this approach the system tends to diverge with respect to the target value and this is mathematically provable.

On the other side, negative feedback leads the system to make a subtractive correction, so the value generated at each iteration tends to damp the system's next state in case of error and this generally leads the system to converge towards the target.

The P component, or proportional (*Kp*), is only a multiplicative constant with respect to the calculated error that makes the quadcopter to turn in the direction of the zero error.
It is sometimes called Gain, and it is the power with which compensation will take place. The higher it is, the more the flight controller will send inputs to the motors to try to correct the quadcopter position.
To have a very stable quadcopter it is important that it is set as high as possible.
If it is too high it can cause persistent oscillation and in extreme cases these oscillations can lead the drone to be completely uncontrollable.

The I component, or integral constant (*Ki*), acts over the persistence with which the variations are compensated. The I parameter determines how long the quadcopter will tend to maintain the desired inclination. The higher it is, the more the quadcopter will tend to remain in the position imparted by the pilot.
It must be modified by taking small steps, generally going up 0.02 units at a time.
It is necessary to ensure that its modification does not induce the quadcopter to low frequency oscillations. The higher it is, the more any corrections in case of external disturbance will be maintained. An external disturbance can be, for example, a gust of wind.
It is strongly recommended to set this parameter after both *Kp* and *Kd*.

The D component, or derivative constant (*Kd*), influences the speed of the compensation. Its behaviour is particularly highlighted in passages from fast flight to hovering and during rapid inflight-setpoint changes.
In several cases it is increased or decreased proportionally to the parameter P.
The setting of this parameter is recommended only after a good parameter for the multiplicative constant *kp* had been found.

This project, having an object able to move independently among the three axes X, Y and Z needs 3 controllers like the one explained above, capable of handling and correcting the error for each component of the three-dimensional space at each loop.

For each PID loop an error is calculated as the difference of the sampled value from the gyro, the one calculated in the function at the very end of the previous chapter, and the setpoint given by the mapping of the radio signal in deg / sec (the same measurement unit of the gyro).
After the calculation of the 3 components the output of the single PID controller is given by

total_pid = *Kp * error + ki + ki * error + kp * (error - previous_error)*

This calculation must be done in the same way for the 3 controllers of the three axes and must be normalized with respect to the smallest value or the largest value that the PWM pulse can assume (in this project the PWM values are all the integer values between 1100 and 2000).

Before calculating the final pulse for each motor we need to save the throttle value in a variable; in such way we do not have to disable interrupts. Than we have to map that value within an interval [1000, x], where x must be chosen to allow correction of the PID controller when the throttle (channel 3) is in its highest position.

Finally, the pulse for each motor is given by the sum of the normalized throttle value with the appropriate combination of the values of the three previously calculated PID controllers as reported below

```
speedVec[MOTOR_1] = throttle + total_pid_y + total_pid_x +  total_pid_z;
speedVec[MOTOR_2] = throttle + total_pid_y  - total_pid_x  -  total_pid_z;
speedVec[MOTOR_3] = throttle -  total_pid_y + total_pid_x  -  total_pid_z;
speedVec[MOTOR_4] = throttle -  total_pid_y  -  total_pid_x + total_pid_z;
```

Note that the sum of 4 normalized values with respect to the same interval is not a normalized value again with respect to the same interval, therefore after calculating the values of the motorPulse array they must be again normalized before passing the array to the real motors driving function.

The last computation performed by the calculate_pid() function, which is very important and cannot be absolutely forgotten, is to save the 3 values of the errors calculated at the beginning of the function as previous_error in order to be used in the next pid loop for the calculation of the derivative component.

ESC - Electronic Speed Controller

The ESC or Electronic Speed Controller is the component that takes care of converting a PWM pulse that arrives as input from the flight controller into the corresponding voltage that will be used by the motor to keep itself in motion, throttle-up or slow down.

The signal that is transferred from this to the motors is a three-phase signal in which each phase is 120° out of phase ($\phi = 120°$) with respect to the others and this is needed to ensure the correct motor rotation direction.

In recent years, numerous protocols have been developed and this is because these devices no longer interface in the traditional way directly with the radio receiver but the signals are processed and supplied by the flight controller.

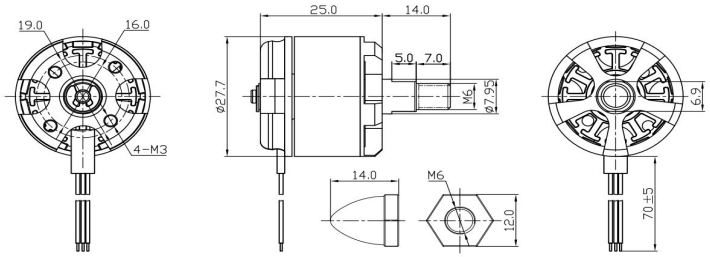
In fact, before the advent of drones, brushless motors were controlled by these devices like normal servos with [1000;2000] microseconds pulses at 50 Hz.

Today there are some brands that provide configuration suites in order to provide the user a choice of the ESC protocol with respect to the flight controller computational power; in this moment while I'm writing the best ESC available protocol is Proshot followed by DShot1200 and DShot600 (all of them are Digital protocols).

