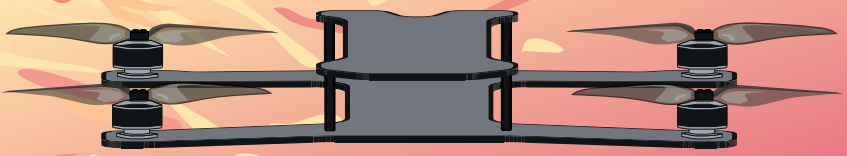
Carbon aeronautics



quadcopter
build and programming manual

learn
aeronautics
programming
electronics

100 %
hackable

< 250 g
weight

10 min
flight time

Carbon aeronautics

you are about to embark on an exciting journey... building and programming a quadcopter from scratch.

this manual will guide you every step of the way, explaining the essentials on aeronautics, electronics and embedded programming.

all components and programs are fully hackable, meaning that you can adapt anything you want and create a quadcopter capable of stuff that goes way beyond the scope of this manual!

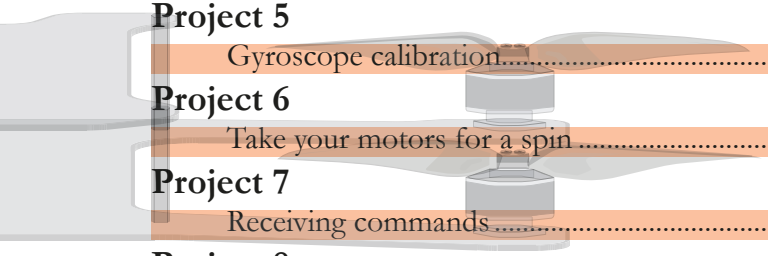
good luck and most importantly: have fun!

Carbon Aeronautics quadcopter build and programming manual

Project, text and figures by Laurens Raes

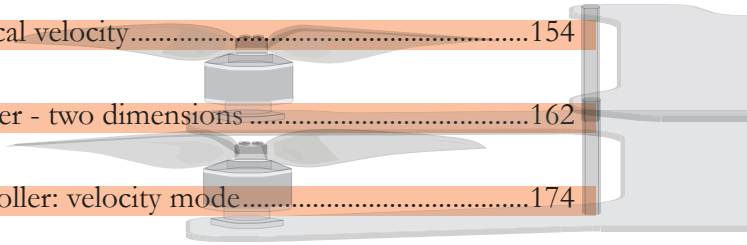
First edition, August 2022.

Contents



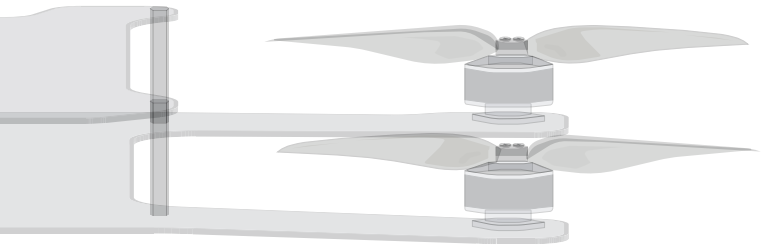
Part II: stabilization mode

Part III: velocity mode



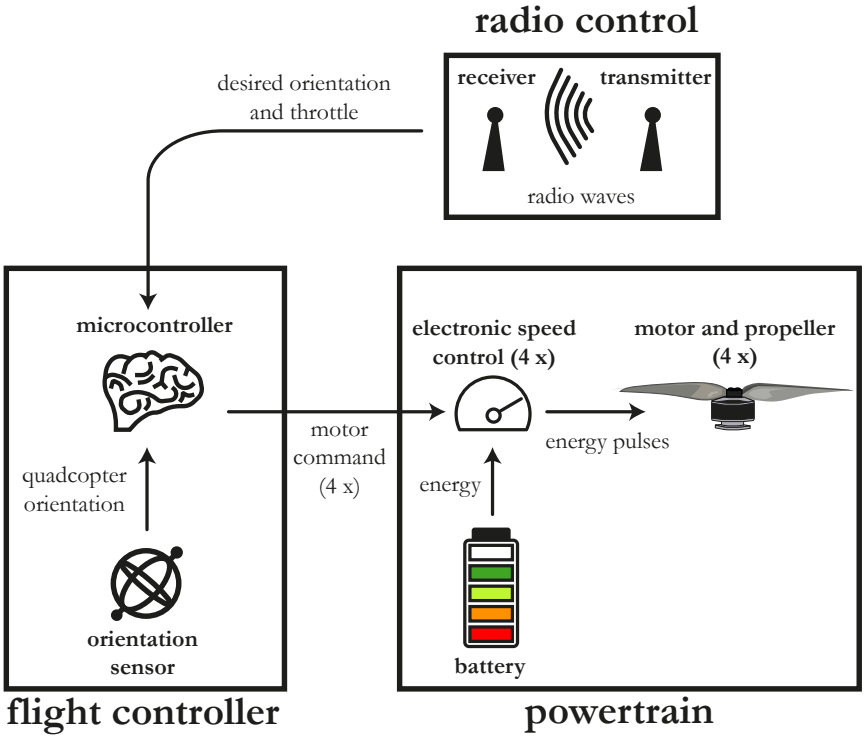
Part IV: quadcopter design and simulation

Part V: expanding your horizon



Project 1

Concept, parts and programming



Explore the basics of your quadcopter

Let's start your exciting journey in the world of aeronautics, electronics and programming with the concept behind the flying machine that you will build and the parts that you need. This manual will help you tackle the basics and enable you to build your own quadcopter; a drone with four motors.

The creation of flying machines is a true engineering challenge and involves solving several problems, from aerodynamics to power systems. In the case of a quadcopter, you rely on four motors and propellers to provide enough thrust to start flying. Obviously, these are not the only necessary components. The figure to the left displays the basic overview of a quadcopter with three major active building blocks:

- The **radio control system**, which consists of a radiotransmitter and a receiver. The position of the sticks on the radiotransmitter are transformed into commands and subsequently sent to the receiver that is situated on your quadcopter.

- The **flight control system**, which consist of a microcontroller and some sensors. The bare minimum you need to stabilize the quadcopter is an orientation sensor, but you can add various other sensors (barometer, GPS, ultrasonic,...) to make your flight easier. The information of your sensor and the commands from your radiotransmitter are then processed in the microcontroller, which is the brain of your quadcopter. The microcontroller calculates the optimal speed of each of the four motors to keep the quadcopter in the air.

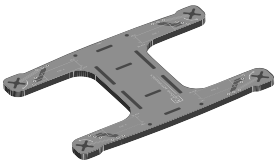
- The third building block is the **powertrain**, which is the high current part of the quadcopter. The battery is the power source of the whole system and sends energy in the form of electrical current to four electronic speed controllers (ESCs); an ESCs converts the provided current into current pulses, with a pulse length proportional to the motor command sent from the microcontroller. This gives a motor speed proportional to the motor command and in turn, a certain thrust allowing you to take off!

And basically, that's all there is to it! With the general idea behind your quadcopter clearly understood, let's have a look at all different physical parts that you will use. Your quadcopter consists of three active building blocks; a radio control system, flight controller and powertrain. Moreover, you also need a frame on which you can mount all these active components. Some auxiliary parts are also necessary, to charge the battery and test your microcontroller, sensors and powertrain before fixing them to the frame.

All necessary components are listed below and divided into parts for the frame, flight controller, powertrain, battery and radio control. Each part is available on the consumer market, so if you break a part during flight or you want to change parts, you can easily buy it yourself. This manual is designed to guide you building your own quadcopter while enabling you to change any aspect of it as well.

1 frame

1a **lower quadcopter frame**
CarbonAeronautics



1x

1b **upper quadcopter frame**
CarbonAeronautics

1x

1c **frame spacers**
M3 x 30 mm

4x

1d **spacer fastening screws**
M3 x 6 mm

12x

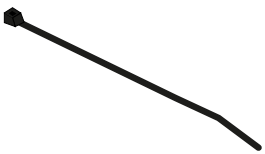
1e **battery strap**
210 mm

1x

1f **landing pad**

4x + 1 reserve

1g **cable ties**
16 mm



6x + 4 reserve

1h **standoff spacer**
M3 x 20 mm

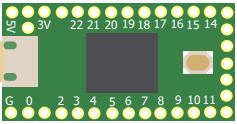
4x + 2 reserve

1i **cable protector**
500 mm

1x

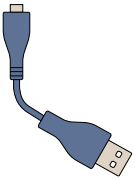
2 flight controller

2a microcontroller
Teensy 4.0



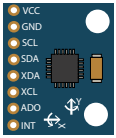
1x

2b microcontroller connector
USB A to micro B



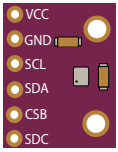
1x

2c orientation sensor
GY-521 MPU-6050



1x

2d barometer
GY-BMP280



1x

2e sensor fastening screws
M3 x 20mm

4x

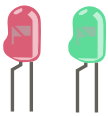
2f sensor locknuts
M3

4x

2g sensor full nuts
M3

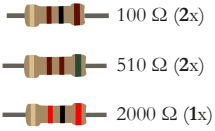
12x

2h green and red LED

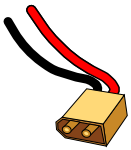


1x + **1**x

2i resistors

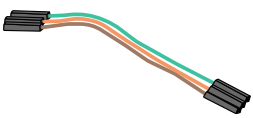


2j **battery connector**
XT60



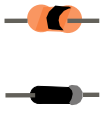
1x

2k **jumper wires**
female to female 10 cm



3x + 3 spares

2l **diodes**

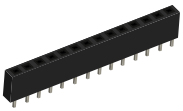


Zener diode (**1**x)
(BZX79C2V4)

Diode (**1**x)
(1N4007)

1x

2m **female headers**
40 pins - 2,54 mm

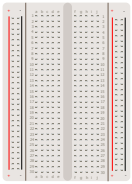


2x

2n **male headers**
40 pins - 2,54 mm - right angle

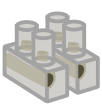
2x

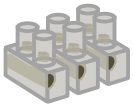
2o **breadboard**
400 points



1x

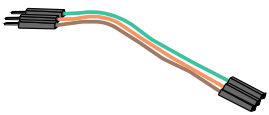
2p **wire terminal strip**





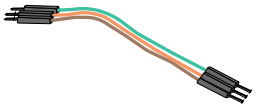
1x **1**x

2q **jumper wires**
male to female 10 cm



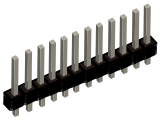
3x + 3 spares

2r **jumper wires**
male to male 10 cm



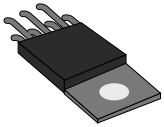
40x

2s **male headers**
40 pins - 2,54 mm - straight



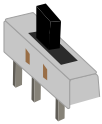
2x

2t **power switch**
BTS50080-1TMB



1x

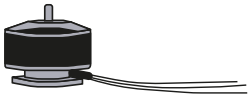
2u **slide switch**
OS102011MS2QN1C



1x

3 powertrain

3a motors
GEPRC GR1105
5000 kV

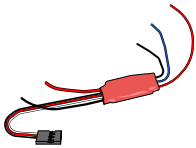


4x

3b motor fastening screws
M2 x 4 mm

16x

3c Electronic speed controllers
HobbyKing 6A
ESC with BEC



4x

3d clockwise propellers
Gemfan 3018R

2x + 2 reserve

3e counter-clockwise propellers
Gemfan 3018

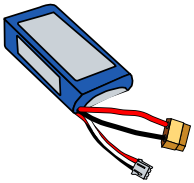
2x + 2 reserve

3f propeller fastening screws
M2 x 8 mm

4x + 8 reserve

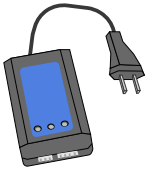
4 Battery

4a Batteries
Turnigy 2S 1300 mAh



2x

4b battery charger
Hobbyking B3AC



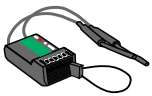
1x

5 Radio control

5a **Radiotransmitter**
Flysky FS-i6

1x

4b **receiver and bind plug**
Flysky FS-iA6B



1x

Alternative parts

You are not limited to the parts that are described in this paragraph and chances are you want to choose different components for various reasons such as higher thrust, longer flight time or lower weight. The parts that can easily be swapped are the propellers, ESC (Electronic Speed Controller), motor and battery. To give you an idea of the possibilities, this paragraph describes some successfully tested variations on the basic quadcopter. The PID values derived later on in this manual are a good match for all variations, but remember that the weight of your quadcopter will be affected: the basic quadcopter weighs 247 gram while the combination of all the heaviest components described in this paragraph weighs 278 gram.

The **battery** is perhaps the easiest interchangeable component. The base part is a 2S battery with a capacity of 1300 mAh and a weight of 70 gram. Other tested possible batteries include a 2S battery with 1000 mAh (weight: 60 gram) or a 2S battery with a 1500 mAh capacity (weight: 80 gram). Additional capacity comes at a cost in the form of extra weight and thus a less flexible quadcopter.

The choice of your **ESC** and **motor** combination needs some more care. You should make sure that the maximal load current of both your ESC and motor are similar: the part with the smallest load current limits the load current of both components. Since a motor or ESC with a higher load current generally weighs more, the optimal combination consists of motors and ESC with similar load current. The base motor and ESC combination (GEPRC GR1105 5000 kV and Hobbyking 6A ESC with BEC) both have a load current of around 6 A. Another tested possibility is the combination of the GEPRC GR1206 4500 kV and Hobbyking 12A ESC, both having a load current of around 12 A. The 6 A combination of four motors and ESCs weighs around 50 gram, while the 12 A combination weighs 66 gram.

Another important value is the motor kV rating: this determines how fast the propeller can turn at full throttle: a 5000 kV motor turns at 5000 rpm/V. Since a 2S battery has a nominal voltage of 7.4 V, this equals to a nominal rmp of 5000 rpm/V x 7.4 V = 37 000 rpm. To lift a quadcopter with a weight between 200 and 300 gram, you need a motor with a kV rating between 4000 and 6000, depending on the propeller.

Once you have chosen your ESC and motor combination, the **propellers** are next. A larger propeller generates more thrust: this is great because it means your quadcopter can weigh more and can successfully combat stronger wind gusts. However, the necessary load current increases and your motor and ESC have to withstand this higher current, otherwise they will overheat and possibly start burning. Larger propellers obviously weigh more as well.

Gemfan propeller (weight in gram for 4 props)	**3018** (2 blade) 3.4 g	**3035** (3 blade) 5.6 g	**4024** (2 blade) 6.4 g	**4019** (3 blade) 8.4 g
GEPRC GR1105 5000 kV + **Hobbyking 6A ESC** with BEC	**5 A**	**7 A**	**9 A**	**12 A**
GEPRC GR1206 4500 kV + **Hobbyking 12A ESC** with BEC	**5 A**	**7 A**	**9 A**	**12 A**

Four different propellers are tested with the two motor/ESC combinations and tabulated above together with their current at full throttle with a full 2S battery. The colour code can be explained as:

- **Green**: the motor and ESC can withstand full throttle with this propeller for longer periods of time - this combination is suitable for beginners.
- **Orange**: the motor and ESC can only withstand short bursts of full throttle with this propeller - this combination is only suited for experienced flyers given the risk of motor or ESC overheating.
- **Red**: this combination is not recommended given the high risk of motor and ESC overheating.

For reference, the **maximal dimensions** for the battery and the propellers given the frame width are included here as well:

- 10.2 cm is the maximal diameter of the propeller (corresponds with a 4 inch propeller).
- The battery bay has a 12 cm x 4 cm x 3 cm dimension, but you need to leave some space for the receiver, electronic cables, protectors and screws. This limits the practical space to 8.5 cm x 3.3 cm x 1.5 cm.

Additional required tools and material

To complete your build, you also require some additional tools and material. Except for a computer, these are only necessary when starting the actual build, not when testing the components in the first projects (except if you still need to solder headers to your Teensy and sensors in order to test them on the breadboard).

- a **soldering iron** or station, to solder the motors wires, ESCs, resistors, LEDs and male/ female headers to each other / the printed circuit frame on you quad-copter frame.
- sufficient **solder material**.
- a **soldering helping hand** to clamp the parts you are soldering together.
- a **wire stripper** to strip the electrical insulation from the ESC and motor wires.
- a **wire cutter** to cut the ESC and motor wires.
- a **computer** capable of running Arduino (see arduino.cc/en/software)
- two **hex keys** (1.5 mm and 2 mm)
- a **multimeter** to check for short-circuits or bad connections.

General safety instructions

Battery

- Read the battery and battery charger manuals carefully before use.
- Never charge the batteries unattended.
- Before connecting the battery and your motor(s) or quadcopter for the first time, make sure there are no short-circuits between your soldered components using your multimeter.

Electronics

- Before connecting the electronics to a power source (such as your computer), make sure that there are no short-circuits between your soldered components using your multimeter.
- Remove the propellers and do not touch the motors unless you are sure that your program is working properly to avoid losing control over your quadcopter.
- Never run your motors without propellers.

Before flying

- Make sure the failsafe and safety-related code lines are implemented and working correctly.
- Check the regulations that are applicable in your country (with regard to maximal altitude, speed, weight,...) when flying your unmanned quadcopter outdoors.

Setup your microcontroller for programming

The core of your quadcopter project is the Teensy microcontroller that you will program in such a way that it becomes the flight controller and thus brains of your project. The Arduino software will be used to program the microcontroller, together with Teensyduino.

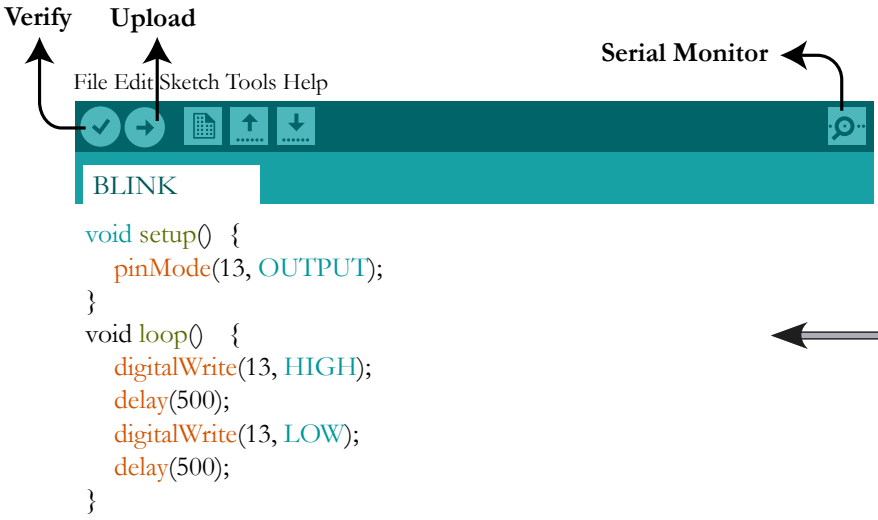
You can find all information with regard to the installation of the necessary software on the website of the Teensy manufacturer: www.pjrc.com/teensy. The installation steps will be described here as well, but please refer to the pjrc and arduino websites if you need additional troubleshooting.

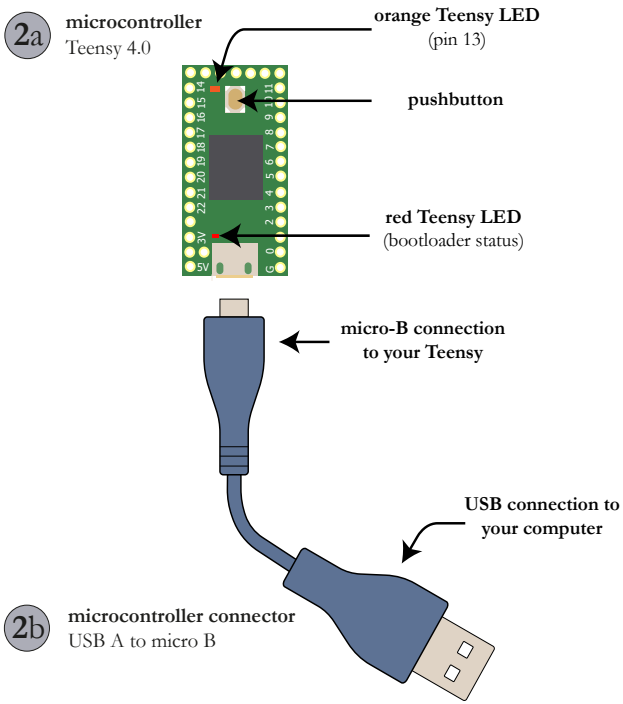
1. Connect your new Teensy to your computer using the USB cable (see figure to the right).

2. Your Teensy should come with the LED blink program pre-loaded; this means that the orange LED on your Teensy should blink slowly after connection with your computer.

3. Press and release the tiny pushbutton on the Teensy. The orange blinking LED should stop and the red Teensy LED should be visible. This means your Teensy works correctly.

4. Disconnect your Teensy from your computer by disconnecting the USB cable.





5. Download and install the Teensy Loader program, which communicates with your Teensy board. Guidance on the installation process can be found at pjrc.com/teensy/loader.html. Click on the operating system of your computer, read the information and click on the Teensy Loader link to start downloading.

6. If you do not have the Arduino software (IDE) yet, download the latest version from arduino.cc/download and install it on your computer. Guidance on the installation process can be found at arduino.cc/en/Guide/Windows or arduino.cc/en/Guide/MacOSX or arduino.cc/en/Guide/Linux.

7. The final piece of software to install is Teensyduino, the software add-on for Arduino. Download it by going to pjrc.com/teensy/td_download.html and follow the instructions on this webpage.

8. Open the Arduino IDE; a new empty sketch should load automatically. Copy the code in the figure to the left of this page and save the file under the name BLINK. Now click on 'Verify'. You will first have to save your sketch. After verification, you should view the message 'Done Compiling' below on your screen. If you get an error, verify whether you copied the code correctly.

9. Before you can upload your verified code to your Teensy, you need to setup your Teensy in the Arduino IDE. Go to tools and:

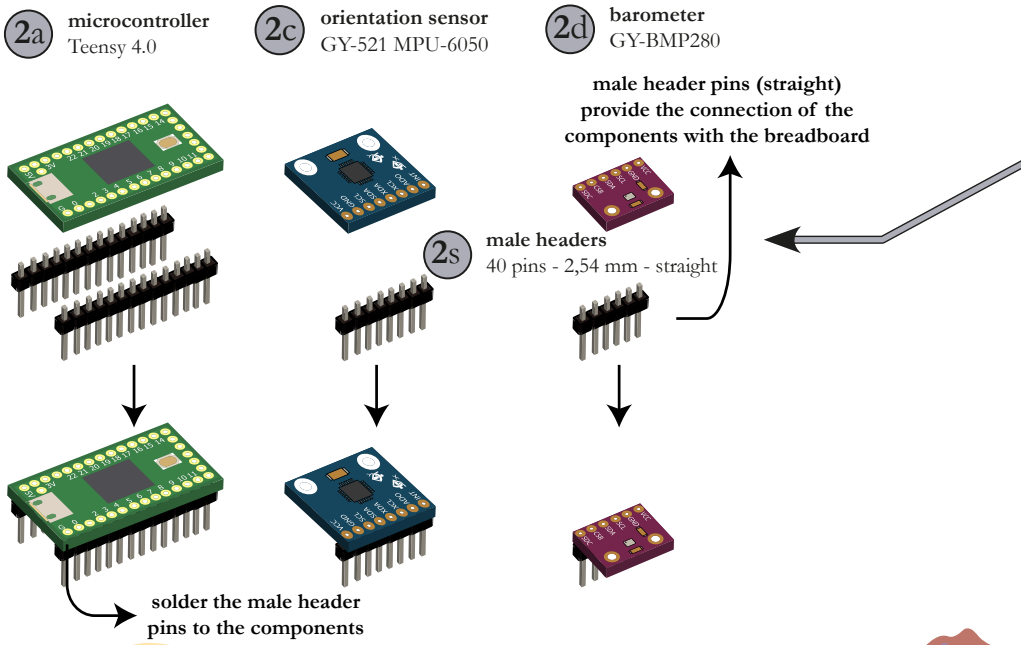
- Click on 'Boards' and 'Teensyduino' and select the Teensy 4.0 board.
- Verify that the USB type is 'Serial'.
- Verify that the CPU speed is 600 MHz.
- Connect your Teensy again with your computer using the USB cable. Under Port, a USB port should be displayed. Click on it.

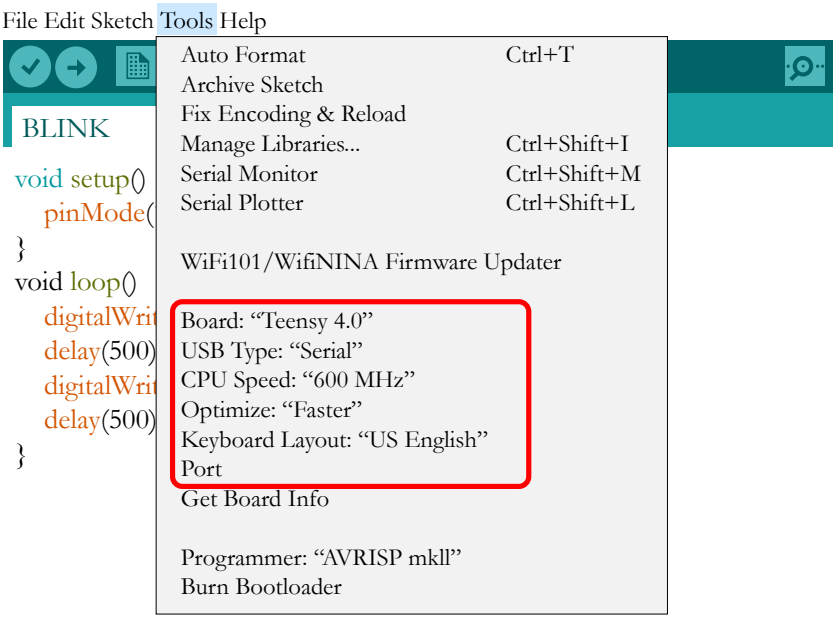
10. Press the upload button on the screen. The internal Teensy LED should start blinking again. Change the blinking speed by changing the delay time of 500 (milliseconds) in the code to for example 100 (milliseconds) to blink faster, or 1000 (milliseconds) to blink slower. Adapt and upload the code to verify that you are truly in control of the Teensy. When this test is successful, you are ready for the next project!

Code compatibility

The code throughout this book is compatible with the following Arduino (library) versions:

- Arduino IDE: 1.8.16
- Teensyduino: 1.55
- BasicLinearAlgebra library: 3.2.0 (only necessary for part III)



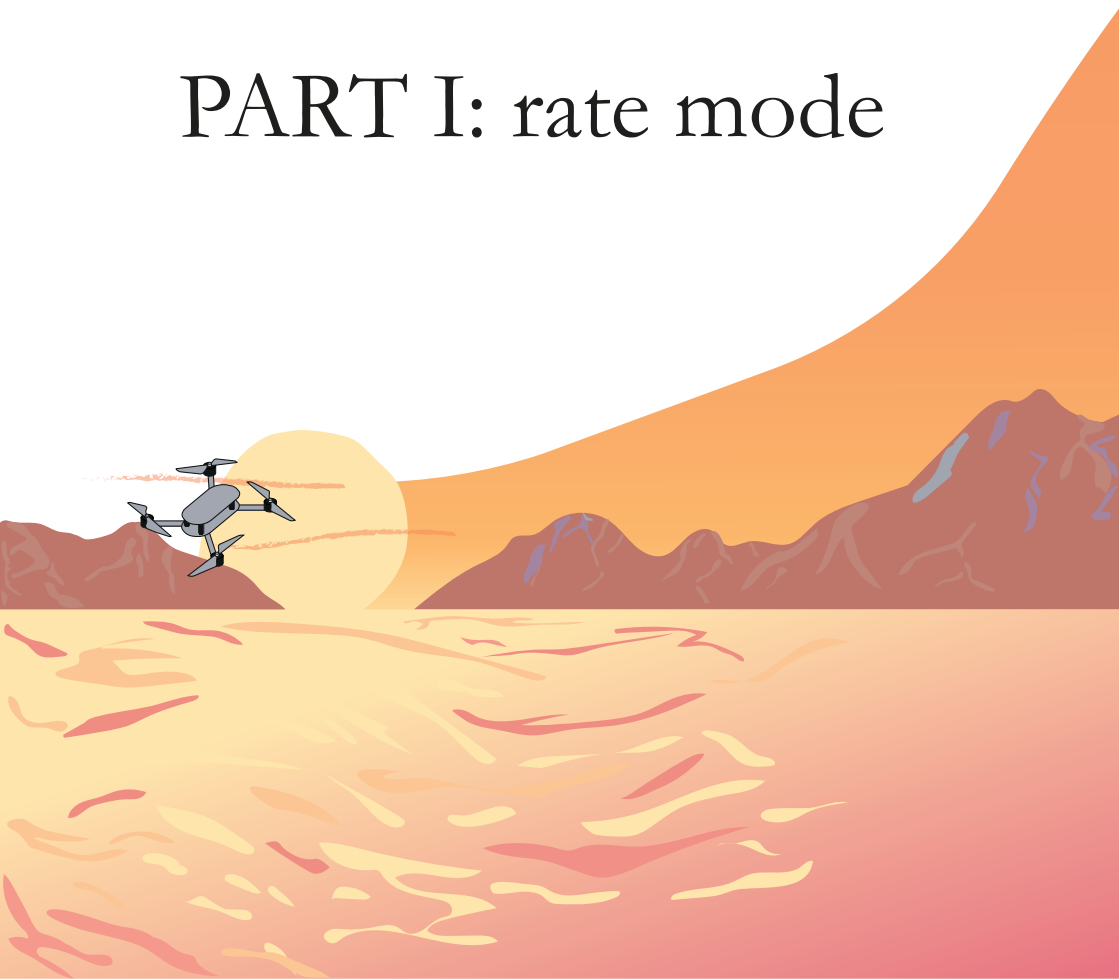


Solder pins to your microcontroller and sensors

You will use a breadboard to separately test the electronic components of your flight controller. To be able to electrically connect the components with the breadboard, you need to use straight male header pins that are soldered to your Teensy microcontroller, the MPU-6050 gyroscope and the BMP-280 pressure sensor. If these parts do not come pre-soldered with header pins, you will need to solder them yourself.

For easy soldering, you can insert the pins in your breadboard and put the component on top such that the pins are soldered straight to the microcontroller and sensors. If you have never soldered before, you can consult the internet for some tutorials.

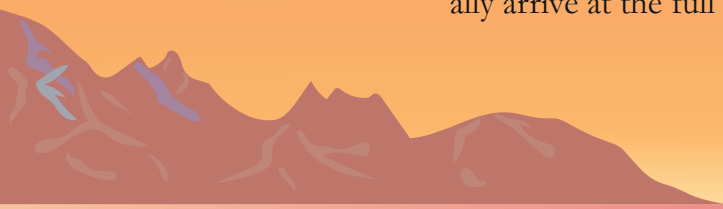
PART I: rate mode



in the first part, you will build your quadcopter and program a flight controller that enables you to fly in rate mode; this is the easiest-to-implement controller that gives you full control over the performance of your quadcopter.

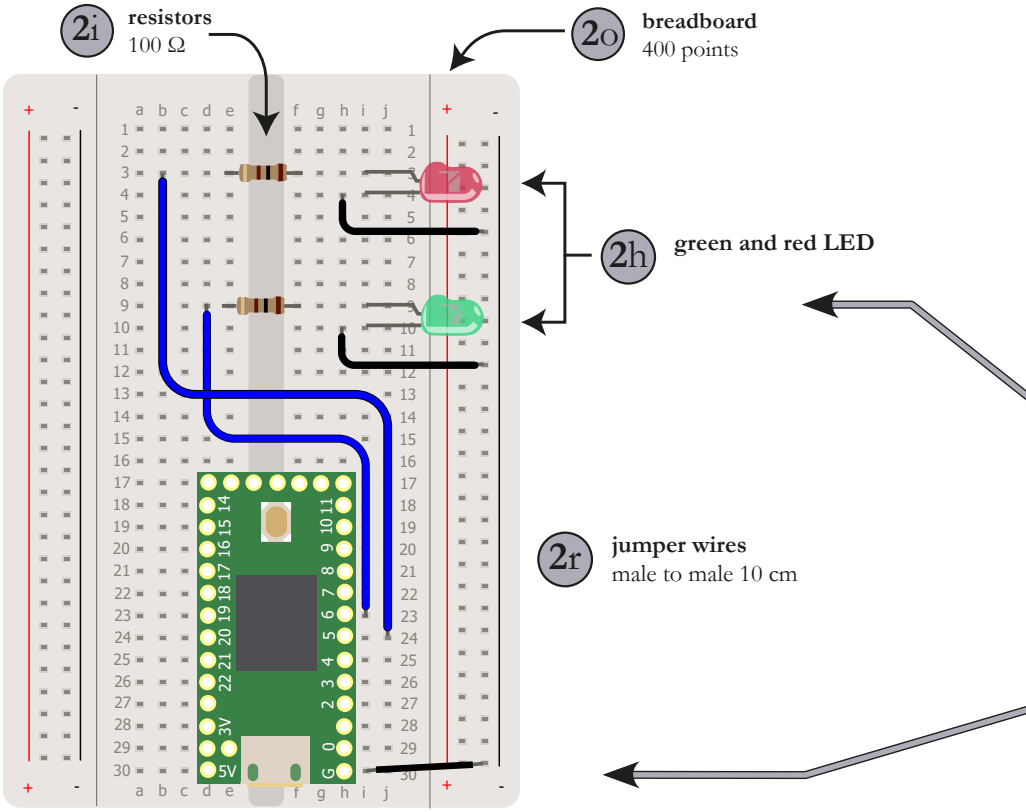
complex projects such as this one are often cut in smaller, independent pieces that are tested separately, before all components are put together.

you will follow this approach and start with simple building blocks and code, to eventually arrive at the full build and flight code.



Project 2

LED control



Use LEDs to receive feedback

Throughout this manual, you will learn how to communicate with your quadcopter by giving it commands. However, this communication goes only one-way from your radiotransmitter to the quadcopter. Sometimes it is useful to receive some feedback from the quadcopter, for example when the setup and calibration process is finished or when the battery voltage becomes low. To do this in an easy way without telemetry, you will use three signal LEDs.

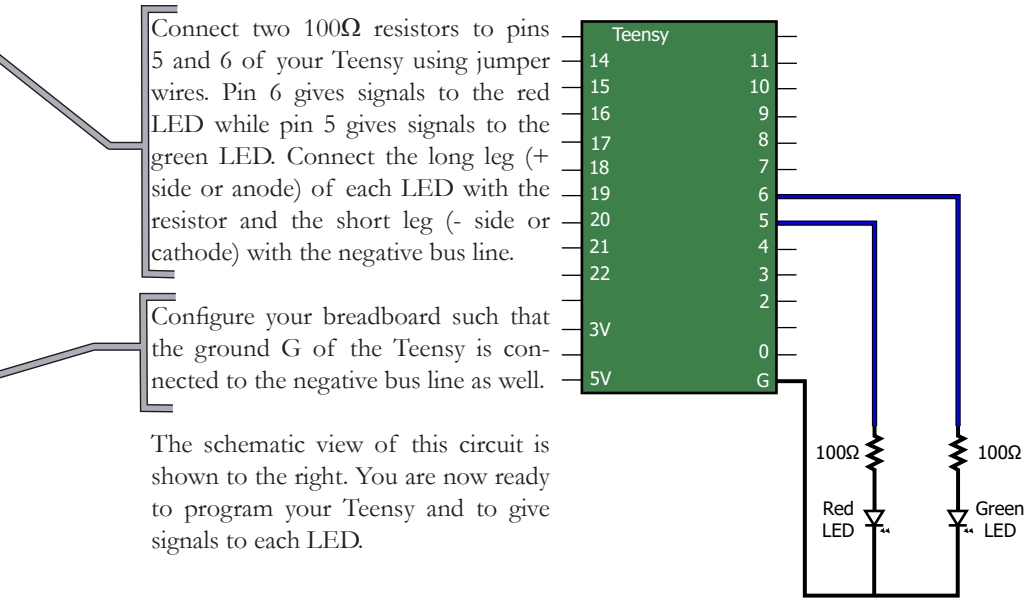
The first led you will use is the internal led of the Teensy, which you already experimented with. This orange led is controllable through pin 13 and requires no additional circuit building. Lighting this led can be useful to show that the microcontroller receives power and is working correctly.

You will also use two additional external LEDs to signal the start and end of the setup program. Before you are able to fly, the microcontroller will have to start the auxiliary sensors and calibrate them. This takes about four seconds during which the quadcopter is not yet able to start. During this time, you turn on the red LED to signal that the quadcopter is still in the setup process. When the setup process is successfully finished, you turn off the red LED and turn on the green LED. Let's start to build the electronic circuit necessary to light these external LEDs.

Connect two 100Ω resistors to pins 5 and 6 of your Teensy using jumper wires. Pin 6 gives signals to the red LED while pin 5 gives signals to the green LED. Connect the long leg (+ side or anode) of each LED with the resistor and the short leg (- side or cathode) with the negative bus line.

Configure your breadboard such that the ground G of the Teensy is connected to the negative bus line as well.

The schematic view of this circuit is shown to the right. You are now ready to program your Teensy and to give signals to each LED.



Coding

All arduino sketches consist of both a setup and a loop part. The code in the setup part of the sketch only runs once, during startup of the microcontroller. The code in the loop part of the sketch runs continuously when the setup part is finished.

As seen in the previous project, you can control the internal orange Teensy LED with pin 13. Configure the pin as an output using the command pinMode() and use the command digitalWrite() to give it the command HIGH, which will light the orange LED and show that the microcontroller is powered and working.

To control the external LEDs, you will use the same commands. You already connected the red LED to pin 5 and the green LED to pin 6. To show that the setup process is ongoing, you light up the red LED by giving it the HIGH command.

Now wait four seconds (=4000 milliseconds) using the delay() command in order to simulate the setup process, which will take around four seconds to be completed in your final quadcopter code.

To indicate that the setup process is finished, turn off the red LED using the command LOW and subsequently turn on the green LED.

The code in the loop part runs continuously. Because you do not write any commands in this part, the green LED will be continuously illuminated as demanded in the last line of the setup part.

Testing

Upload your new code to your Teensy using the USB cable and verify that all LEDs light up in the correct order. Only the green LED should remain on after four seconds.

1	`void setup() {`	Initialize the setup part
2 3	`pinMode(13, OUTPUT);` `digitalWrite(13, HIGH);`	Turn on the internal LED
4 5	`pinMode(5, OUTPUT);` `digitalWrite(5, HIGH);`	Turn on the red LED
6	`delay(4000);`	Wait 4 seconds

7
8
9
10
	`digitalWrite(5, LOW);`
	`pinMode(6, OUTPUT);`
	`digitalWrite(6, HIGH);`
	`}`
		Turn off the red LED and turn on the green LED

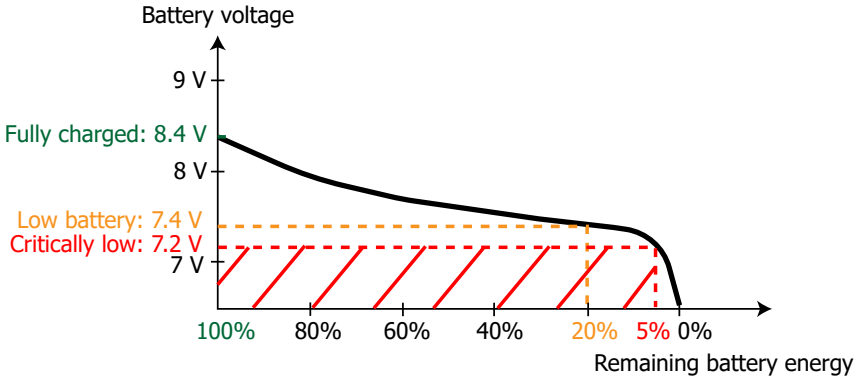
11 12	`void loop() {` `}`	Start the loop part

Understanding digital output pins

You used the pins 5, 6 and 13 as digital pins, meaning that their output voltage is binary: either HIGH (3V) or LOW (0V). This very simple command is sufficient to turn on a LED (when the 3V voltage is applied) or turn off the LED (when the 0V voltage is applied). The current necessary to light the LEDs is provided with the resistors you placed in series; through Ohm's law, you can calculate that a voltage of 3V and a resistance of 100 Ω gives a current of 3V/100 Ω=0.03 Ampere or 30 mA.

Project 3

Reading your battery level



The evolution of the battery level in function of the battery voltage is displayed by the figure above. It is important to notice that discharging your battery to a too low voltage can degrade the battery and lead to a reduced capacity over time. Therefore, a good guideline for prolonged battery lifetime is to not discharge your 2S battery below its nominal voltage of 7.4V. Because the battery voltage fluctuates during flight and can drop temporarily when you suddenly increase the throttle, your flight controller will check if the voltage is above 7.5V before starting the motors.

Now how can you measure the voltage of the battery? Easy: the voltage applied to any pin of your microcontroller can be read digitally. Unfortunately, there is one catch: the pins of the Teensy are only 3.3V-tolerant, meaning that applying a voltage higher than 3.3V can damage the microprocessor. Therefore, you need to use a **voltage divider**: this electronic circuit divides the voltage of the battery to a value low enough to be used by your Teensy. Consider the first circuit displayed on the right: through Ohm's law, the current I is equal to the battery voltage $V_{battery}$ divided by the resistance R_1.

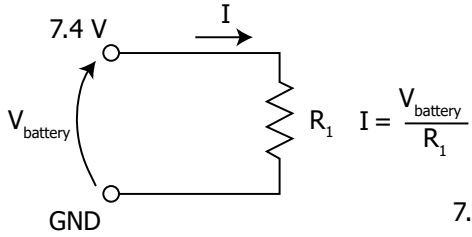
In the second circuit, a second resistance R_2 is used. The battery voltage is now equal to the current divided by the sum of two resistances. With the third circuit, you connect a pin of the Teensy between both resistances.

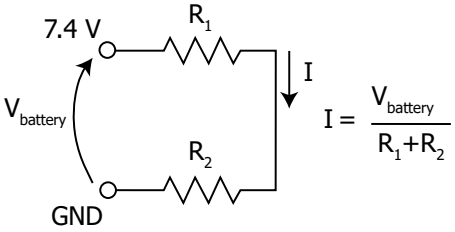
Learn to measure voltage and battery lifetime

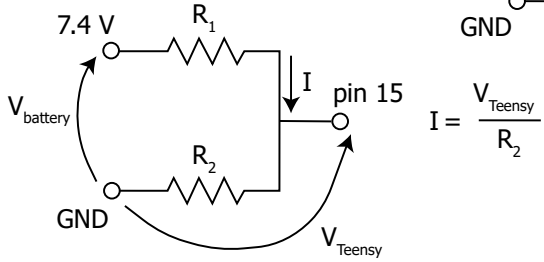
A critical part of your quadcopter is the battery; it stores enough energy to let you fly for quite a while. But how do you know when the battery is almost empty? In this project, you will learn how the battery voltage drops during the flight and measure it in order to estimate the remaining battery lifetime.

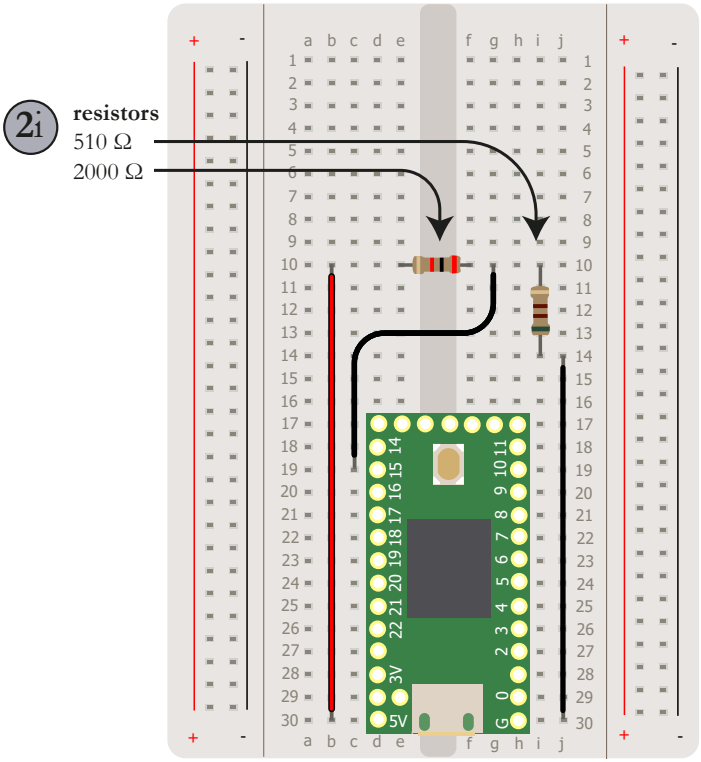
The battery you use in this project is a **2** cell lithium-polymer battery, where the cells are placed in **S**eries (=**2S**). Each cell has a nominal voltage of 3.7V and since the cells are placed in series, the total nominal voltage is equal to 7.4V. A 3S battery would give you 3 x 3.7 = 11.1V. The nominal voltage is the reference voltage of the battery, but you will always charge the battery up to the charge voltage, which is equal to 8.4V for a 2S battery.

When using a fully charged battery to fly your quadcopter, the battery voltage will drop from the charge voltage of 8.4V to the nominal voltage of 7.4V and even lower when you use more energy. This is inevitable and results in a lower thrust over time, because the speed of the motors is proportional to the provided voltage. Fortunately you can use this property also to your advantage, because by measuring the battery voltage you are able to estimate the remaining battery energy.









Coding

Lets first declare the voltage as a floating point number. To be able to measure the voltage multiple times without rewriting the same lines of code over and over, you will create the function battery_voltage. This function can be called as often as you want.

The analog voltage over pin 15 can be measured using the function analogRead(). Since the default resolution for analogRead is equal to **10** bit, a voltage of 0V gives you the digital number 0 and the maximal input voltage of 3.3V gives the digital number 2^{10}-1=1023. Moreover you have built a 1:5 voltage divider. This means that the battery voltage is equal to the measured voltage divided by 1023 / (3.3 x 5) = 62.

You will visualize the voltage at pin 15 in real-time on your computer with the serial monitor. Set the speed at which the Teensy communicates with your laptop to 57600 bits per second.