In our project to keep the budged as low as possible but also due to the poor performance of the Arduino UNO we use a refresh rate of 4ms (250 Hz frequency) based on classic PWM pulses of [1000; 2000] microseconds that it's more than enough for a good fly experience.

Brushless motors

MOTOR OUTLINE DRAWING



MOTOR PERFORMANCE DATA

MODEL	KV (rpm/V)	Voltage (V)	Prop	Load Current (A)	Pull (g)	Power (W)	Efficiency (g/W)	Lipo Cell	Weight (g) Approx
B2212	920	11.1	8045	7.3	465	81	5.7	2-4S	50
			1045	9.5	642	105	6.1		
	980	11.1	8045	8.1	535	90	5.9		
			1045	10.6	710	118	6.0		

Brushless motors are undoubtedly the most used motors in most aircraft model projects because they have the advantage of wasting less energy (better efficiency in power consumption), delivering more power and having very limited maintenance compared to brushed motors present in most of low cost or non professional quadcopters.

Another important feature is the speed and accuracy of response to commands due to the fact that this type of motor has no internal friction and therefore any variation can be performed in a much more reactive and precise way without dispersing most of the energy in heat at high RPM.

The motors chosen for the project, whose characteristics are shown at the top of the page, have a relatively low KV value and this is because since we have a quite big frame for a racer we can equip it with bigger propellers.

Choosing low KV motors is a good choice also due to the fact that the ESCs we use allow the use of batteries up to 3S (3 cells in series) while the more powerful motors use small propellers and make many more RPMs; but to do this they usually require minimum 3 / 4S batteries and have lower efficiency.

From the characteristics we can see that when powered with a 3S LiPo battery and using 8045 propellers, where 80 equals 8 inches and 45 is the pitch of the blades, in the full throttle position is able to generate a thrust equal to 465 g.
Multiplying this value for the 4 motors, we obtain 465 x 4 = 1860 g which is the maximum thrust that the quadcopter is able to deliver at full throttle.

Notice now that a good practical rule is to ensure that the weight of the quadcopter is no more than half of the maximum thrust that the motors are able to deliver in order to have a reactive aircraft.

Another parameter to consider and to keep under strict observation is the Load Current which, as can be seen in the case of our 920 KV, is 7.3 A which multiplied by 4 results in 7.3 x 4 = 29.2 A

This last calculation gives us two fundamental information on two further constraints to be included in the model:

1. The load current of the single motor at full speed is 7.3 A therefore we deduce that each single ESC must be at least 10 A but personally I advise to oversize the ESCs with respect to this parameter to reduce stress and avoid excessive overheating; anyway 10 A surely will work.

2. The lithium polymer battery (LiPo) with which we are going to feed our system must be able to supply the power simultaneously to 4 motors at full throttle; it means that as calculated before it must be able to provide 29.2 A and this parameter can be easily calculated as the capacity of the battery expressed in Ah multiplied by its C value that can be found written on the battery or is provided by the producer.
   If two C values are given take the lowest one because the other represents the burst value that is the peak value which can be delivered for a little time interval (usually 10 seconds).

However, the best performing racer drones reach ratios *weight* : *max_thrust* even in the order of 1 : 8 but they have very big values of Load Current for each motor which translates into high power consumptions and consequently less autonomy.

This should guide you to the correct sizing of the battery which is the last but not the least important choice that has to be done.

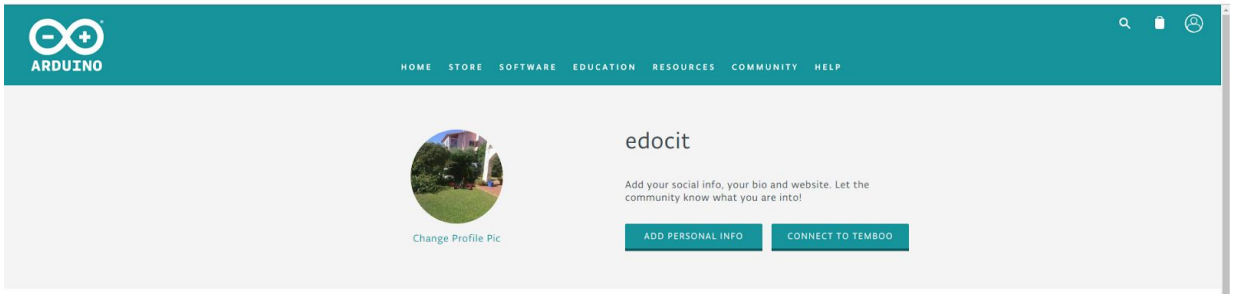
A final piece of advice I would like to give is to evaluate the purchase of graphene LiPo because the weight is no longer so much greater compared to normal LiPos and the higher price is totally compensated by the in flight performance and by the sensibly reduced charging times.

Conclusions

To conclude I would like to thank Prof. Giorgio Buttazzo for allowing me to develop this project.

I would also like to thank all the Arduino User Group Cagliari members, of which I am also a member, who have taught me a lot in the last two years about electronics.

If you liked my project please check http://www.augc.it/ and for any question I'm available in the Arduino forum



You can also find quadcopter videos on my YouTube channel edoardo cittadini where you will also find updates or new projects in the future.