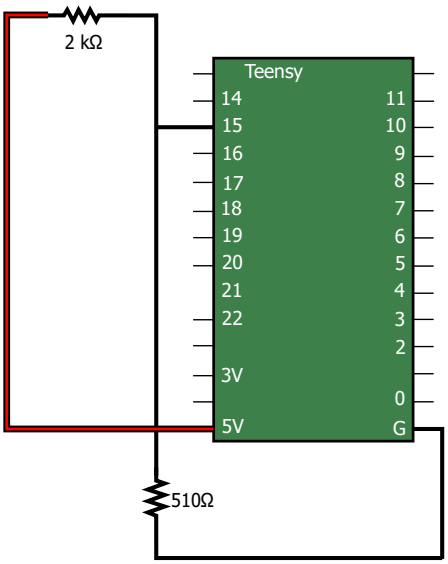
The voltage applied to the pin of the Teensy (which will be pin 15) is equal to the current divided by the second resistance R_2. Since the current I will be the same for the second and third circuit, the following equation holds:

$$\frac{V_{battery}}{R_1 + R_2} = I = \frac{V_{Teensy}}{R_2}$$

$$V_{Teensy} = V_{battery} \cdot \frac{R_2}{R_1 + R_2}$$

By choosing the value of R_1 to be equal to 2000 Ω and the value of R_2 to be equal to 510 Ω, V_{Teensy} becomes equal to $V_{battery}$ divided by 5. You have now designed a 1:5 voltage divider! With a battery voltage of 8.4V, the voltage measured by your Teensy equals 1.7V, low enough to respect the 3.3V tolerance of the Teensy pins.

To test your circuit, you will not yet connect your battery but use the 5V output pin of the Teensy as voltage source, and measure this value with pin 15 and your new voltage divider. Connect the 5V pin with a 2000 Ω resistor to pin 15 and the ground pin with a 510 Ω resistor to pin 15 as shown on the figure to the left. You are now ready to code.




```
1 float Voltage;
2 void battery_voltage(void) {
3         Voltage=(float)analogRead(15)/62;
4 }
```

Read the battery voltage

```
5 void setup() {
6         Serial.begin(57600);
```

Setup the serial monitor

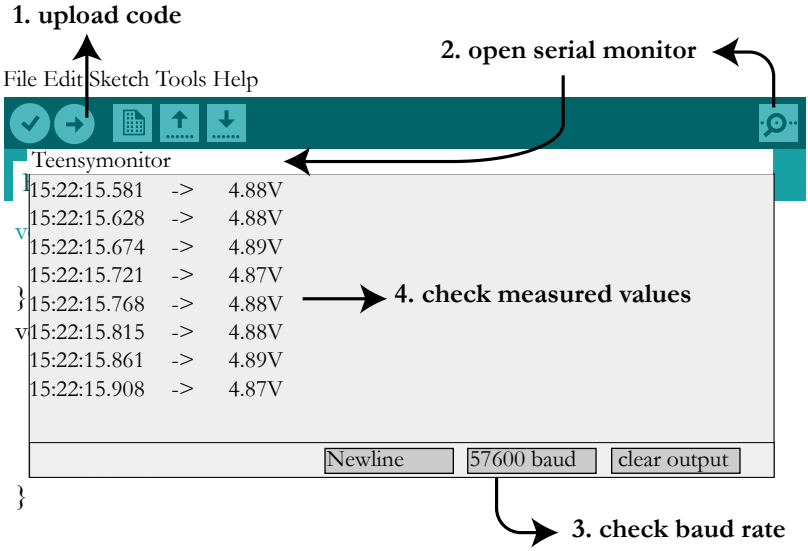
Measure the voltage each 50 milliseconds and print it to the serial monitor, with each time the unit V behind.

Testing

Upload the code and open the serial monitor (Ctrl+Shift+M or click on the serial monitor icon) with the USB adapter still connected to your Teensy. To see values that make sense, you should set the baud rate such that it corresponds with the baud rate that you have chosen in the code, namely 57600 baud. Now you should see the measured values, who will be more or less equal to 5V. When you connect the battery in a later stage, the measured voltage will vary between 8.4V and 7V.

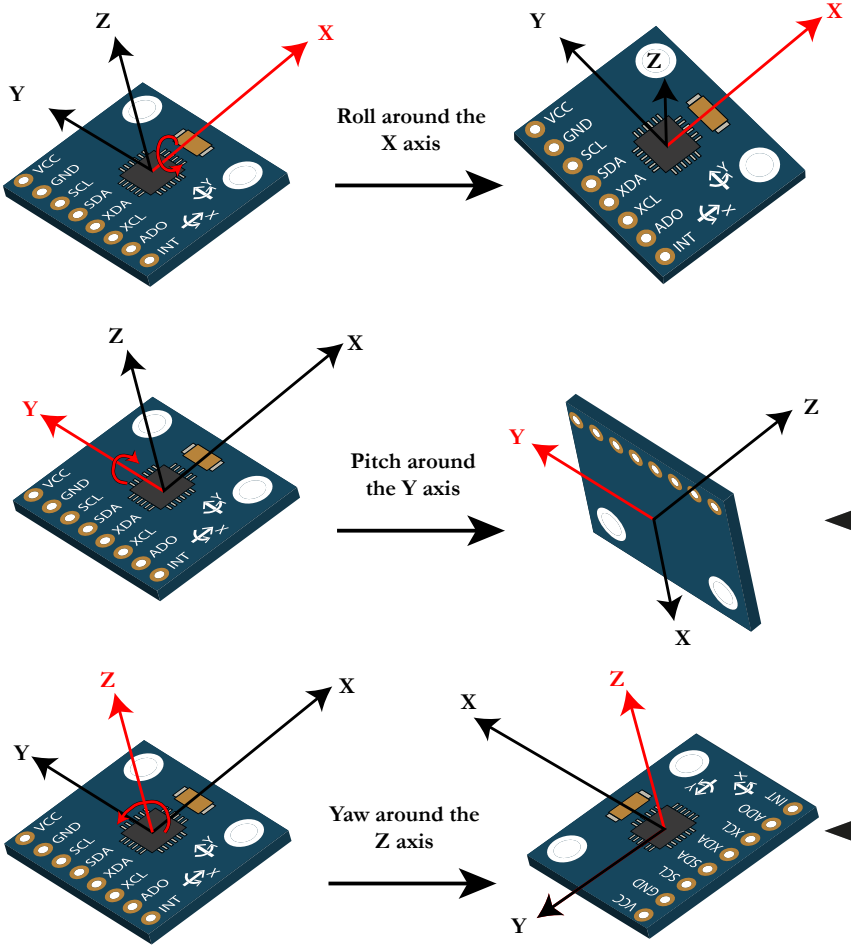
```
7         pinMode(13, OUTPUT);
8         digitalWrite(13, HIGH);
9  }
10 void loop() {
11        battery_voltage();
12        Serial.print(Voltage);
13        Serial.println("V");
14        delay(50);
15 }
```

Print the battery voltage to the serial monitor



Project 4

Sensing the rotation rate



Measure the rotation of your quadcopter

To stabilize your quadcopter, you need measurements of its three-dimensional orientation. In rate control mode, it is sufficient to know the rotation rates when rolling, pitching and yawing. A sensor able to record these rotation rates is called a gyroscope. During this project, you will learn how to read the data sent from your gyroscope.

The gyroscope that you will use is included in the MPU-6050, a low-cost off-the shelve orientation sensor. While the MPU is not a very precise sensor, its accuracy is sufficient to get great results balancing your quadcopter. You will use the gyroscope to measure the roll rate, pitch rate and yaw rate: this means that you do not measure absolute angles in degrees (°), but rather angular rates in degrees per second (°/s). An angular rate of 30°/s for example, means that you rotate 30° each second and will perform a full 360° rotation in 12 seconds (360°/ (30°/s)). You will learn later on that you can use these rotational rates to keep the quadcopter balanced; for example when you want the drone to stay at its current orientation, its angular rate needs to be 0°/s.

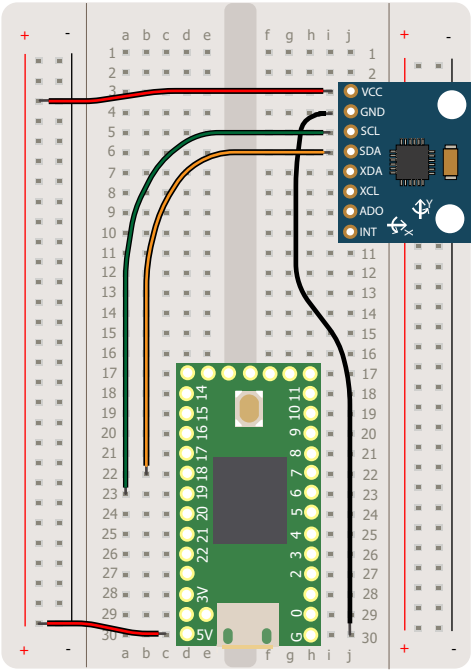
Before you continue, you need to be fully aware of the direction of the roll, pitch and yaw rotational rates. These three rotations are visualized on the figure to the left:

- A roll rotation means that you rotate clockwise around the X axis of the gyroscope.
- A pitch rotation means that you rotate clockwise around the Y-axis of the gyroscope.
- A yaw rotation means that you rotate counter clockwise around the Z axis of the gyroscope.

The respective axis around which you turn, is the only axis that keeps pointing to the same direction during the turn: on the figure, this is each time the red axis.

Mounting instructions of the gyroscope on your quadcopter

Notice that the X and Y axes and their respective rotation directions are also written physically on the MPU-6050 sensor itself. When building the quadcopter and soldering the MPU-6050 to it, always make sure that the axes written on the sensor are aligned with the roll, pitch and yaw axes of the quadcopter itself.



Coding

The code for the I2C protocol is rather complex, so you use a predefined library called Wire.h. This was normally installed automatically when you installed Teensyduino, but you can still install it at this point if necessary: in the Arduino IDE, go to sketch → include library → manage libraries and type Wire in the search bar. Click on install and you are ready to use it.

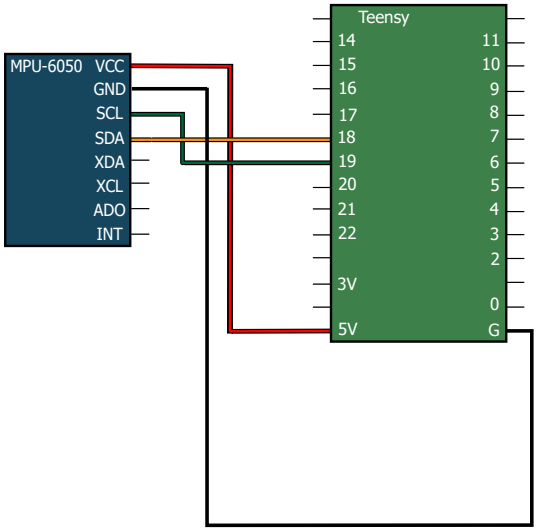
Define the roll, pitch and yaw rates in degrees/s (°/s) as global variables. You will write the output of the measurements from the sensor to these variables.

Use once again a function to get the data from the gyro. With the I2C protocol, each device (sensor) has its unique address. For our MPU-6050, this address can be found in the register documentation and has a default value of 0x68. Routing function Wire.beginTransmission to this address starts the communication with the sensor.

Sensor documentation

All information about sensors and their setup can easily be accessed online; try and look up the MPU-6050 register map and product specification documentation.

How can you connect the MPU-6050 with your microcontroller? The communication protocol that you will use is I2C. This protocol needs two wires: a serial communication line (=SDA) through which the data can be transferred bit by bit, and a line that carries the clock signal (=SCL). The exact design of this protocol is beyond the scope of your project, but one of its advantages is the transfer of information from multiple sensors using the same SDA and SCL lines to the microcontroller. This will prove useful when you will connect a barometric sensor later on.



The wiring of the MPU-6050 to the Teensy is rather straightforward: connect 5V to Vcc and G to GND to feed the sensor. Subsequently, you connect the serial communication output SDA on the sensor to pin 18 of the Teensy and the clock signal output SCL to pin 19. You are now ready to start programming.

```
1  #include <Wire.h>
```

Include the Wire library

```
2  float RateRoll, RatePitch, RateYaw;
```

Declare the global variables

```
3  void gyro_signals(void) {
4          Wire.beginTransmission(0x68);
```

Start I2C communication with the gyro

Some registers can be accessed to select some predefined options of the MPU-6050. One of these options is a low pass filter, which will be necessary to filter out high frequency vibrations and hence sharp increases and decreases in rotation rates that are caused by running motors. The configuration register, where you can activate this option, has the hexadecimal address 0x1A according to the documentation (this is equal to a decimal address of 26). The options for the low-pass filter correspond to bits 0, 1 and 2 in this address (the Digital Low Pass Filter DLPF setting). You choose a low pass filter with a cut-off frequency of 10 Hz, which corresponds to a value for the DLPF setting of 5. This corresponds in turn to the following binary representation: 00000101. Converting this to a hexadecimal value gives an address of 0x05.

The table from the register documentation that explains the configuration register of the MPU-6050 is displayed on the right. The binary representation for setting the value for the DLPF is given in the third column.

In addition to the low pass filter, you also need to set the sensitivity scale factor of the sensor. The measurements of the MPU-6050 are recorded in LSB (Least Significant Bit). Choose a sensitivity setting of FS_SEL=1 to set the scale factor to 65.5 LSB/(°/s). This means that 1°/s corresponds to 65.5 LSB. You will take into account this scale factor later on in the code. The gyroscope configuration register to activate this option has the hexadecimal address 0x1B (or a decimal address of 27). The FS_SEL setting of 1 corresponds to a 2-bit binary representation of 01. The other settings in the register can be set to zero, giving a binary representation of 00001000. Converting this to a hexadecimal value gives an address equal to 0x08.

The table from the register documentation that explains the gyroscope configuration register of the MPU-6050 is displayed on the right. The binary representation for the setting of the FS_SEL value is given in the third column.

Now you are ready to start importing the measurement values of the gyro. These are located in the registers that hold the gyroscope measurements, which have the hexadecimal numbers 43 to 48. You start writing to address 0x43 to indicate the first register you will use.

Request 6 bytes from the MPU-6050 such that you can pull the information of the 6 registers 43 to 48 from the sensor.

```
5   Wire.write(0x1A);
6   Wire.write(0x05);
7   Wire.endTransmission();
```

Switch on the low-pass filter

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1A	26	-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		
		0	0	0	0	0	1	0	1

```
8   Wire.beginTransmission(0x68);
9   Wire.write(0x1B);
10  Wire.write(0x8);
11  Wire.endTransmission();
```

Set the sensitivity scale factor

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

```
12  Wire.beginTransmission(0x68);
13  Wire.write(0x43);
14  Wire.endTransmission();
```

Access registers storing gyro measurements

```
15  Wire.requestFrom(0x68,6);
```

Registers 43 and 44 contain the gyro measurements of the rotational rate around the X axis, in LSB (Least Significant Bit). According to the documentation, they are the result of a unsigned 16-bit measurement. This means you will declare GyroX as an unsigned 16-bit integer int16_t. Because the measurement of the rotational rate around the X axis is spread out over two registers with each 8 bits, you will have to merge this information by calling Wire.read() twice.

You repeat the same code for registers 45 and 46 (rotational rate around the Y axis) and registers 47 and 48 (rotational rate around the Z axis).

The measurements are expressed in LSB but you want this information in °/s, not LSB. You have set the LSB sensitivity scale factor of the MPU-6050 equal to 65.5 LSB/(°/s). Therefore you just divide the values in LSB by 65.6 LSB/(°/s) to get the measurement values in °/s. Take care of converting the 16-bit integer values of the measurements in LSB to a floating point representation. As discussed earlier this project, the roll rate corresponds to the rotation around the X axis, the pitch rate to the rotation around the Y axis and the yaw rate to the rotation around the Z axis.

Set the clock speed of the I2C protocol to 400 kHz. This value comes from the product specifications of the MPU-6050 which states that communication with all registers of the device must be performed using I2C at 400 kHz. Use a delay of 250 milliseconds to give the MPU-6050 time to start.

To activate the MPU-6050, write to the power management register, which has the hexadecimal number 6B. All bits in this register have to be set to zero in order for the device to start and continue in power mode. Hence the hexadecimal address becomes 0x00.

Terminate the connection with the gyroscope and end the setup section.

In the loop part of the code, call your function and print the roll, pitch and yaw rates on the serial monitor. Wait 50 milliseconds after each loop to be able to read the values on the serial monitor and close the loop function.

```
16        int16_t GyroX=Wire.read()<<8 | Wire.read();
```

Read the gyro measurements around the X axis

```
17        int16_t GyroY=Wire.read()<<8 | Wire.read();
18        int16_t GyroZ=Wire.read()<<8 | Wire.read();
```

```
19        RateRoll=(float)GyroX/65.5;
20        RatePitch=(float)GyroY/65.5;
21        RateYaw=(float)GyroZ/65.5;
22  }
```

Convert the measurement units to °/s

```
23  void setup() {
24        Serial.begin(57600);
25        pinMode(13, OUTPUT);
26        digitalWrite(13, HIGH);
```

```
27        Wire.setClock(400000);
28        Wire.begin();
29        delay(250);
```

Set the clock speed of I2C

```
30        Wire.beginTransmission(0x68);
31        Wire.write(0x6B);
32        Wire.write(0x00);
```

Start the gyro in power mode

```
33        Wire.endTransmission();
34  }
```

```
35  void loop() {
36        gyro_signals();
37        Serial.print("Roll rate [°/s]= ");
38        Serial.print(RateRoll);
39        Serial.print(" Pitch Rate [°/s]= ");
40        Serial.print(RatePitch);
```

Call the predefined function to read the gyro measurements

Testing

Upload the code to your microcontroller and open the serial monitor. You will notice that not all values are equal to zero even though you do not move the MPU-6050:

Roll rate [°/s]= -8.70 Pitch Rate [°/s]= 0.89 Yaw Rate [°/s]= 1.95
Roll rate [°/s]= -8.69 Pitch Rate [°/s]= 0.92 Yaw Rate [°/s]= 1.97
Roll rate [°/s]= -8.66 Pitch Rate [°/s]= 0.87 Yaw Rate [°/s]= 1.94

It is normal when you do not have the same values as mentioned above. You will learn more on how to solve this phenomenon through calibration in the next project.

What are registers?

Registers are places on a microcontroller (the Teensy but also the MPU-6050, since this sensor has a microcontroller as well) that are used as:

- Fast storage locations to store data temporary.
- Locations where you can set predefined options.

You select a register by using its unique address, which is given in the documentation of the microcontroller or sensor. With the I2C arduino library, you use the function Wire.write(address) to select the register of choice.

If you select a register to set some predefined option, you once again use the Wire.write function. You find the predefined options once again in the documentation. Usually each register has a number of bits: you can set each bit to 0 or 1, which corresponds to different options. Converting the resulting binary representation to a hexadecimal representation gives you the argument for the Wire.write function.

If you select a register to read data, use the function Wire.read() after selecting the address and reserving the necessary bytes. Let's go back to the example of the low pass filter. Assume you want to set a low pass filter of 20 Hz instead of 10 Hz. You already know that the register address is 0x1A. The documentation of the MPU-6050 says that you need to set the value of DLPF_CFG to 4 for the 20 Hz filter. Moreover, DLPF_CFG occupies the first three bits of the 0x1A register:

```
41        Serial.print(" Yaw Rate [°/s]= ");
42        Serial.println(RateYaw);
43        delay(50);
44  }
```

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1A	26	-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		

The number 4 in a three bit binary representation is equal to 1 0 0: you just multiply each binary number with 2^n, where n is the bit number:

	Bit2	Bit1	Bit0
Binary representation	1	0	0
2^n	2^2=4	2^1=2	2^0=1
Decimal representation	1 x 4 + 0 x 2 + 0 x 1 = 4		

If you decide to not set an option for bit3 to bit7 and stick with the default value, the full 8 bit binary and decimal representation becomes:

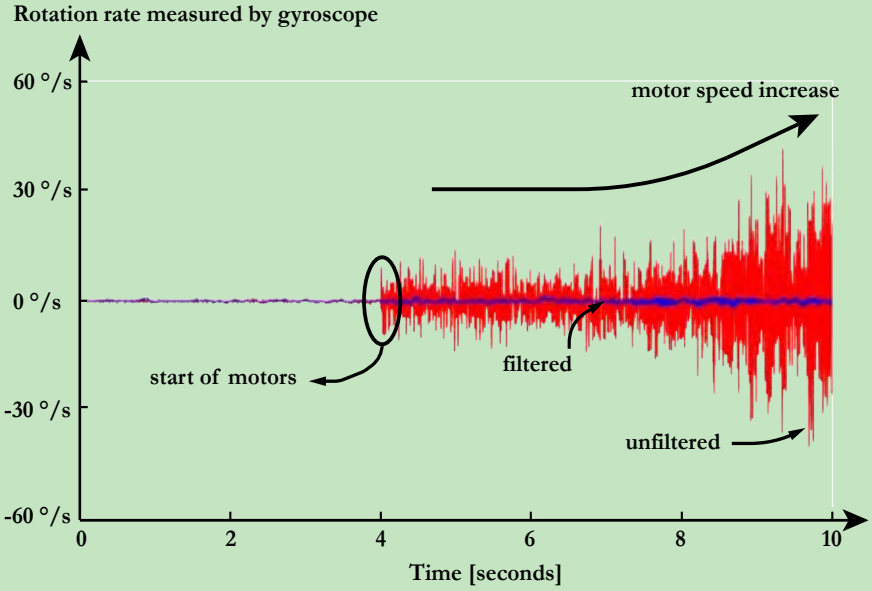
	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Binary representation	0	0	0	0	0	1	0	0
2^n	2^7=128	2^6=64	2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1
Decimal representation	0 x 128 + 0 x 64 + 0 x 32 + 0 x 16 + 0 x8 + 1 x 4 + 0 x 2 + 0 x 1 = 4							

The Wire function use hexadecimal representation: conversion from decimal to hexadecimal is a little bit more complex, but you can use an online converter. 4 in hexadecimal form is equal to 0x04.

A low pass filter?

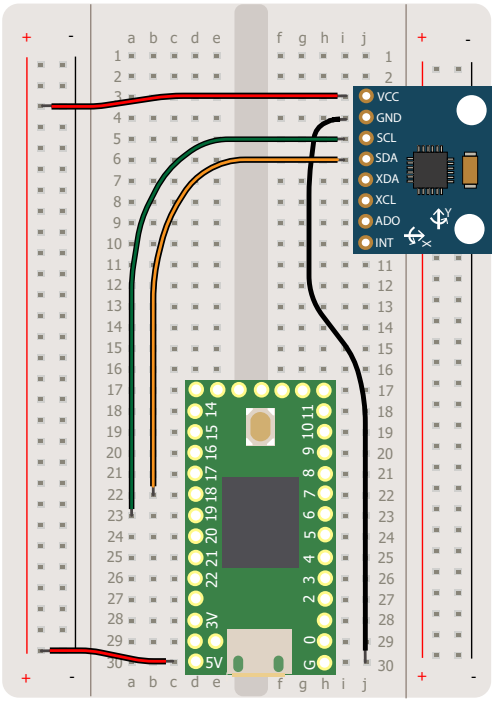
In line six of the code, you choose the option to send the measurement data through a low-pass filter with a cut-off frequency of 10 Hz. This is a crucial line of code, as the flight controller would not be able to stabilize the drone without it. Why? Your gyroscope is a very sensitive sensor and its readings will be dramatically affected by the vibrations caused by the brushless motors. The sample rate of the gyroscope is equal to 8 kHz, which corresponds to a measurement each 1/8000 = 0.000125 seconds. The high frequency vibrations from the motors will cause small but very fast accelerations of the quadcopter frame, which are recorded by the gyroscope. The faster the motors spin, the larger the vibrations become; the figure to the right shows the readings of the gyroscope with the motors switched off, switched on and with increasing throttle. During all three cases, the quadcopter stays stationary on the ground so the rotation rate should be equal to 0°/s. From the figure you observe that once the motors are started, the unfiltered gyroscope values start to fluctuate a lot. It is clear that it becomes impossible to stabilize your quadcopter with measurement values that vary between 40 and -40°/s while the quadcopter itself remains stationary and the real rotation rate is equal to 0 °/s.

To solve this issue, you use a low pass filter with a cut-off frequency of 10 Hz. The filter attenuates the measurements of the sensor with a frequency of 10 Hz and higher. This means that sensor variations that happen faster than 1/10=0.1 seconds will only have a limited impact on the final measurement values. The value of 10 Hz is chosen through trail-and-error during testing of the quadcopter; motor vibration frequencies change with each brushless motor and damping of the vibration depends on the whole frame. The blue line on the figure to the right shows the filtered values; they are not affected by the vibrations caused by the brushless motors and are hence suited for your flight controller.


Rotation rate measured by gyroscope
60 °/s
30 °/s
0 °/s
-30 °/s
-60 °/s
motor speed increase
start of motors
filtered
unfiltered
0
2
4
6
8
10
Time [seconds]

Project 5

Gyroscope calibration



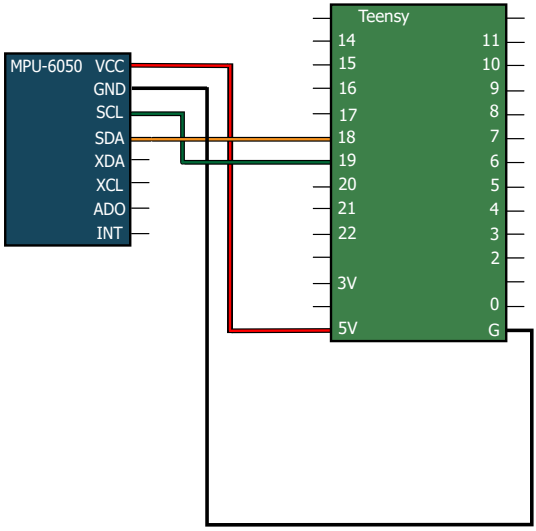
Coding

You need four additional variables for calibration: the calibration values for the roll, pitch and yaw rotation rate and a variable to keep track of the number of values you have already recorded to use for the calibration.

Teach your gyroscope the correct rotation rates

With this short project, you will teach your gyroscope the correct rotation rates using a technique called calibration. You will use known rotation rates to correct the values given by your sensor.

At the end of the previous project, you saw that the rotation measurements given by your gyroscope were not correct; even though you did not move the MPU-6050, it still gave non-zero values. You still need to tell the instrument what its physical reference point is. Adjusting the measurements of a sensor such that they correspond with real physical values is called calibration.



In the case of a gyroscope, the easiest reference value that you can use is the rotation rate when the sensor is not moving: this rotation rate should obviously be zero. Because the gyro measurements always tend to fluctuate due to small vibrations in the environment, you take the average of a large number of uncorrected measurement values when the sensor is not moving, calculate their average value and subtract this average value from all future measurement values. You can easily integrate these additional calibration steps in the code of the previous project. The electronic circuit stays the same.

```
1  #include <Wire.h>
2  float RateRoll, RatePitch, RateYaw;
3  float RateCalibrationRoll, RateCalibrationPitch,
       RateCalibrationYaw;
4  int RateCalibrationNumber;
5  void gyro_signals(void) {
6      Wire.beginTransmission(0x68);
```

Declare the calibration variables

```
7       Wire.write(0x1A);
8       Wire.write(0x05);
9       Wire.endTransmission();
10      Wire.beginTransmission(0x68);
11      Wire.write(0x1B);
12      Wire.write(0x08);
13      Wire.endTransmission();
14      Wire.beginTransmission(0x68);
15      Wire.write(0x43);
16      Wire.endTransmission();
17      Wire.requestFrom(0x68,6);
18      int16_t GyroX=Wire.read()<<8 | Wire.read();
19      int16_t GyroY=Wire.read()<<8 | Wire.read();
20      int16_t GyroZ=Wire.read()<<8 | Wire.read();
```

In the setup part of the program, create a loop in which you take 2000 measurement values from the gyroscope. Each value is taken 1 millisecond after the other (hence the delay(1)) which means this step takes 2000 x 1 ms= 2 seconds. You add all measured values in the Roll/Pitch/YawRateCalibration variables. During this measurement step, it is important to not move your gyroscope as the goal is to determine the measured values at a rotation rate of zero.

Take the average calibration value by dividing the sum of the 2000 measurement values by 2000. Now you have the measurement values at which the rotation rates are zero.

Once the setup is finished and you have determined the calibration values, subtract them from the measured values in order to get the correct physical values. Print the corrected values to the serial monitor.

```
21        RateRoll=(float)GyroX/65.5;
22        RatePitch=(float)GyroY/65.5;
23        RateYaw=(float)GyroZ/65.5;
24  }
25  void setup() {
26        Serial.begin(57600);
27        pinMode(13, OUTPUT);
28        digitalWrite(13, HIGH);
29        Wire.setClock(400000);
30        Wire.begin();
31        delay(250);
32        Wire.beginTransmission(0x68);
33        Wire.write(0x6B);
34        Wire.write(0x00);
35        Wire.endTransmission();
36        for (RateCalibrationNumber=0;
37              RateCalibrationNumber<2000;
38              RateCalibrationNumber ++) {
39                    gyro_signals();
40                    RateCalibrationRoll+=RateRoll;
41                    RateCalibrationPitch+=RatePitch;
42                    RateCalibrationYaw+=RateYaw;
43                    delay(1);
44        }
45        RateCalibrationRoll/=2000;
46        RateCalibrationPitch/=2000;
47        RateCalibrationYaw/=2000;
48  }
49  void loop() {
50        gyro_signals();
51        RateRoll-=RateCalibrationRoll;
52        RatePitch-=RateCalibrationPitch;
53        RateYaw-=RateCalibrationYaw;
54        Serial.print("Roll rate [°/s]= ");
55        Serial.print(RateRoll);
56        Serial.print(" Pitch Rate [°/s]= ");
57        Serial.print(RatePitch);
58        Serial.print(" Yaw Rate [°/s]= ");
59        Serial.println(RateYaw);
60        delay(50);
61  }
```

Perform the calibration measurements

Calculate the calibration values

Correct the measured values

Testing

When you run the code and open the serial monitor, the roll, pitch and yaw rates should be almost zero when you do not move the gyroscope. Remember that during the setup phase, when no values are yet displayed on the serial monitor, you should not move the gyroscope in order to ensure a correct calibration.

- Roll rate [°/s]= 0.09 Pitch Rate [°/s]= -0.10 Yaw Rate [°/s]= -0.03
- Roll rate [°/s]= 0.09 Pitch Rate [°/s]= -0.09 Yaw Rate [°/s]= -0.03
- Roll rate [°/s]= 0.04 Pitch Rate [°/s]= -0.04 Yaw Rate [°/s]= -0.03

Now try to experiment by moving the gyroscope in the directions displayed in the figure to the right. When you for example pitch around the Y axis from 0 to 45°, wait and go back to 0°, the pitch rate should first increase with a value proportional on how fast you rotate, subsequently fall to around 0°/s and then go negative with a value proportional on how fast you rotate back to 0°.

Pitch from 0 to 45° and hold at 45°:

- Roll rate [°/s]= 0.01 Pitch Rate [°/s]= 0.02 Yaw Rate [°/s]= 0.00
- Roll rate [°/s]= -0.06 Pitch Rate [°/s]= 185.71 Yaw Rate [°/s]= -0.10
- Roll rate [°/s]= -0.05 Pitch Rate [°/s]= 0.06 Yaw Rate [°/s]= 0.02

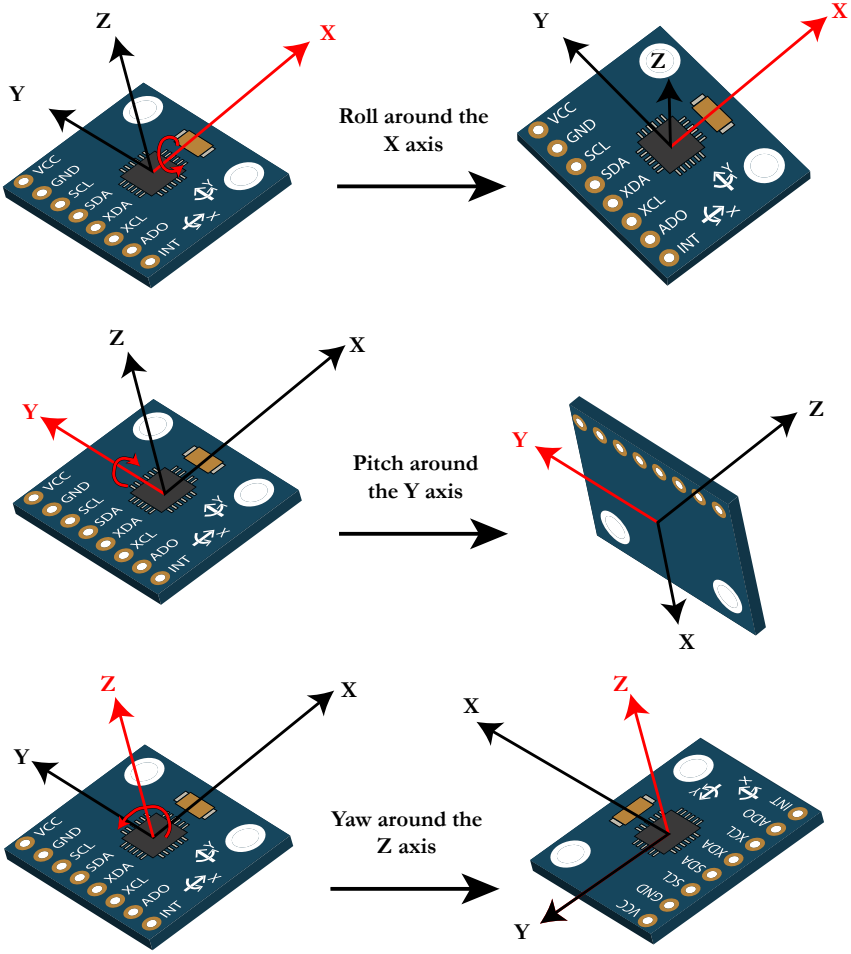
Pitch from 45° back to 0° and hold at 0°:

- Roll rate [°/s]= -0.05 Pitch Rate [°/s]= 0.06 Yaw Rate [°/s]= 0.02
- Roll rate [°/s]= 0.65 Pitch Rate [°/s]= -177.02 Yaw Rate [°/s]= 0.39
- Roll rate [°/s]= -0.03 Pitch Rate [°/s]= 0.06 Yaw Rate [°/s]= 0.00

Try the same test for the other directions to verify that your code is working properly.

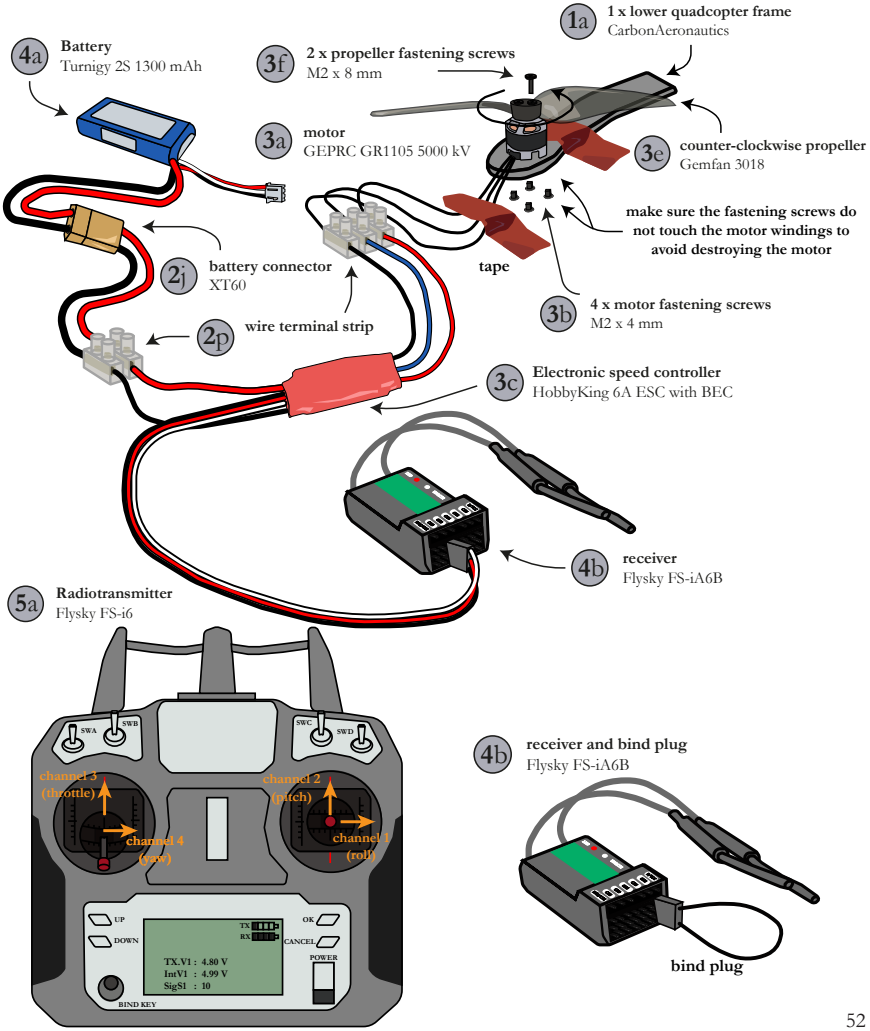
Time to fly?

The calibration needs to be performed during each start-up procedure, because the gyroscope measurement values tend to drift over time. You cannot start the motors yet during calibration, because their vibrations will impact the quality of the calibration. This means that the setup procedure takes some seconds before you can actually start the motors and begin your flight. That is why some projects ago, you learned to signal the status of your quadcopter with the red and green LEDs. Be mindful also to not move your quadcopter during this startup phase, as this will affect the calibration quality and hence the smoothness of your subsequent flight.


Z
X
Y
VCC
GND
SCL
SDA
XDA
XCL
ADO
INT
Roll around the
X axis
Y
X
Z
Pitch around
the Y axis
Y
Z
X
Z
X
Y
Yaw around the
Z axis
X
Z
Y

Project 6

Take your motors for a spin



Test your radio, motors and ESCs

You will now test your radiotransmitter, receiver and motors. Use this opportunity to calibrate each ESC and verify that all motors spins in the correct direction.

Each radio, ESC (Electronic Speed Controller) and motor combination needs to be tested, calibrated and verified before you start soldering all parts together. Let's start with the first motor:

a. Set up your test stand

1. Slide a propeller down the motor shaft and push it firmly on the top of the motor. For motor 1, you need a counter-clockwise propeller: the leading edge of this propeller must be the first to turn counter-clockwise as shown on the picture to the left.
2. Fasten the propeller with two long M2 screws but be sure that the screw does not touch the inner motor windings.
3. Attach the motor to the drone frame with four short M2 screws.
4. Attach the drone frame firmly to your desk using tape.
5. Connect the three black motor wires using a wire terminal strip with the black, blue and red cables coming out of the ESC. It does not matter yet which motor wire is connected with which ESC wire.
6. Connect the red and black cable of the XT60 plug using another wire strip with the red and black power wires of the ESC. Do not connect the battery yet.
7. Connect the white, red and black cables with channel 3 of the receiver as visualised on the picture.

b. Bind the receiver to the radiotransmitter through the bind plug

1. Make sure the throttle stick on the radiotransmitter is in the lowest possible downward position.
2. Turn on the POWER button of your radiotransmitter while simultaneously holding the BIND KEY button. The text *RX Binding...* should be displayed.
3. Connect the bind plug with the B/VCC pins on the receiver as shown on the picture. Keep the connection between the receiver and ESC in place.
4. Connect your battery using the XT60 plug. The red LED on the receiver should illuminate and your radiotransmitter should beep one time, indicating that binding is successful. The *text SigS1* on the radiotransmitter confirms binding of the receiver. Disconnect the battery again.
5. Remove the bind plug. When connecting the battery again, your radiotransmitter should automatically connect to your receiver.

c. Test your motor and verify its turning direction

1. Turn on the radiotransmitter through the POWER button.
2. Connect your battery using the XT60 plug. You should hear one beep from your radiotransmitter indicating that it is connected with the receiver, and subsequently **two** beeps from the motor to indicate that you are connected to a **2**S battery followed by two more beeps from the motor indicating that the ESC preparation is completed.
3. Now slowly increase the throttle stick on your radiotransmitter to turn on the motor and spin it at increasing speeds.
4. Verify that the propellers are spinning in the correct direction (motors 1+3: counter-clockwise, motors 2+4: clockwise). If they do not spin in the correct direction, remove the battery and switch two of the ESC wires going to the motor with each other to reverse the spinning direction. Note: if you connect the black, blue and red ESC wires in the correct order with the black motor wires as shown in the figure, the spinning direction will always be correct.

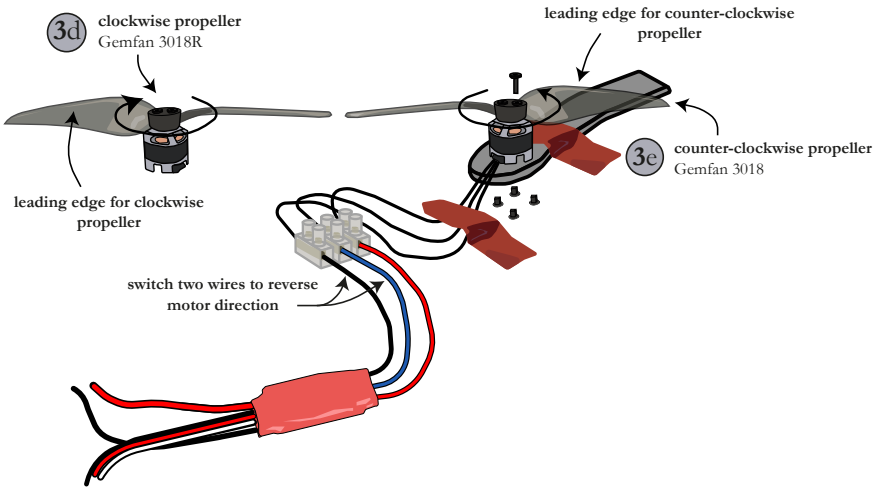
d. Calibrate the ESC

ESC calibration always necessary to be able to control your quadcopter. Through calibration, you tell the ESC what the upper and lower positions of the radiotransmitter sticks are. Carry out the calibration with the help of the following steps:

1. Make sure your battery is not plugged in and the radiotransmitter is turned off.
2. Turn on the radiotransmitter and put the throttle stick to its uppermost position. When connecting the battery, this will make sure the ESC goes in programming mode.
3. Do **not connect the battery yet but first read these instructions**: after connecting the battery, you will first hear one beep from the radiotransmitter, next you hear subsequent beeps from the motor. You will need to move the throttle stick to its lowest position **between the first and the fourth beep of the motor**! If you are too late, do not attempt anymore to lower the throttle stick but just disconnect the battery and try again starting from step 1.
4. When you understand the instructions from step 3, connect the battery, wait until the beep from the radiotransmitter has passed and lower the throttle stick to its lowest position between the first and the fourth beep of the motor.
5. After two seconds, the motor should give once again two times two beeps indicating that the calibration is finished. Congratulations, you finished your motor setup! Go once again to step c to test the throttle response.

c. Testing motors 2 to 4

Steps a, c and d need to be carried out for the other motors and ESC too. **Try to reverse the spinning directions of some motors by switching the ESC wires and changing the propellers.**



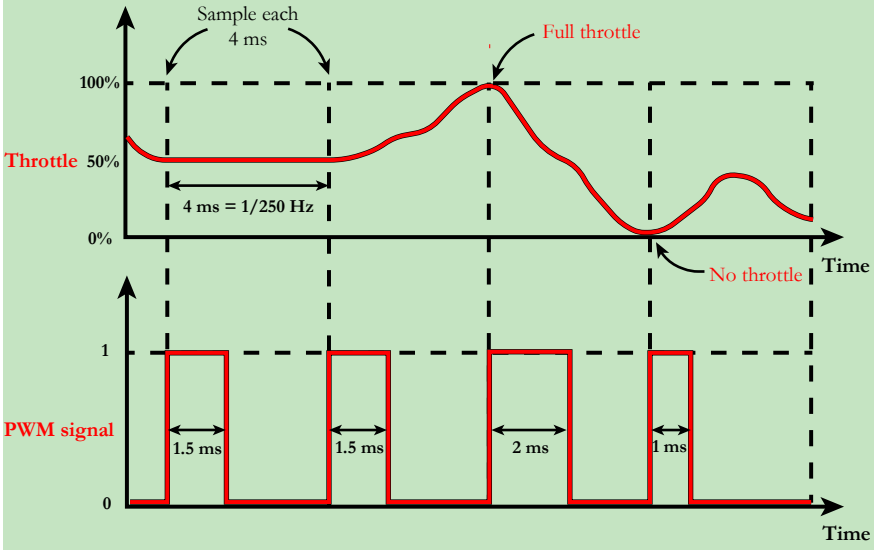
ESC programming

Not only ESC calibration can be carried out by putting the ESC into programming mode, but other settings can be adjusted as well. The first four beeps are followed by four beeps with a slightly different noise, indicating a different setting. By moving your throttle stick down during these beeps, you can choose to activate the corresponding setting. Through this method, you can choose from multiple settings; more information can be found in the datasheet of your ESC.

ESC control through PWM

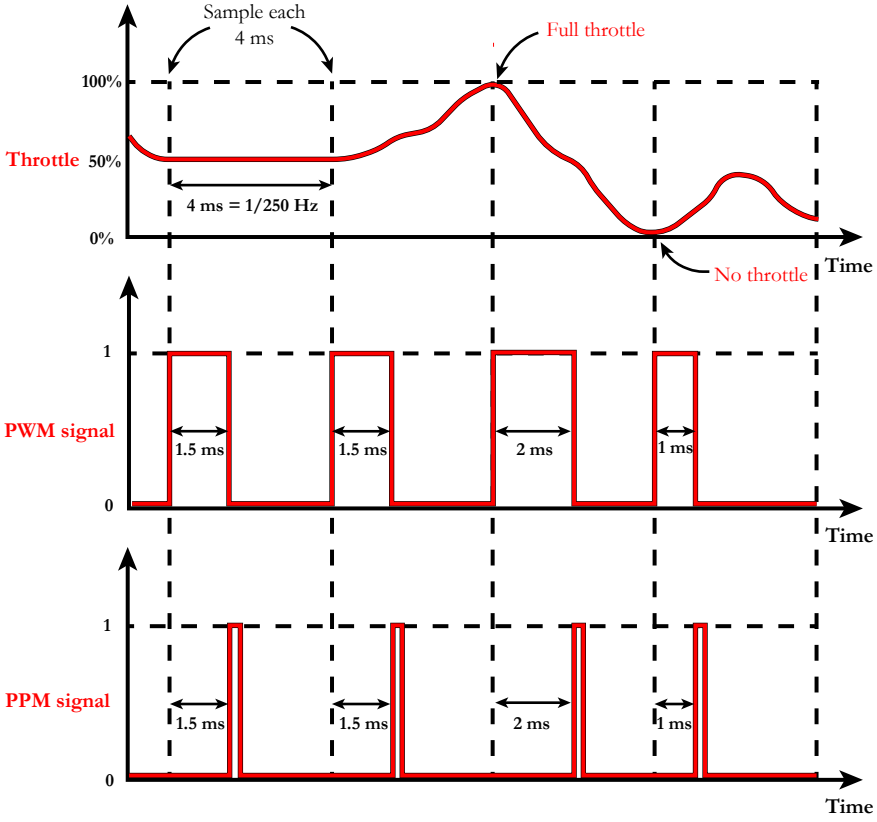
You control the speed of your motor through the ESC, which in turn gets its commands from channel 3 (the throttle channel) of your receiver. The receiver sends Pulse Width Modulated (PWM) signals to the ESC to this channel, indicating a desired throttle value between 0 and 100%. But what is a PWM signal, really?

In essence, a PWM signal is just a signal that varies between a HIGH voltage (for example 5V) and a LOW voltage (for example 0V), or between 1 and 0 to keep the concept simple. It is the time length during which the signal stays HIGH that is used to transfer information. Suppose for example that a time length of 1 ms HIGH corresponds with no throttle (0%) and a time length of 2 ms HIGH corresponds with full throttle (100%). The PWM frequency of most receivers and ESCs is 4 ms (or 1/0.004=250 Hz), meaning that your chosen signal repeats itself every 4 ms with a length that corresponds with your throttle command. This concept is illustrated in the figure below and will be used later on to send commands to your ESCs using your Teensy instead of directly from the receiver.



Project 7

Receiving commands



Process commands sent to the receiver

The commands that you give through your radio controller are transmitted via radio waves and picked up by your receiver. The receiver then converts the radio waves to a signal which can be read by your microcontroller. You will now learn how to convert these signals from the receiver to variables that can be used further in the flight code.

There are multiple methods to transfer information through digital signals, one of which you already learned: Pulse Width Modulation (PWM). PWM is an easy method to send information of one radiochannel from the receiver to the microcontroller. However, receiving information from multiple radiochannels would require one signal cable to the microcontroller for each channel. This is cumbersome when you need a lot of channels, meaning you will have to master another technique: Pulse Position Modulation (PPM).

You already saw that you can use the width of a PWM signal to transfer information: a signal width of 1 ms (=1000 µs) corresponds for example with no throttle (0%) and a width of 2 ms (=2000 µs) corresponds with full throttle (100%). Using the technique of Pulse Position Modulation, you are able to transfer the same information using the position of the signal in time, instead of the width. With PPM, the width of each signal stays the same but its position changes each time depending on the value of the radiochannel.

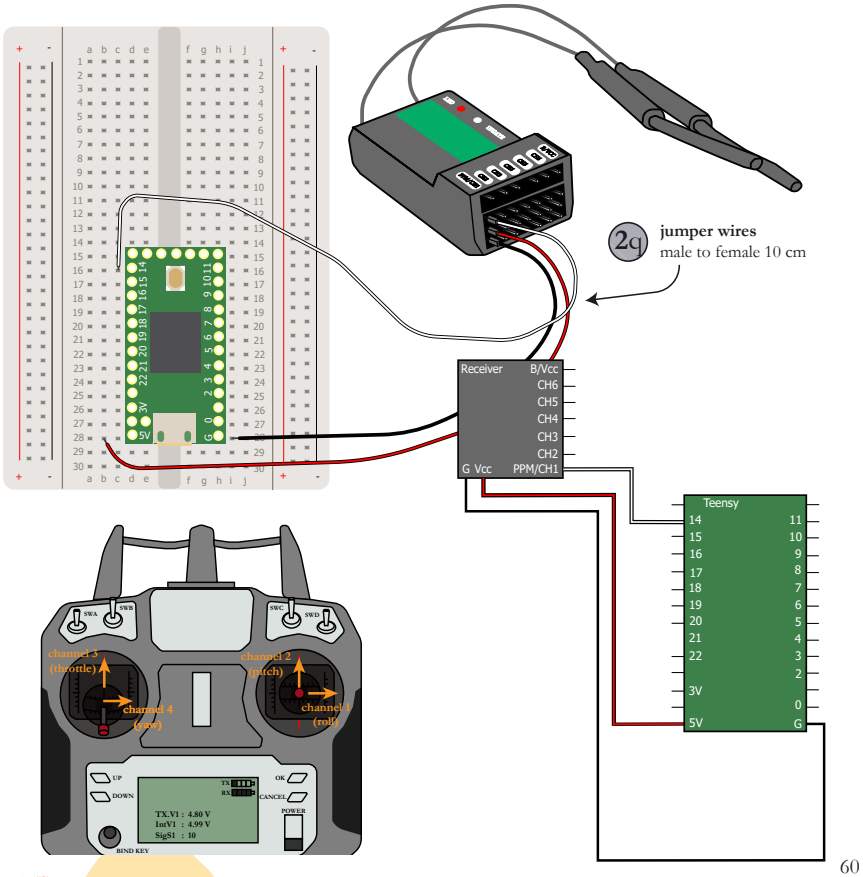
An example is displayed to the left: the first throttle value sampled from the radiochannel is equal to 50%. Using PWM, this corresponds to a signal width of 1.5 ms (=1500 µs). With PPM, the signal starts after 1.5 ms and has the same width each time. When 4 ms have passed, another throttle value is sampled and the cycle begins again.

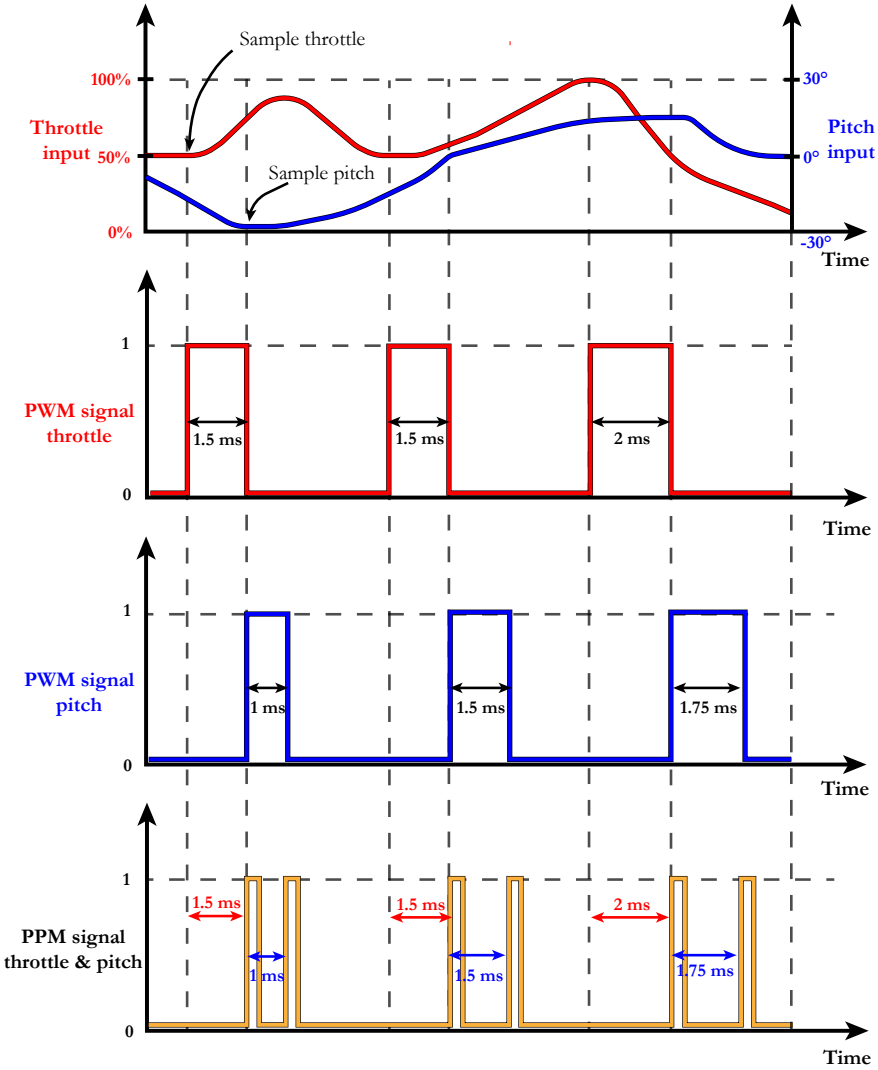
Setting up your radiotransmitter in PPM mode

The standard operating mode of your transmitter is PWM, not PPM! This means you have to configure the transmitter to use PPM for this project with the following steps:
Switch on the transmitter → hold the OK button for two seconds → choose System → go down and choose "RX setup" → go down and choose "Output mode" → choose "PPM" and hold the CANCEL button for two seconds to save your choice.

Now what about receiving information from multiple radiochannels? Consider a two-channel example: you want to receive information about the throttle and the pitch input. These inputs are independent of each other and require two separate signal cables if you would use PWM. When receiving the two PWM signals in the microcontroller, you will need two different analogRead() commands. Because they cannot be processed simultaneously in the microcontroller, either the throttle or the pitch values will be sampled (=read) first, after which the other signal follows directly. Besides the need for two signal cables, this also requires more calculation time from the microcontroller.

You are now ready to take advantage of the interesting property of PPM: because only the position of the signals changes and not their width, sampling both the throttle and pitch signals directly after each other allows you to keep track of their original values by measuring the time from each rising signal value. This means that with one signal cable, information from multiple signals can be 'transported'.





Using this knowledge, you can now easily built the connection from the receiver to the Teensy. Channel 1 of the receiver has also "PPM" written on it, which means that you can use these connectors to send PPM signals to your microcontroller. Use a cable to connect the second connector of the receiver starting from above to pin 14 of the Teensy as displayed on the figure to the left. Connect the third and fourth connector of the receiver to 5V and ground on the Teensy respectively. This will ensure that the receiver is fed from the microprocessor. You are now ready to start coding.

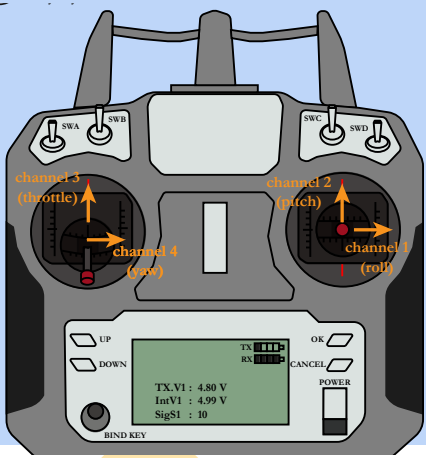
Coding

The code for handling Pulse Position Modulation is rather complex, so you again use a predefined library called PulsePosition.h. This library is normally already installed when you installed Teensyduino, but you can still install it at this point with the "manage libraries tool" like you already did with Wire.h. Next you create a PPM input object, which is in this case the receiver input. You track each pulse starting from their rising edge.

Use two global variables for this project: one array which can store up to eight channel values (initialized as eight zeroes) and one integer that stores the number of channels transmitter by the receiver.

To be able to read the receiver data multiple times in the code, you create a function. This function will first check how many channels are available by writing .available behind the PPM input object; if there are channels available, it reads the value of each channel and stores it in the array of receiver values.

Read the values sent from the receiver by calling the function defined in line 5. Print the available number of channels followed by the values for each channel.



Channels 1, 2, 3 and 4 correspond respectively with the roll, pitch, yaw and throttle inputs. Because the array numbering in the Arduino IDE starts with 0 instead of 1, ReceiverValue[0] actually corresponds with channel 1 or the value for roll.

Finally, you have to use a delay of 50 milliseconds to be able to read the values that will be displayed on the serial monitor.

```
#include <PulsePosition.h>
PulsePositionInput ReceiverInput(RISING);
```

Use the PulsePosition library

```
float ReceiverValue[]={0, 0, 0, 0, 0, 0, 0, 0};
int ChannelNumber=0;
```

Declare the variables to store channel info

```
void read_receiver(void){
      ChannelNumber = ReceiverInput.available();
      if (ChannelNumber > 0) {
         for (int i=1; i<=ChannelNumber;i++){
               ReceiverValue[i-1]=ReceiverInput.read(i);
               }
      }
}
void setup() {
      Serial.begin(57600);
      pinMode(13, OUTPUT);
      digitalWrite(13, HIGH);
      ReceiverInput.begin(14);
}
```

Define a function to read the receiver data

```
void loop() {
      read_receiver();
      Serial.print("Number of channels: ");
      Serial.print(ChannelNumber);
      Serial.print(" Roll [µs]: ");
      Serial.print(ReceiverValue[0]);
      Serial.print(" Pitch [µs]: ");
      Serial.print(ReceiverValue[1]);
      Serial.print(" Throttle [µs]: ");
      Serial.print(ReceiverValue[2]);
      Serial.print(" Yaw [µs]: ");
      Serial.println(ReceiverValue[3]);
      delay(50);
}
```

Read and display the PPM information on the serial monitor

Testing

Now you are ready to read the data send from the transmitter:

- Connect the Teensy to your computer through the USB cable;
- Upload to code to your Teensy and open the serial monitor.

If the radio transmitter is not yet switched on, the LED on the receiver should blink. The values displayed on the serial monitor will all be zero, except for the number of channels, which should be equal to -1. This is the default value when no channels are discovered.

When you switch on your radio transmitter, the LED on the receiver should stop blinking and the transmitter commands should be displayed on the serial monitor in microseconds [µs]. Since the roll, pitch and yaw sticks are always centred, the values you record will be around 1500 µs for the corresponding channels.

Now change the positions of the roll, pitch, throttle and yaw sticks and verify that they vary between 1000 and 2000 µs. You have now successfully made a radio-connection between your transmitter, the receiver and microcontroller!

Switching your radiotransmitter back to PWM mode

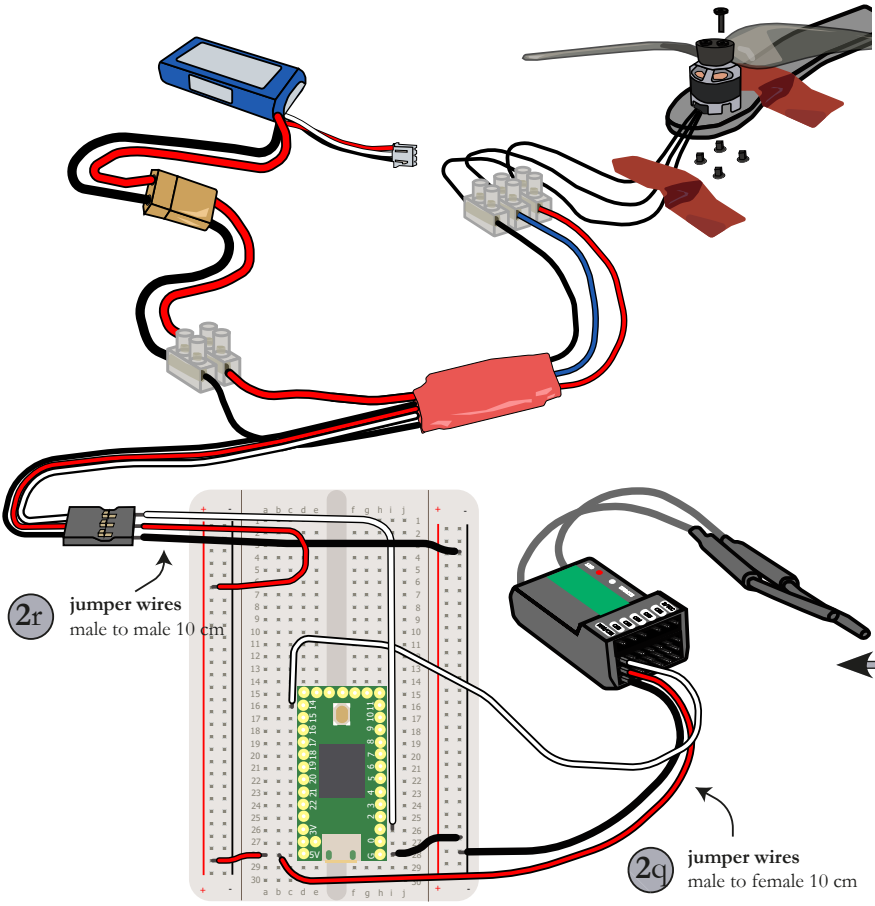
For the next projects, the receiver needs to stay in PPM mode. However, if you want to redo the previous project in which you controlled one motor and one ESC through PWM without a microcontroller in between, do not forget to switch your radiotransmitter back to its default PWM setting. This can be carried out using the same procedure as switching to PPM mode, which you learned at the start of this project.

```
11:26:52.994 -> Number of channels: -1 Roll [µs]: 0.00 Pitch [µs]: 0.00    ...
11:26:53.041 -> Number of channels: -1 Roll [µs]: 0.00 Pitch [µs]: 0.00    ...
```

```
11:26:53.089 -> Number of channels: 8 Roll [µs]: 1499.97 Pitch [µs]: 1498.99    ...
11:26:53.135 -> Number of channels: 8 Roll [µs]: 1498.96 Pitch [µs]: 1500.00    ...
11:26:53.181 -> Number of channels: 8 Roll [µs]: 1498.96 Pitch [µs]: 1496.99    ...
```

Project 8

Controlling your motors



Use PPM to control motor speed

As the name implies, an electronic speed controller is able to control the speed of your brushless motor. In this project, you will learn how to send commands to this controller and thus the motor through your microcontroller.

The electronic speed controller (ESC) is like your Teensy a microcontroller, but it has only one purpose: adapting the voltage going to the motor in such a way that the motor changes its speed. The speed at which a brushless motor turns depends on the supplied voltage, according to the kV constant. A motor rated at 4000 kV turns at 4000 rpm/V. A supplied voltage of 3V gives a turning rate of 4000 rpm/V x 3V=12 000 rpm. If you would directly connect your battery to the brushless motor, it would turn at a constant speed, as the voltage supplied by the battery does not change.

So how does the ESC adapts the voltage supplied to the brushless motor? By closing and opening the connection between the battery and the brushless motor, you can change the average voltage supplied to the motor. The longer the connection stays closed, the higher the supplied average voltage will be (with the maximum being the voltage supplied by the battery if the connection remains closed the whole time). You can control this average voltage through pulse-width modulation. This means that you need to sent a PWM signal from the Teensy to the electronic speed controller in order to control the brushless motor.

You will now control the first brushless motor through the physical circuit displayed on the left;

- Use some tape to make sure that the motors do not lift off.
- Re-use the same circuit with whom you already connected the receiver when testing the motors.
- Connect the three wires coming out of the other side of the ESC to the wires of the brushless motors using the wire terminal strip.
- Connect the power output of your ESC with the XT60 cables using another wire terminal strip. Be careful not to switch the polarity of the cables!
- Connect the white signal cable from the ESC to pin 1 of the Teensy.
- Connect the 5V and 0V of the ESC to 5V and GND of the Teensy respectively. Besides the power circuit going to the motor, your ESC has a second circuit that stays at 5V all the time and which enables you to power both the motors and your electronics with the same battery.

Only connect the battery once you uploaded the code to the Teensy and disconnected your USB cable from your computer (as a precautionary measure if something is wrong with the wiring).

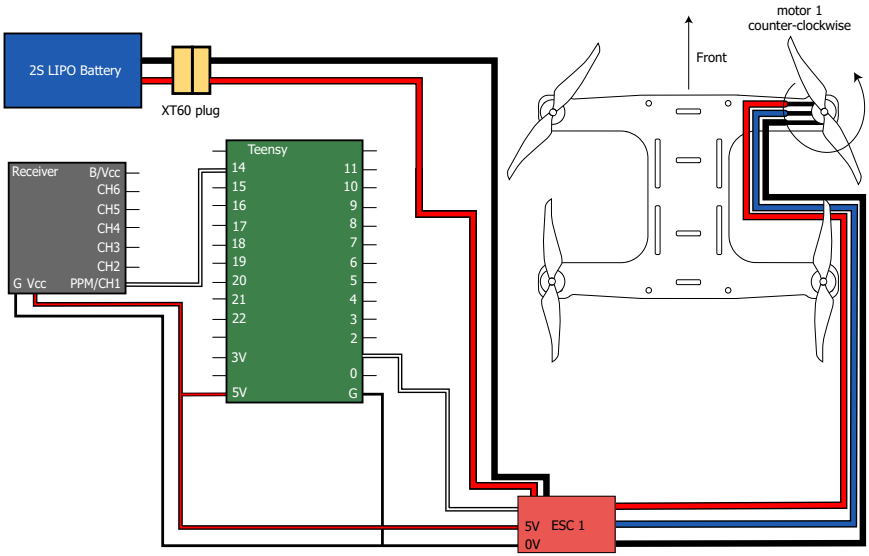
Make sure you use motor 1 (turning counter-clockwise) for this project. If you notice during testing that it spins clockwise instead of counter-clockwise, just switch two wires coming out of the motor with each other using the wire terminal strips.

In the code, you will control the brushless motor using a PWM signal sent from the Teensy to the ESC but you will use a PPM signal coming from the receiver to the Teensy. The throttle (so channel 3 or ReceiverValue[2] in Arduino language) will be used to set the speed of the brushless motor. The PWM values that you send to the ESC are the same as the PWM values that you get from the receiver: 1000 µs gives no throttle (motor does not turn), while 2000 µs gives full throttle.

Coding

Define the value for the throttle as a floating point number. The future value for this variable lies between 1000 and 2000 µs.

You will send the PWM signals from pin 1 of your Teensy to motor 1. Configuring pin 1 to send PWM signals can be done with the function analogWriteFrequency(pin number, PWM frequency). The PWM frequency used in most ESCs is equal to 250 Hz (=1/250=0.004 s=4000µs). This value can be found in the manual of your ESC.




```
1  #include <PulsePosition.h>
2  PulsePositionInput ReceiverInput(RISING);
3  float ReceiverValue[]={0, 0, 0, 0, 0, 0, 0, 0};
4  int ChannelNumber=0;
```

```
5  float InputThrottle;
```

Define the throttle variable

```
6  void read_receiver(void){
7          ChannelNumber = ReceiverInput.available();
8          if (ChannelNumber > 0) {
9          for (int i=1; i<=ChannelNumber;i++){
10                 ReceiverValue[i-1]=ReceiverInput.read(i);
11                 }
12         }
13 }
14 void setup() {
15         Serial.begin(57600);
16         pinMode(13, OUTPUT);
17         digitalWrite(13, HIGH);
18         ReceiverInput.begin(14);
```

```
19         analogWriteFrequency(1, 250);
```

Send PWM signals from your Teensy

By default, the resolution of PWM signals sent by the Teensy is 8 bit. This means that the signal ranges between 0 and 2^8-1=255. For our application, this would give a too coarse control, so you will change from an 8 bit to a 12 bit resolution; the PWM signal going from the Teensy to the ESC ranges between 0 and 2^{12}-1=4095. For a frequency of 250 Hz=4000 µs, 0 then corresponds with 0 µs and 4095 corresponds with 4000 µs. When you want to sent a PWM command in µs to the ESC, do not forget to multiply that value with 4095/4000=1.024.

SAFETY RELATED LINES: in order to avoid a motor start when you did not yet touch the radiotransmitter (for example when the throttle stick was not in the lowest position after your last flight), you add some additional lines before finishing the setup process. If the values sent from channel 3 (ReceiverValue[2] = the throttle channel) are bigger than 1050 µs or lower than 1020 µs, the code will not continue. When you move the throttle stick around its lowest value range (between 0.2 and 0.5%), the code continues and you can start the motor.

You already know that the throttle corresponds to channel 3 or ReceiverValue[2] from the radiotransmitter. The value of the throttle ranges from 1000 (no throttle) to 2000 µs (full throttle). You send this value to pin 1 and subsequently also the ESC and motor 1 through the analogWrite function. Remember to convert the throttle values in µs to their 12 bit equivalent by multiplying them with 1.024.

Testing

Once you uploaded the code successfully, disconnect the USB from the Teensy and connect the battery. Normally the LEDs on both the Teensy and the receiver should be illuminated as they receive power from the battery. Next, switch on your radiotransmitter (remember it should still be in PPM mode, not PWM); you should hear two times two beeps from your motor. Now slowly increase the throttle on your radiotransmitter to start the motor. Verify that the motor spins in the correct (counter-clockwise) direction.

Troubleshooting when the motor does not start:

- Verify that your receiver and Teensy get power by looking at their LEDs.
- Verify that your radiotransmitter is connected with your receiver (the LED on the receiver should not blink but should stay illuminated).
- Verify that the setting of your radiotransmitter is correct (in PPM mode).
- Verify the connections to the motor.

```
20        analogWriteResolution(12);
21        delay(250);
```

Set the PWM frequency

```
22        while (ReceiverValue[2] < 1020 ||
                  ReceiverValue[2] > 1050) {
23                  read_receiver();
24                  delay(4);
25                }
26  }
```

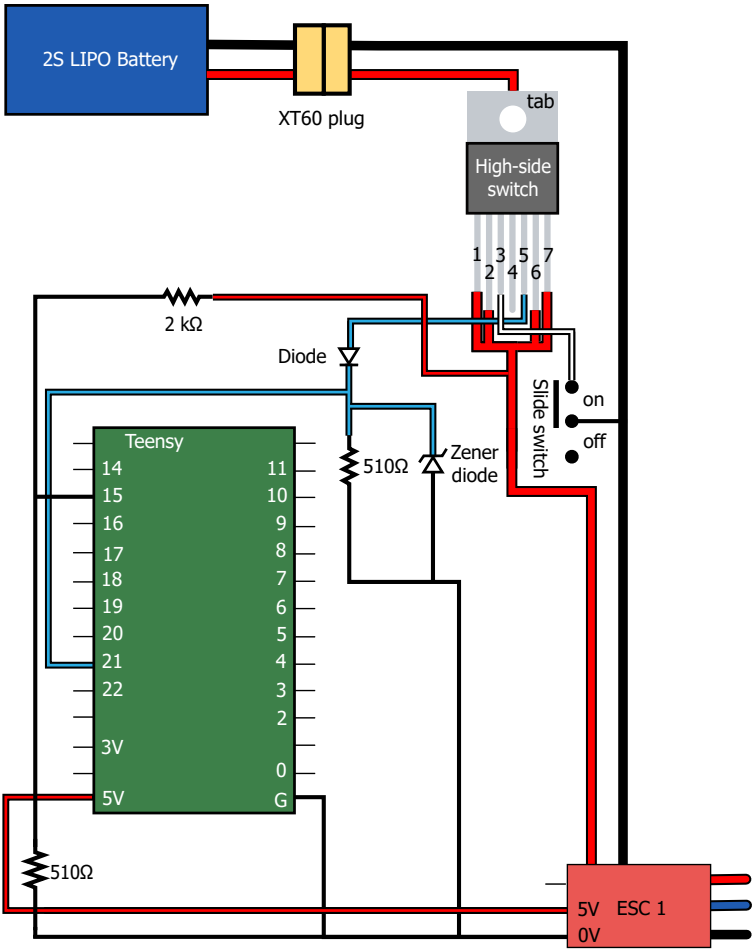
Avoid uncontrolled motor start

```
27  void loop() {
28        read_receiver();
29        InputThrottle=ReceiverValue[2];
30        analogWrite(1,1.024*InputThrottle);
31  }
```

Send the throttle input to the motor.

Project 9

Battery management



Monitor and protect your battery

Batteries store energy - a lot of energy. When this amount of energy is accidentally released in a short period of time, due to for example a short circuit, it can cause significant damage. Monitoring and closely protecting the battery is therefore very important.

A first obvious feature for each battery management system is **a switch**: you want to be able to turn off the power from the battery towards your motors at all time, even during full throttle. A standard (breadboard) slide switch can only switch off currents less than 1 A at 12 V; these type of switches are insufficient for our application: at full throttle, the current drawn by all four motors can easily surpass 20 A.

Additionally, you want some form of **battery protection** as well. When the battery is accidentally short-circuited, all the energy in the battery will be released nearly instantaneous. This causes the point with the highest resistance in the short-circuit to heat up and start burning; this can be a cable, a trace on the printed circuit board or the battery itself. The burning of the cable or the trace will cause the short circuit to disappear because the conductive band is broken, at the cost of destroying the printed circuit board. But when the battery itself has the highest resistance in the short circuit, it can burn and explode causing further damage.

A third feature is some form of **current measurement** to be able to monitor the battery level more closely - more about this will be explained further in this project. For your quadcopter application, you use a special transistor that integrates all of the above features: a high side power switch. This type of transistor is placed between the positive side of the battery and the load (hence the name high side). An Infineon BTS50055-1TMB or BTS50080-1TMB transistor will be used for this application because these switches are capable of conducting rather high load currents of respectively 70 and 37.5 A.

To be able to conduct these high loads, you need to use the tab of the high side switch, then connect output pins 1, 2, 6 and 7 with each other before making the connection with the load (which will be the ESCs in this case). You will connect pin 3 with a slide switch that is in turn connected with the high-current GND line; the transistor is switched on when a current flows between pin 3 and the GND. When the transistor is switched off, the voltage between pin 3 and the GND almost equals the battery voltage (maximally 8.4V for a 2S battery). The transistor has the capability to switch off the load current at full throttle, so you can instantaneously switch off your quadcopter at full power using the slide switch.

A downside of this high load current is the very high short circuit current limit: up to 180 A. This means that a short circuit in your printed circuit board can nonetheless cause serious damage!

Monitoring the current

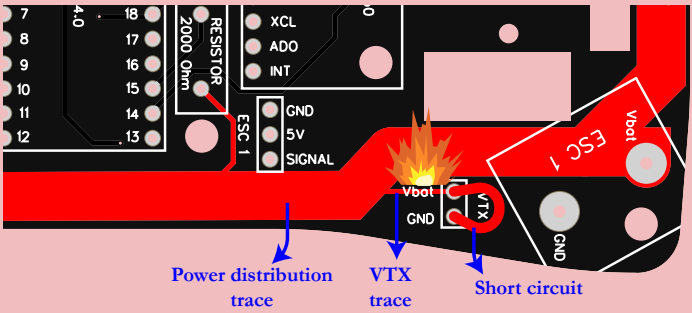
You already learned that measuring the voltage of your battery enables you to predict its remaining lifetime. Whenever you put the battery under heavy load, which will be the case when you throttle up the motors, the voltage will drop significantly without a real decrease in battery level, reducing your capability to accurately monitor the remaining battery level during flight. The only way to overcome this problem is to directly measure the current drawn from the battery.

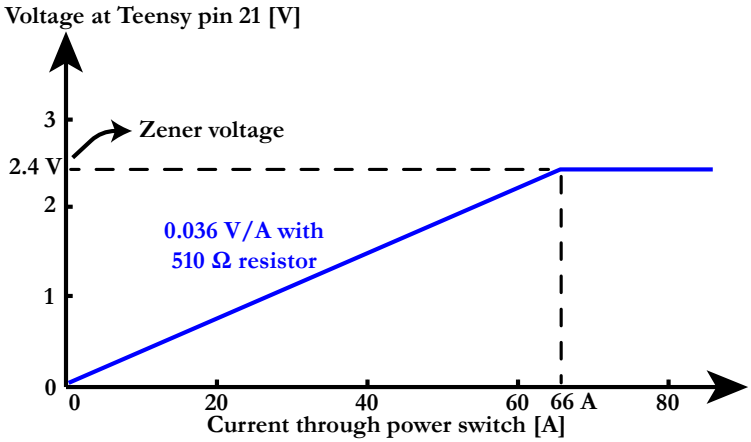
A special feature of your Infineon transistor is its current sensing capability: a current that is smaller but proportional to the load current flows from output pin 5 of the transistor. According the datasheet of the transistor, the current sense ratio is typically equal to 14 000. This means that a load current of 14 A through the transistor corresponds with a current of 1 mA from pin 5. With your Teensy, you cannot measure a current, only voltages. That is why you use a resistor to convert the current to a measurable voltage. Using Ohm's law, you know that a current of 1 mA through a resistor of 510 Ω will cause a voltage drop over the resistor of 1 mA x 510 Ω = 510 mV or 0.51 V. In other words, a current of 14 A through the transistor corresponds to a voltage drop over the resistor of 0.1 V. Assuming this remains proportional over the full measuring range gives a voltage to current ratio of 0.51 V/14 A or 0.036 V/A.

You will connect the 510 Ω resistor to pin 21 of your Teensy for voltage measurement. Before wiring everything up, you also need to consider the effect of a short circuit in the high current part of your current sensing circuit. In the event of a short circuit, the load current can easily exceed 180 A for a couple of milliseconds before the transistor switches off. Such a high current will lead to a high voltage over our resistor: 180 A x 0.036 V/A = 6.5 V. Knowing that the input pins of our Teensy are only 3.3V-tolerant, a voltage this high will probably destroy the microcontroller. To solve this issue, you will use a Zener diode. A Zener is a special diode that does not conduct any current below a certain fixed voltage, the "Zener voltage". For this project, you use a diode that has a Zener voltage of 2.4 V and needs a minimum current of 5 mA to start conducting at the right time. Adding the Zener diode in parallel to the resistor will cause the voltage over your Teensy to never exceed 2.4 V, protecting the microcontroller in the event of a short circuit on the load side. Of course the opposite can still happen: if the battery is connected in reverse and you have a short circuit, the Zener diode will not protect the Teensy because there is no "negative" Zener voltage limit. That's why you add a normal diode as well to prevent any negative voltage occurring over the Teensy.

Beware - you are not protected from all type of short circuits!

The installation of high side load switch does not prevent all types of short circuits and hence damage to your battery and/or quadcopter can still occur. Therefore, always make sure that you do not have any shorts in your circuit using your multimeter before connecting the battery. One example of a short circuit that the switch does not protect against is visualized in the figure below. Your quadcopter comes with a connection for a video transmitter (VTX), in case you want to add an FPV camera. This VTX connection is directly connected to the positive and negative power lines but it has a much smaller trace width than the power lines. The VTX traces are therefore not suited to accommodate high currents, and certainly not a short circuit current that can exceed 180 A before it is shut off by the high side load switch. This means that if you would try to create a short circuit by connecting the VTX Vbat and GND lines to each other and subsequently connect the battery, the traces on your power distribution board will start to burn and be destroyed.





Combining current and voltage for battery monitoring

It is not possible to test the current measurement on a solderless breadboard, since it requires currents too high for the traces on the breadboard. Instead, this part will explain the necessary coding and reasoning for accurate battery monitoring. All lines will be directly implemented in the flight controller and can be tested once the quadcopter is fully constructed.

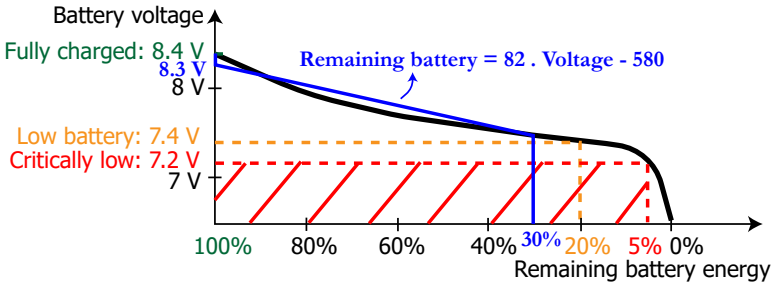
The variables necessary for accurately monitoring the battery capacity - besides the voltage and current - are the remaining battery capacity, the battery capacity at start, the current that you have consumed during flight and the default battery capacity. All battery capacity variables have the unit mAh; a battery of 1300 mAh can sustain a current of 1 A or 1000 mA during a period of 1300 mAh/1000 mA = 1.3 hours. The default battery used in this project has a capacity of 1300 mAh; initialize the BatteryDefault variable with this number.

The current can be read from the voltage of pin 21 using the function analogRead. Remember that the default resolution for analogRead is equal to **10** bit, so a voltage of 0 V gives you the digital number 0 and the maximal input voltage of 3.3 V gives the digital number 2^{10}-1=1023. Moreover you have built a system for which you have a voltage of 0.036 V for each Ampere flowing through the power switch. This means that de current flowing through the power switch is equal to the measured digital number at pin 21 divided by ((1023 / 3.3 V) x 0.036 V/A). Or equivalently, multiplying the measured digital number with 0.089.

When connecting the battery, you first need an indication of the actual battery capacity before you can calculate its evolution during flight. Luckily, the measurement of the battery voltage using our voltage divider and pin 15 is highly accurate if the motors are not started yet (and thus no voltage drop is present). In the setup phase, you hence determine the battery level using pin 15. There is only a (quasi) linear relation of the battery voltage to its capacity between 8.3 V and 7.5 V. If the voltage is higher than 8.3 V, you considered it to be at 100 % capacity (=1300 mAh) and you turn off the red LED. If the voltage lies below 7.5 V, you consider the battery to be at a capacity of 30% or less and the red LED stays illuminated. The following linear approximation between voltage and capacity is valid for the 1300 mAh - 2S battery:

$$\text{Remaining capacity } [\%] = 82 \cdot \text{Voltage} - 580$$

This approximation can be extracted experimentally with a sophisticated battery charger that indicates both the charged current and the actual voltage and subsequently plot it in a graph, like the one on top of the next page.




```
1  float Voltage, Current, BatteryRemaining, BatteryAtStart;
2  float CurrentConsumed=0;
3  float BatteryDefault=1300;
```

Define the battery monitoring variables

```
4  void battery_voltage(void) {
5          Voltage=(float)analogRead(15)/62;
6          Current=(float)analogRead(21)*0.089;
7  }
```

Measure the voltage and current of the circuit

```
8  void setup() {
9          digitalWrite(5, HIGH);
10         battery_voltage();
11         if (Voltage > 8.3) { digitalWrite(5, LOW);
12                 BatteryAtStart=BatteryDefault; }
13         else if (Voltage < 7.5) {
14                 BatteryAtStart=30/100*BatteryDefault ;}
15         else { digitalWrite(5, LOW);
16                 BatteryAtStart=(82*Voltage-580)/100*
17                 BatteryDefault; }
18 }
```

Determine the battery capacity prior to flight

During flight, you use the measured current to follow the evolution of your battery capacity. Since each iteration k in our main loop takes 0.004 seconds, the consumed current can be calculated with the formula:

$$\text{Current}_{\text{Consumed}}(k)[\text{mAh}] = \text{Current}_{\text{Measured}}(k)[\text{A}] \cdot \frac{1000\frac{\text{mA}}{\text{A}}}{3600\frac{\text{s}}{\text{h}}} \cdot 0.004\ \text{s} + \text{Current}_{\text{Consumed}}(k-1)[\text{mAh}]$$

The remaining capacity is subsequently calculated by subtracting the consumed current from the battery capacity at startup. When the battery capacity falls below 30%, illuminate the red LED.

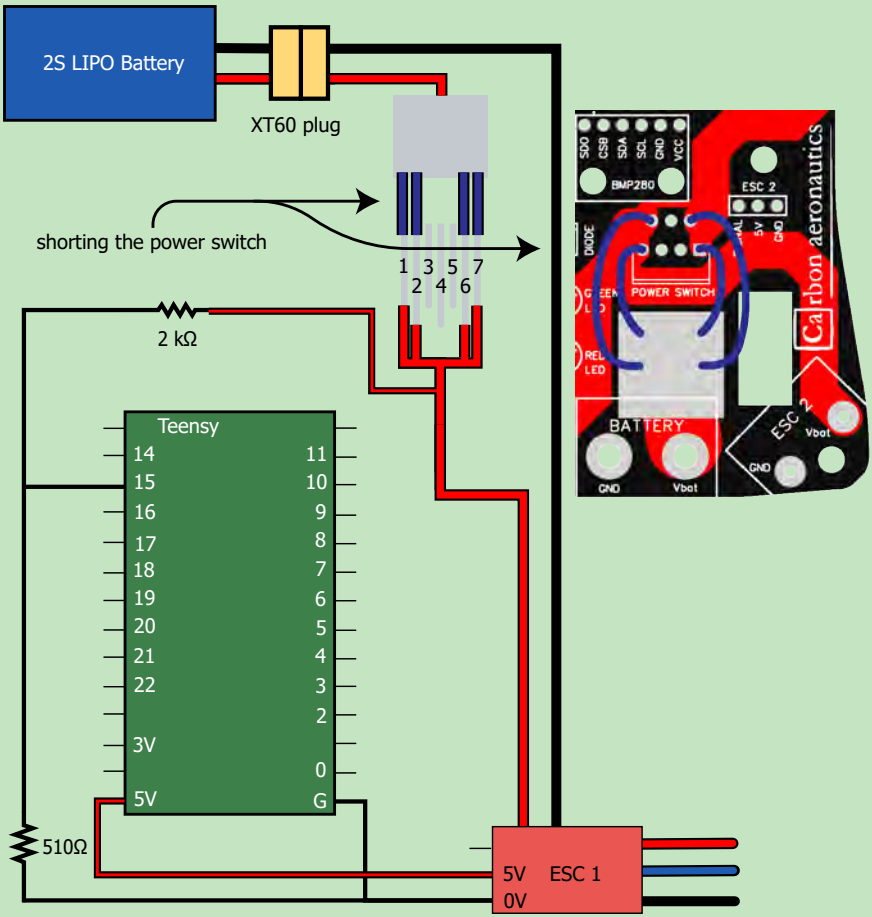
Operating the quadcopter without power switch

The high-side power switch is one of the more exotic components in this project and can be hard to come by. Most quadcopters made by hobbyist do not contain such a feature. Although it is not recommended, you can decide to exclude the power switch by shorting the battery tab on the printed circuit board, as illustrated on the figure to the right. Be aware that this removes any short-circuit protection and any control over switching on and off your quadcopter, other than physically connecting or removing the battery via the XT60 plug. The slide switch, (Zener) diodes and 510 Ω resistor are not necessary anymore because you will not have any current sensing capabilities. This means that a voltage measurement is the only way of monitoring the remaining battery energy. The code for this basic method is displayed below; once the voltage drops below 7.5 V the red LED is illuminated. Remember that when voltage drops during for example a sudden throttle increase, the measured voltage does not give an accurate reflection of the remaining battery energy.

```
1  float Voltage;
2  void battery_voltage(void) {
3          Voltage=(float)analogRead(15)/62;
4  }
5  void setup() {
6          if (Voltage > 7.5) digitalWrite(5, LOW);
7  }
8  void loop() {
9          battery_voltage();
10         if (Voltage < 7.5) digitalWrite(5, HIGH);
11         else digitalWrite(5, LOW);
12 }
```

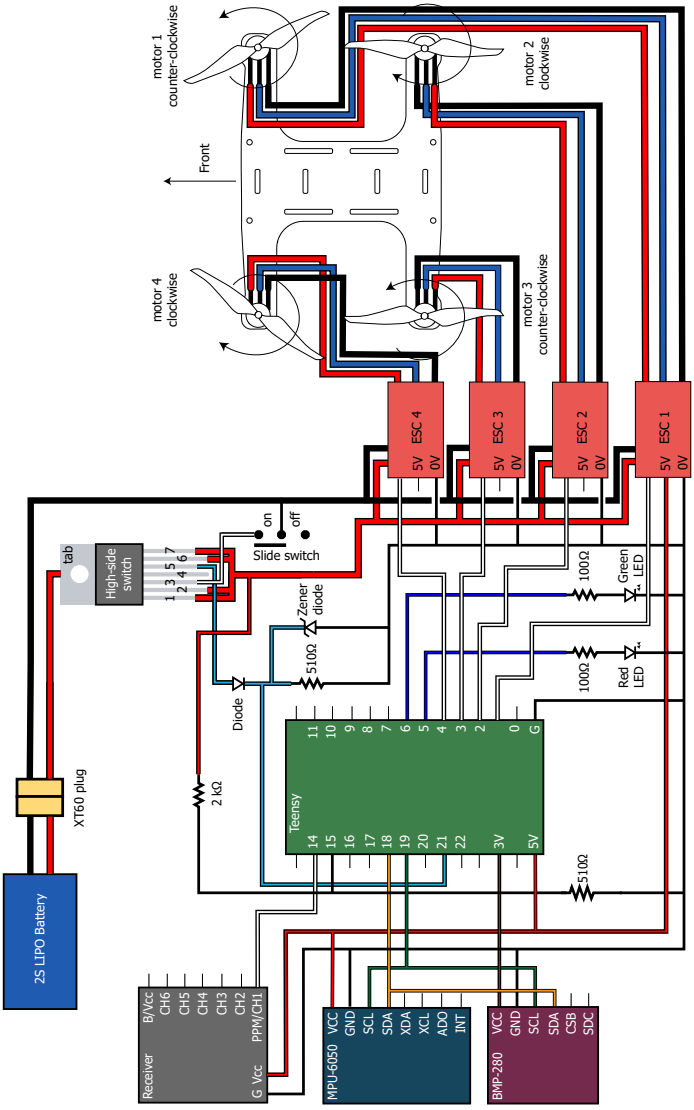
```
19 void loop() {
20       battery_voltage();
21       CurrentConsumed=Current*1000*0.004/3600+
22                              CurrentConsumed;
23       BatteryRemaining=(BatteryAtStart-
24                  CurrentConsumed)/BatteryDefault*100;
25       if (BatteryRemaining<=30) digitalWrite(5, HIGH);
26       else digitalWrite(5, LOW);
27 }
```

Calculate the battery capacity during flight



Project 10

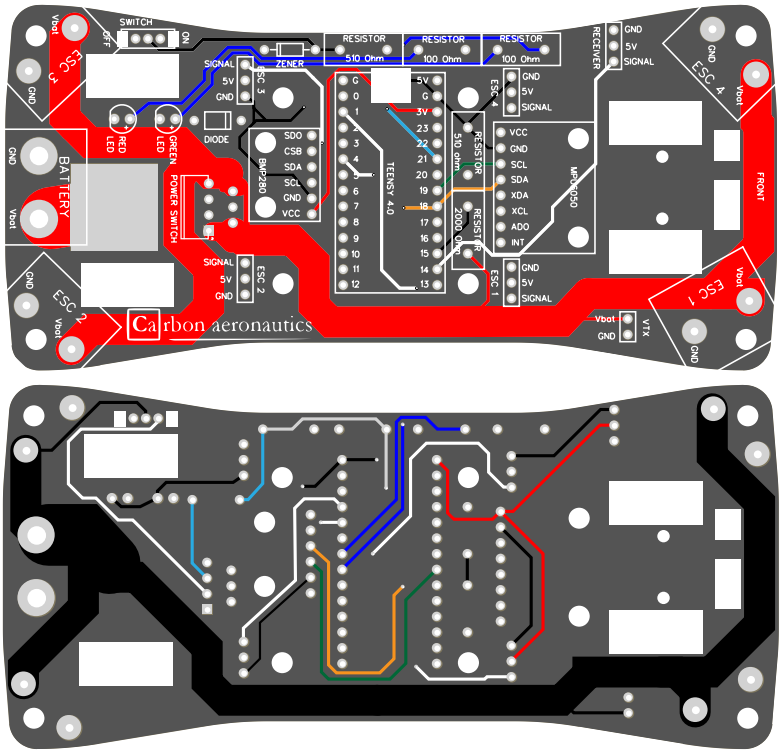
Assembling your quadcopter



Build your quadcopter

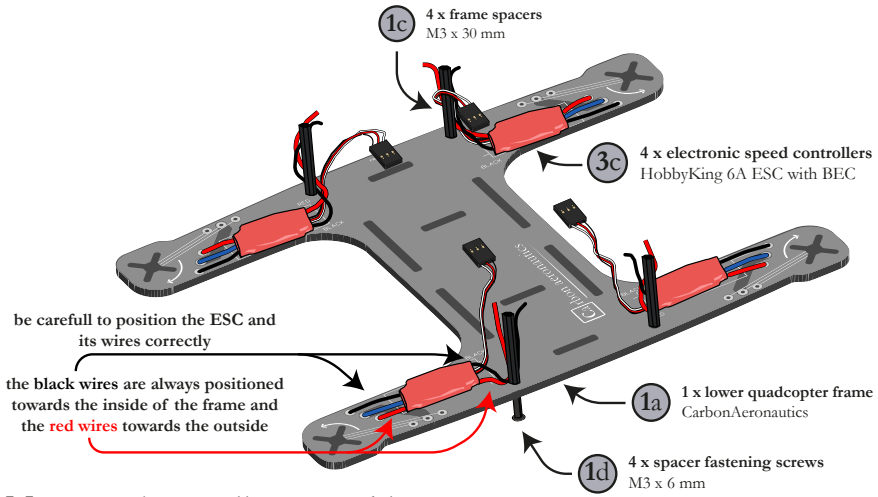
Now that you understand and tested all electronic components of your quadcopter, it is finally time to put it all together! The lower quadcopter frame will house all power electronics and the motors, while the upper frame mainly holds the control electronics and the power distribution.

The upper part of the quadcopter frame houses all the connections necessary for powering and controlling your quadcopter. In essence, it is a so called Printed Circuit Board or PCB: this is nothing more than alternating layers of insulating material and conductive copper. Inside your PCB, all necessary connections between the components are provided in the form of conductive traces. This means that most wires in the schematic of your quadcopter electronics on the left are already integrated in the upper quadcopter frame. The upper and bottom layer of your upper frame PCB are visualized below, together with the traces which are coloured similarly to the wires in the schematic. Notice that the power traces are much wider than the signal traces.

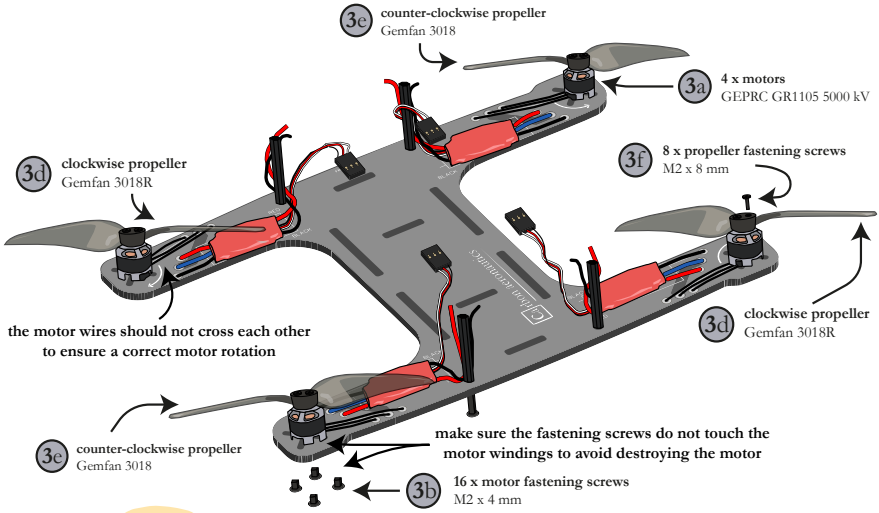


You are now ready to start the quadcopter assembly. During the first two steps, you position the four ESC and motors on the lower quadcopter frame and solder each of the three wires coming out of the ESC and motors to the frame. To ensure a correct rotation direction of the motors, make sure that the cables do not cross each other. The black and red wires coming out of the ESC should match the colours indicated on the frame itself.

ESC and spacer assembly

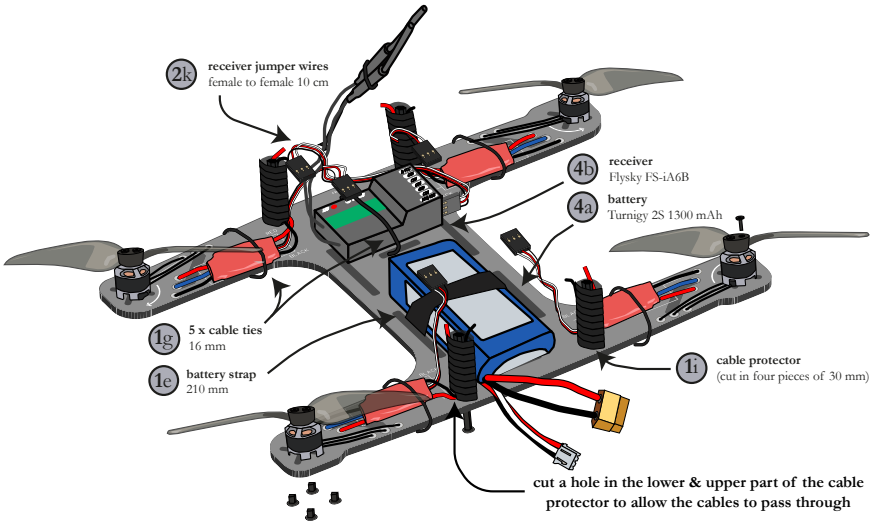


Motor and propeller assembly

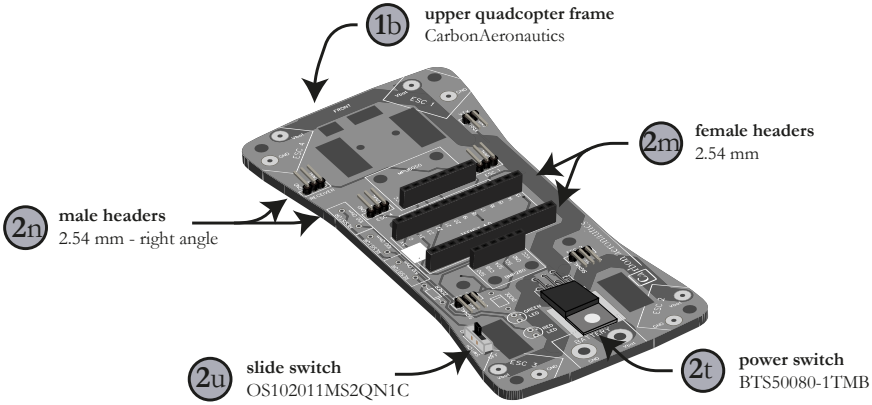


Use the lower frame to hold the receiver and the actual battery. Attach the receiver with a cable tie to the frame and provide a future connection for its signal and power to the microcontroller using a jumper wire. Attach the battery to the frame with a battery strap, making future battery replacement easy. Protect the cables coming from the ESCs using cable protectors, which you cut in pieces such that they cover the full length of the frame spacers. Cut some additional holes to allow the cables to go the ESC itself and to the upper frame. Once finished, start mounting the necessary headers and switches on the upper quadcopter frame.

Battery and receiver assembly

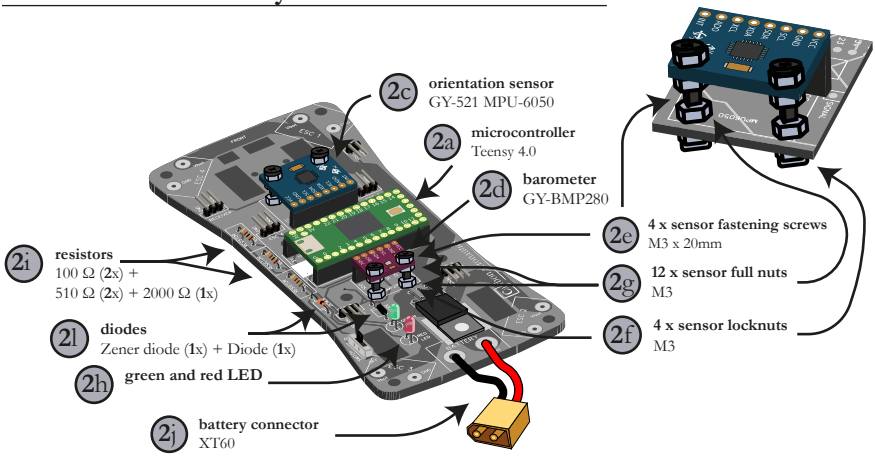


Headers and switches assembly

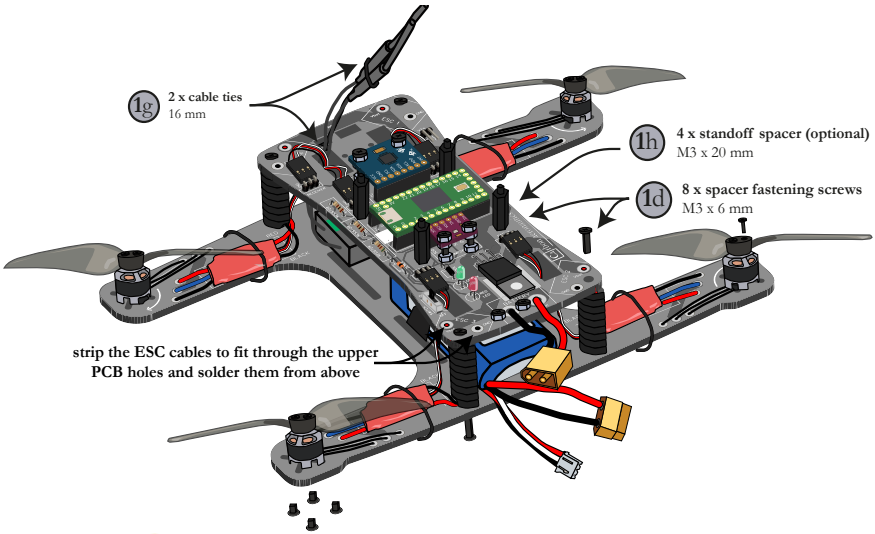


Now solder the additional electronics to the upper quadcopter frame: the resistors, diodes and battery connector. Assemble the orientation sensor and the barometer to the frame using fastening screws, full nuts and locknuts. The use of the barometer will be explained further in the project. Slide the microcontroller in the headers to finalize the upper frame. To connect the upper to the lower frame, first solder the ESC power cables to the upper frame then connect both frames using fastening screws to each other with the spacers.

Electronics assembly



Upper and lower frame assembly



Testing

Now that construction is finished, it is time to check the correct wiring and soldering of all components. **Do this before connecting the battery.** While this may sound boring, it is an essential step after assembling any complex product and will make troubleshooting in the next phases easier. A good test procedure would be to:

- Verify all connections using a multimeter and the schematic given at the beginning of this project.
- Verify that there are no short circuits between wires/pins that should not be connected with each other, also through the use of your multimeter. Pay extra attention to the absence of shorts between the red and black wire of the XT60 battery connector.
- Next, apply power to the prototype board using the USB port of the Teensy. Install the previous Arduino programs that you developed to illuminate the LEDs, read the gyro data and read the receiver data. Verify that they function correctly; this ensures a thorough check of your correct soldering and wiring.

When you are sure that there are no short circuits, connect the battery to continue testing:

- Measure the battery voltage by installing the correct Arduino program.

In the last step, you will test the motors and their correct rotation direction. Connect one ESC with channel 3 of the receiver, just like you did previously. Make sure you configure your radiotransmitter back to PWM instead of PPM. Turn on the radiotransmitter through the POWER button. Connect your battery with the XT60 plug. You should once again hear one beep from your radiotransmitter which indicates that it is connected with the receiver, and subsequently four beeps from the motor. Now slowly increase the throttle stick and verify that the motor turns in the required direction. Carry out the same test for all motors.

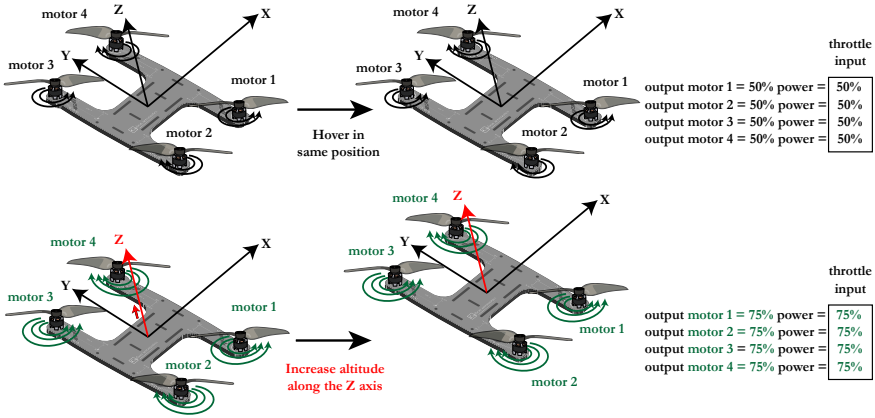
When (one of) the motors do not work, verify that:

- The battery is connected;
- The red LED on the receiver is illuminated continuously (a blinking led indicates that the transmitter is not connected, no led means no power to the ESC);
- The ESC is connected to channel 3 of the receiver;
- The transmitter setting is PWM instead of PPM.

If none of the above verifications solves the problem, verify that you soldered all respective wires correctly and resolder where necessary.

Project 11

Quadcopter dynamics



When you want to change the direction of your drone things become more tricky; assume you want the drone to stay at the same altitude but move sideways to the right (= roll around the X axis). The throttle input will be equal to 50% for all motors as you do not change altitude, but in order to initiate this sidewards movement the power output of motors 3 and 4 (=the left motors) should be higher than the power output of motors 1 and 2 (=the right motors). This means that you need an additional **roll input**, which will lower the power of motors 1 and 2 with for example 25% and at the same time increase the power of motors 3 and 4 with 25%. The same reasoning holds for the **pitch input** and the **yaw input**, but with other motor combinations as displayed in the figure on the right.

A nice property of this definition of throttle, roll, pitch and yaw input is that you can write all movements as a linear combination of each other, for all motor outputs:

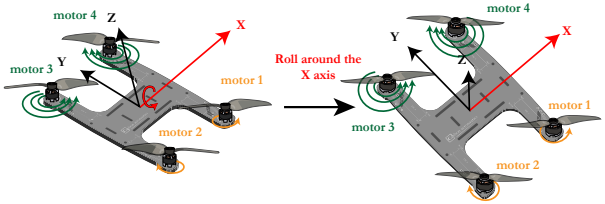
output motor 1 = throttle input - roll input - pitch input - yaw input
output motor 2 = throttle input - roll input + pitch input + yaw input
output motor 3 = throttle input + roll input + pitch input - yaw input
output motor 4 = throttle input + roll input - pitch input + yaw input

Learn how your quadcopter travels in space

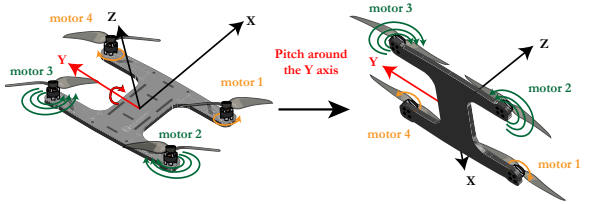
With all ingredients for a control system available and tested, it is time to learn how a quadcopter moves through space with your inputs given to the radio-transmitter.

You already know how to control the motors with your Teensy, how to measure the rotation rate with the gyroscope and how to receive and read commands with your radiotransmitter and receiver. These are all essential ingredients, but you still need to learn how they have to work together to be able to fly.

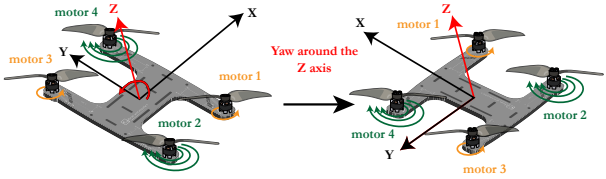
The first thing you need to understand is how you can use the four motors of your quadcopter to steer it in the directions you want. You do this by changing the power and thus rotation speed of the motors. Let's assume a perfect world for this project: no wind disturbances, instantaneous motor reaction and a uniform weight distribution. To let your quadcopter hover in the same position, each motor will have to work at around 50% of its power as shown on the left figure. To increase the altitude at which your drone is flying, you can simply increase the power of all four motors to for example 75%. To keep the quadcopter level, it is important that all motors should increase their power at the same time in order to keep the quadcopter level. The command to keep all motors at the same power level will be called the **throttle input**.



	throttle input	roll input
output motor 1 = 25% power =	50%	- 25%
output motor 2 = 25% power =	50%	- 25%
output motor 3 = 75% power =	50%	+ 25%
output motor 4 = 75% power =	50%	+ 25%



	throttle input	pitch input
output motor 1 = 25% power =	50%	- 25%
output motor 2 = 75% power =	50%	+ 25%
output motor 3 = 75% power =	50%	+ 25%
output motor 4 = 25% power =	50%	- 25%

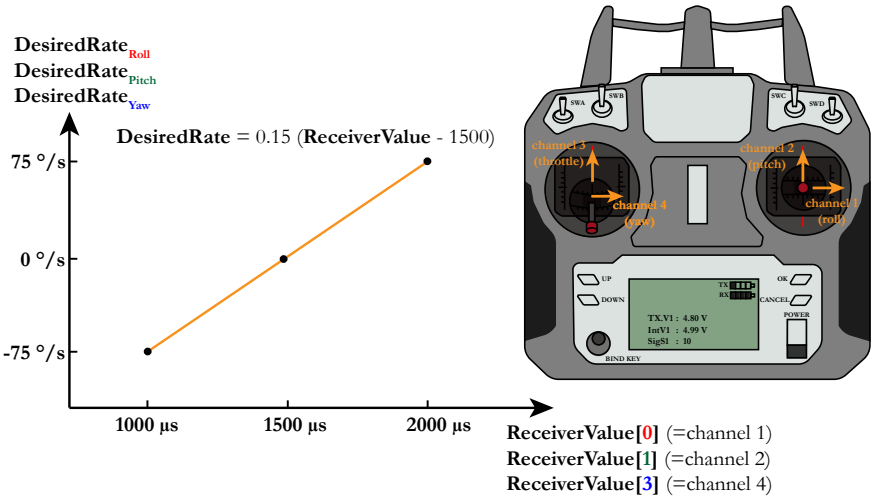


	throttle input	yaw input
output motor 1 = 25% power =	50%	- 25%
output motor 2 = 75% power =	50%	+ 25%
output motor 3 = 25% power =	50%	- 25%
output motor 4 = 75% power =	50%	+ 25%

In reality you do not send a power percentage to the motor but rather a PWM value between 1000 and 2000 µs, where 1000 µs corresponds with 0% motor output and 2000 µs with 100% motor output. In your code, the throttle input will vary between 1000 and 1800 µs to leave 20% motor output available at all times for rolling, pitching and yawing.

From receiver commands to desired rotation rates

The receiver sends commands to the microcontroller that vary between 1000 and 2000 µs according to the position of the radiotransmitter stick. For the throttle stick, this corresponds nicely to 0 and 100% power. For the roll, pitch and yaw sticks, whose default position is physically in the middle of the radiotransmitter at 1500 µs, you need to transform the PWM values to physical rotation rates. You can choose your maximal and minimal desired rotation rate; the higher the values the more agile your drone will be, but also the harder to control. For now, take the limit values of 75 °/s and -75 °/s. The transformation from the PWM values to the rotational rates is then visualised in the figure below together with the corresponding linear correlation.



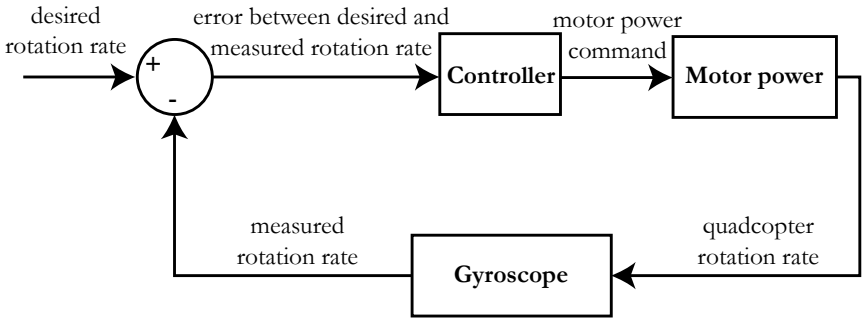
Transforming desired rotation rates to motor input?

You might think a second transformation is necessary: from the desired roll/pitch/yaw rotational rates to the motor roll/pitch/yaw input. This is true, but it is very difficult to measure which rotational rates correspond to a certain motor power level. Moreover, even if you obtain accurate data for this transformation, a huge problem remains; the reasoning in this project holds only for a quadcopter in a perfect world. In the real world however, even small disturbances will destabilize your quadcopter if you would introduce a fixed transformation.

For example, assume that the weight of your quadcopter is slightly higher in the back than in the front. This means that in order to hover, the back motors will need a slightly higher power output than the front motors. Any wind disturbance during the flight will have to be corrected immediately by varying the output of each motor accordingly. It is impossible to adjust for both imperfections using manual corrections within normal human reaction times; these phenomena need to be corrected by your fast microcontroller and a technique called PID feedback control.

Project 12

Quadcopter rate control



Now suppose that the controller just consists of the difference between the desired and the measured rotation rate, multiplied with a constant P:

$$Input_{motor} = P \cdot (DesiredRate - Rate)$$

By defining the error during each iteration k of the control loop:

$$Error(k) = DesiredRate(k) - Rate(k)$$

you can simplify the equation rewriting the motor input during iteration k as:

$$Input_{motor}(k) = P_{term}(k) = P \cdot Error(k)$$

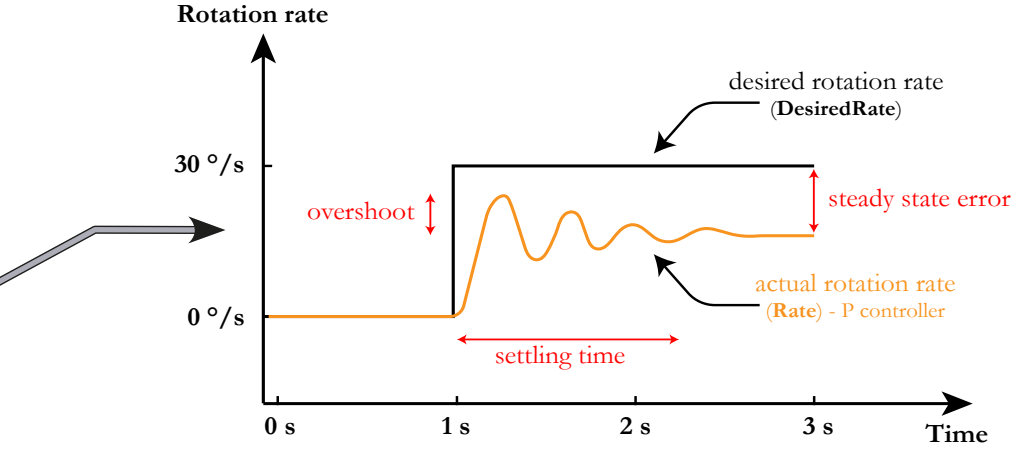
where P_{term} is the **proportional term** of the controller. The response of such a controller to a change in the desired rotation rate is visualised on the graph to the right: the higher you choose the value for P to be, the faster the actual rotation rate will approach the desired rotation rate and the smaller the settling time, which is a good thing. However, a larger P will also give a larger overshoot, meaning that the quadcopter might bounce violently when changing the desired rotation rate. Whatever the value of P there might also be a steady state error: the actual rotation rate never reaches the desired rotation rate. You overcome this issue by adding an **integral term**: this term will sum the past errors hence eliminating the steady state error.

Learn how to stabilize your quadcopter

With normal human reaction times, it is not possible to keep a quadcopter stable in the air. In this project you learn how to use a fast control loop to stabilize the quadcopter automatically while also taking into account the commands you give it with the radiotransmitter.

To stabilize your quadcopter, you need to use a very fast automated control loop that sends new commands to each of the four motors, multiple times per second. The control system that you will use for your quadcopter will be a 250 Hz system; this means that every 1/250 = 0.004 seconds, all four motors will receive new commands. These commands are generated depending on the commands you give yourself through the radiotransmitter, but are also generated automatically based on the actual rotation rate of the quadcopter in space, which is measured by your gyroscope.

The closed control loop that you will use for the roll, pitch and yaw rotation rates is displayed on the left. You use the gyroscope sensor to measure the actual rotation rate of the quadcopter and compare it to the desired rotation rate, which you have sent from the radiotransmitter. The error between both is transformed by the controller to a motor power command that is sent to each of the four motors. The resulting change in motor power changes the rotation rate of the quadcopter to a value that should be closer to the desired rotation rate than before. The actual rotation rate is measured once again and the process restarts. Each loop occurs every 0.004 seconds during flight.



The addition of the integral term can be implemented in the control equation through:

$$Input_{motor}(k) = P_{term}(k) + I_{term}(k) = P \cdot Error(k) + I \cdot \int_0^{k \cdot T_s} Error(t) \cdot dt$$

where the T_s is the length of one iteration, 0.004 s for our 250 Hz control loop. Discretization of the integral can easily be done through:

$$Input_{motor}(k) = P \cdot Error(k) + I_{term}(k-1) + I \cdot \frac{(Error(k) + Error(k-1)) \cdot T_s}{2}$$

The figure on the right shows the response of the Proportional-Integral (PI) controller; the steady state error disappeared but the system still has a large overshoot and a long settling time. A final improvement can be realized by adding a **derivative term**. Since a derivative along a function predicts its future value, this term will help to reduce the overshoot and hence the settling time:

$$Input_{motor}(k) = P_{term}(k) + I_{term}(k) + D_{term}(k)$$
$$Input_{motor}(k) = P_{term}(k) + I_{term}(k) + D \cdot \frac{d}{dt} Error(t)$$

the derivative will be discretized as well, giving the final discrete equation for a PID controller:

$$Input_{motor}(k) = P \cdot Error(k) + I_{term}(k-1) + I \cdot \frac{(Error(k) + Error(k-1)) \cdot T_s}{2} + D \cdot \frac{(Error(k) - Error(k-1))}{T_s}$$

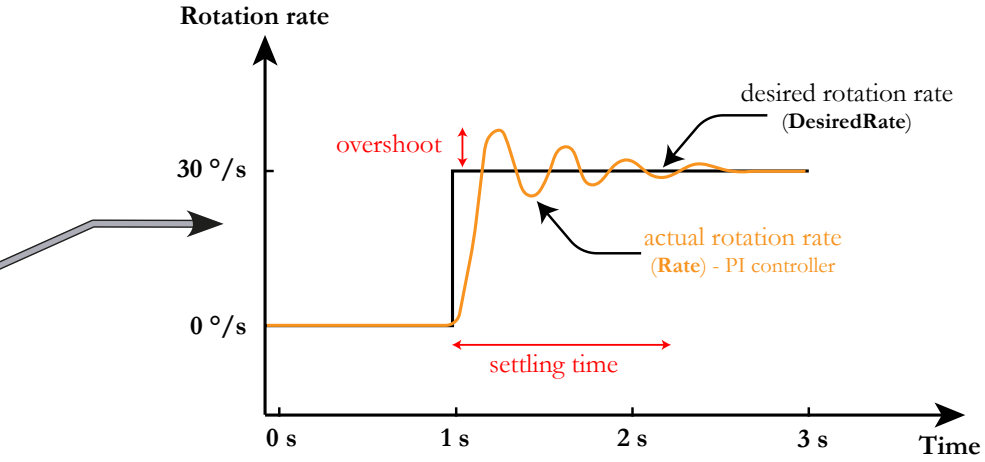
As the figure shows, the PID controller allows the quadcopter to approach the desired rotation rate quite fast with a small overshoot and settling time.

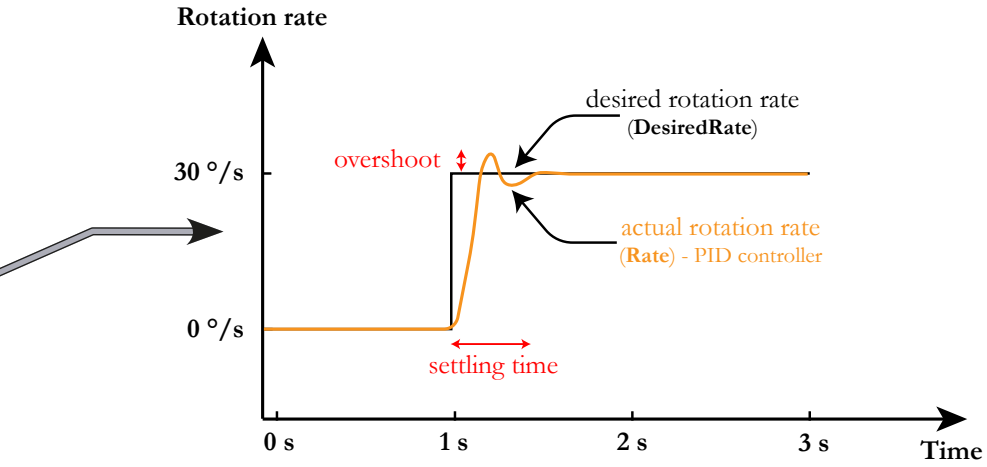
Obviously, the PID controller needs to be implemented for all three rotation rates; roll, pitch and yaw. For the roll rotation rate for example, the PID equation becomes:

$$Input_{Roll}(k) = P_{Roll} \cdot Error_{Roll}(k) + I_{term,Roll}(k-1) + I_{Roll} \cdot \frac{(Error_{Roll}(k) + Error_{Roll}(k-1)) \cdot T_s}{2} + D_{Roll} \cdot \frac{(Error_{Roll}(k) - Error_{Roll}(k-1))}{T_s}$$

which can be further simplified by saving the Iterm and Error during each iteration for the next iteration through the equations $PrevError_{Roll}$=$Error_{Roll}$(k-1) and $PrevIterm_{Roll}$=$Iterm_{Roll}$(k-1). This way, the iteration indexes k can be removed from the above equation:

$$Input_{Roll} = P_{Roll} \cdot Error_{Roll} + PrevIterm_{Roll} + I_{Roll} \cdot \frac{(Error_{Roll} + PrevError_{Roll}) \cdot T_s}{2} + D_{Roll} \cdot \frac{(Error_{Roll} - PrevError_{Roll})}{T_s}$$


Rotation rate
desired rotation rate
(DesiredRate)
overshoot
30 °/s
actual rotation rate
(Rate) - PI controller
0 °/s
settling time
0 s
1 s
2 s
3 s
Time


Rotation rate
desired rotation rate
(DesiredRate)
overshoot
30 °/s
actual rotation rate
(Rate) - PID controller
0 °/s
settling time
0 s
1 s
2 s
3 s
Time

The error for the roll rate is given by the difference between the desired roll rate and the measured roll rate by the gyroscope:

$$Error_{Roll} = DesiredRate_{Roll} - Rate_{Roll}$$

To make the notation easier, simplify the PID equation for the roll rate by saying that the motor input for the roll rate is function of the different parameters:

$$Input_{Roll} = f(Error_{Roll}, P_{Roll}, I_{Roll}, D_{Roll}, PrevError_{Roll}, PrevIterm_{Roll})$$

The above equations can be copied for the pitch and yaw rates in order to get the full PID controller.

In the right figure, the full control loop with equations is visualized. Start each loop with obtaining the commands from receiver, corresponding to the position of the sticks on your radiotransmitter. Transform these receiver values to the desired roll, pitch and yaw rates and the value for the throttle. Next, obtain the actual roll, pitch and yaw rates of the quadcopter from your gyroscope. These are subtracted from the desired rotation rates to obtain the error between both. Now you have the necessary information for the PID equations, which gives you the motor input values for roll, pitch and yaw. Introduce these input values in the motor power commands that were derived in project 11. With the first iteration now complete, wait until 0.004 seconds have passed to start the next iteration.

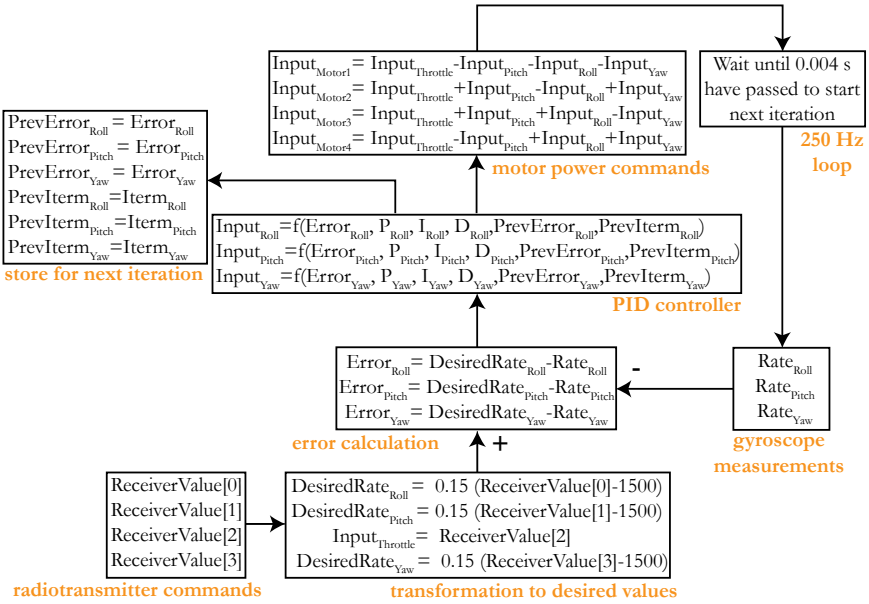
PID tuning

An important unknown that you did not yet determine are the values for P, I and D. These constants need to be chosen in such a way that their combination stabilizes the flight of your quadcopter. The following values are a good compromise between agility and stability for your quadcopter for all tested motor/ESC/propeller/battery combinations (see project 1):

- $P_{RateRoll} = P_{RatePitch} = 0.6$
- $I_{RateRoll} = I_{RatePitch} = 3.5$
- $D_{RateRoll} = D_{RatePitch} = 0.03$

Notice that the values for the roll and pitch rates are equal; this is evident since the quadcopter is (almost) symmetrical in both directions. For the yaw rates, the PID values are:

- $P_{RateYaw} = 2$
- $I_{RateYaw} = 12$
- $D_{RateYaw} = 0$



Finding these optimal values is not easy; there exist some basic methods to obtain them through calculations, but in the end you will always have to test and retest to find the values that work for your quadcopter. Usually the trail and error method is done by first choosing and testing a P value, then a value for I and finally also a value for D. The values can be chosen with the following guidelines:

- A high P value increases the responsiveness of your quadcopter, but a too high P value will cause your quadcopter to overcorrect and experience high frequency oscillations.
- A high I value stops unwanted drifting of your quadcopter, but a too high I value will cause your quadcopter to feel unresponsive.
- Finally the D value reduces the oscillations caused by the P value. Setting the D value too high causes motor vibrations.

It can be quite cumbersome to test different PID values, fortunately it only needs to be done once for each design.

Project 13

The flight controller: rate mode

```
1  #include <Wire.h>
2  float RatePitch, RateRoll, RateYaw;
3  float RateCalibrationPitch, RateCalibrationRoll,
4          RateCalibrationYaw;
5  int RateCalibrationNumber;
```

Define the gyro variables (projects 4 and 5)

```
6  #include <PulsePosition.h>
7  PulsePositionInput ReceiverInput(RISING);
8  float ReceiverValue[]={0, 0, 0, 0, 0, 0, 0, 0};
9  int ChannelNumber=0;
```

Define the receiver variables (project 7)

```
10 float Voltage, Current, BatteryRemaining, BatteryAtStart;
11 float CurrentConsumed=0;
12 float BatteryDefault=1300;
```

Define the battery variables (project 9)

```
13 uint32_t LoopTimer;
```

Define the parameter containing the length of each control loop

```
14 float DesiredRateRoll, DesiredRatePitch,
          DesiredRateYaw;
15 float ErrorRateRoll, ErrorRatePitch, ErrorRateYaw;
16 float InputRoll, InputThrottle, InputPitch, InputYaw;
17 float PrevErrorRateRoll, PrevErrorRatePitch,
          PrevErrorRateYaw;
18 float PrevItermRateRoll, PrevItermRatePitch,
          PrevItermRateYaw;
19 float PIDReturn[]={0, 0, 0};
20 float PRateRoll=0.6 ; float PRatePitch=PRateRoll;
          float PRateYaw=2;
21 float IRateRoll=3.5 ; float IRatePitch=IRateRoll;
          float IRateYaw=12;
22 float DRateRoll=0.03 ; float DRatePitch=DRateRoll;
          float DRateYaw=0;
```

All variables necessary for the PID control loop are declared in this part, including the values for the P, I and D parameters

Create your first flight controller

After a lot of hard work, you now have all the ingredients to create your first flight controller and test it on your quadcopter. Let's put all pieces together!

```
23 float MotorInput1, MotorInput2, MotorInput3,
       MotorInput4;
```

Declare the input variables to the motors

```
24 void battery_voltage(void) {
25       Voltage=(float)analogRead(15)/62;
26       Current=(float)analogRead(21)*0.089;
27 }
```

Battery function (projects 3 and 9)

```
28 void read_receiver(void){
29       ChannelNumber = ReceiverInput.available();
30       if (ChannelNumber > 0) {
31            for (int i=1; i<=ChannelNumber;i++){
32                 ReceiverValue[i-1]=ReceiverInput.read(i);
33            }
34       }
35 }
```

Receiver function (project 7)

```
36 void gyro_signals(void) {
37       Wire.beginTransmission(0x68);
38       Wire.write(0x1A);
39       Wire.write(0x05);
40       Wire.endTransmission();
41       Wire.beginTransmission(0x68);
42       Wire.write(0x1B);
43       Wire.write(0x08);
44       Wire.endTransmission();
45       Wire.beginTransmission(0x68);
46       Wire.write(0x43);
47       Wire.endTransmission();
48       Wire.requestFrom(0x68,6);
49       int16_t GyroX=Wire.read()<<8 | Wire.read();
50       int16_t GyroY=Wire.read()<<8 | Wire.read();
51       int16_t GyroZ=Wire.read()<<8 | Wire.read();
```

Gyro function (project 4)

Define a function that is called for each PID calculation of the roll, pitch and yaw rotation rates. You saw the equations for each term already in project 12. An important addition here is the limit of 400 µs on the I term; this term is used to avoid integral windup. Integral windup is a phenomena in which the integral term accumulates a significant error due saturation and causes a large overshoot. For example when the quadcopter cannot achieve the desired setpoint because it did not yet lift off the ground. Another limit is placed on the full output, to avoid a significant imbalance between to roll, pitch and yaw commands to the motor.

Return the values for the motor command, the error and the integral term from the PID equations to the main program.

To ensure a bumpless restart after landing your quadcopter, the PID error and integral values that are passed to the next iterations need to be reset once you land the quadcopter. This is also necessary to avoid any windup as well.

At the start of the setup phase, the red LED connected with pin 5 is illuminated to show that the microcontroller is still in the setup phase. As usual, the LED on the Teensy itself is illuminated as well to show that it receives power.

```
52        RateRoll=(float)GyroX/65.5;
53        RatePitch=(float)GyroY/65.5;
54        RateYaw=(float)GyroZ/65.5;
55  }
```

```
56  void pid_equation(float Error, float P , float I, float D,
        float PrevError, float PrevIterm) {
57        float Pterm=P*Error;
58        float Iterm=PrevIterm+I*(Error+
                    PrevError)*0.004/2;
59        if (Iterm > 400) Iterm=400;
60        else if (Iterm <-400) Iterm=-400;
61        float Dterm=D*(Error-PrevError)/0.004;
62        float PIDOutput= Pterm+Iterm+Dterm;
63        if (PIDOutput>400) PIDOutput=400;
64        else if (PIDOutput <-400) PIDOutput=-400;
```

PID function

```
65        PIDReturn[0]=PIDOutput;
66        PIDReturn[1]=Error;
67        PIDReturn[2]=Iterm;
68  }
```

Return the output from the PID function

```
69  void reset_pid(void) {
70        PrevErrorRateRoll=0; PrevErrorRatePitch=0;
          PrevErrorRateYaw=0;
71        PrevItermRateRoll=0; PrevItermRatePitch=0;
          PrevItermRateYaw=0;
72  }
```

PID reset function

```
73  void setup() {
74        pinMode(5, OUTPUT);
75        digitalWrite(5, HIGH);
76        pinMode(13, OUTPUT);
77        digitalWrite(13, HIGH);
```

Visualize the setup phase using the red LED

```
78        Wire.setClock(400000);
79        Wire.begin();
80        delay(250);
81        Wire.beginTransmission(0x68);
82        Wire.write(0x6B);
83        Wire.write(0x00);
84        Wire.endTransmission();
```

Communication with the gyroscope and calibration (project 4 and 5)

```
85      for (RateCalibrationNumber=0;
            RateCalibrationNumber<2000;
            RateCalibrationNumber ++) {
86              gyro_signals();
87              RateCalibrationRoll+=RateRoll;
88              RateCalibrationPitch+=RatePitch;
89              RateCalibrationYaw+=RateYaw;
90              delay(1);
91      }
```

When the time consuming part of the setup process is finished, illuminate the green LED to show that the quadcopter is able to start. However, only dim the red LED when the battery voltage is higher than 7.5 V.

SAFETY RELATED LINES: just before finishing the setup process, you need to check that the throttle stick is in its lowest position. Otherwise, if you accidentally left the throttle stick in a higher position and the radiotransmitter is not nearby, the motors could suddenly start after the setup process without you controlling it. With these lines, you stay in an infinite while loop until you move the throttle stick between 1020 and 1050 µs (so moving it from the lowest position to a slightly higher position).

In the last line of the setup process, start a timer that will count the time in the loop process and go to the next iteration after exactly 4000 µs or 0.004 s, to create a 1/0.004 s = 250 Hz control loop.

```
92      RateCalibrationRoll/=2000;
93      RateCalibrationPitch/=2000;
94      RateCalibrationYaw/=2000;

95      analogWriteFrequency(1, 250);
96      analogWriteFrequency(2, 250);
97      analogWriteFrequency(3, 250);
98      analogWriteFrequency(4, 250);
99      analogWriteResolution(12);

100     pinMode(6, OUTPUT);
101     digitalWrite(6, HIGH);
102     battery_voltage();
103     if (Voltage > 8.3) { digitalWrite(5, LOW);
104             BatteryAtStart=BatteryDefault; }
105     else if (Voltage < 7.5) {
106             BatteryAtStart=30/100*BatteryDefault ;}
107     else { digitalWrite(5, LOW);
108             BatteryAtStart=(82*Voltage-580)/100*
                BatteryDefault; }

109     ReceiverInput.begin(14);
110     while (ReceiverValue[2] < 1020 ||
            ReceiverValue[2] > 1050) {
111             read_receiver();
112             delay(4);
113     }

114     LoopTimer=micros();
115 }

116 void loop() {
117     gyro_signals();
118     RateRoll-=RateCalibrationRoll;
119     RatePitch-=RateCalibrationPitch;
120     RateYaw-=RateCalibrationYaw;

121     read_receiver();
```

Set the PWM frequency to 250 Hz and the resolution to 12 bit for all motors (project 8)

Show the end of the setup process and determine the initial battery voltage percentage (project 9)

Avoid accidental lift off after the setup process

Start the timer

Measure the rotation rates and subtract the calibration values (project 5)

Read the receiver commands (project 7)

Transform the commands from the receiver in µs to the desired roll, pitch and yaw rates in °/s as explained in project 11. The throttle command remains in µs.

Calculate the difference between the desired rotation rates and the measured rotation rates.

Start the PID calculations for each of the three rotation rates,. The outputs of these calculations are stored in the array PIDReturn; the roll/pitch/yaw input for the motors is stored in position 0, the error value and value for the Iterm that needs to be used for the next iteration is stored in positions 1 and 2. Retrieve these values each time for the corresponding rotation rate to be able to use them in the next iteration.

With the throttle stick, you are able to go to 2000 µs, which would give maximal power to all four motors. However, this would give no room to stabilize the roll, pitch and yaw rates. That is why you limit the throttle value to 1800 µs or 80%.

Now calculate the motor inputs with the quadcopter dynamics equations you saw during project 11. Remember to convert the throttle values in µs to their 12 bit equivalent by multiplying them with 1.024.

```
122 DesiredRateRoll=0.15*(ReceiverValue[0]-1500);
123 DesiredRatePitch=0.15*(ReceiverValue[1]-1500);
124 InputThrottle=ReceiverValue[2];
125 DesiredRateYaw=0.15*(ReceiverValue[3]-1500);
```

Calculate the desired roll, pitch and yaw rates

```
126 ErrorRateRoll=DesiredRateRoll-RateRoll;
127 ErrorRatePitch=DesiredRatePitch-RatePitch;
128 ErrorRateYaw=DesiredRateYaw-RateYaw;
```

Calculate the errors for the PID calculations

```
129 pid_equation(ErrorRateRoll, PRateRoll, IRateRoll,
        DRateRoll, PrevErrorRateRoll,
        PrevItermRateRoll);
130     InputRoll=PIDReturn[0];
        PrevErrorRateRoll=PIDReturn[1];
        PrevItermRateRoll=PIDReturn[2];
131 pid_equation(ErrorRatePitch, PRatePitch,
        IRatePitch, DRatePitch, PrevErrorRatePitch,
        PrevItermRatePitch);
132     InputPitch=PIDReturn[0];
        PrevErrorRatePitch=PIDReturn[1];
        PrevItermRatePitch=PIDReturn[2];
133 pid_equation(ErrorRateYaw, PRateYaw,
        IRateYaw, DRateYaw, PrevErrorRateYaw,
        PrevItermRateYaw);
134     InputYaw=PIDReturn[0];
        PrevErrorRateYaw=PIDReturn[1];
        PrevItermRateYaw=PIDReturn[2];
```

Execute the PID calculations

```
135 if (InputThrottle > 1800) InputThrottle = 1800;
```

Limit the throttle output

```
136 MotorInput1= 1.024*(InputThrottle-InputRoll
      -InputPitch-InputYaw);
137 MotorInput2= 1.024*(InputThrottle-InputRoll
      +InputPitch+InputYaw);
138 MotorInput3= 1.024*(InputThrottle+InputRoll
      +InputPitch-InputYaw);
139 MotorInput4= 1.024*(InputThrottle+InputRoll
      -InputPitch+InputYaw);
```

Use the quadcopter dynamics equations (project 11)

Make sure that the inputs to the motors do not exceed 2000 μs after the dynamic equations to avoid overloading them.

To avoid stopping the motors in mid-flight, keep them turning at 18% when the motor input decreases below 1180 μs (= the ThrottleIdle value).

SAFETY RELATED LINES: the previous lines would mean you can never switch off the motors, as they keep turning at minimally 18%. Just before sending the commands to the motors, you add the condition that if the throttle stick is brought to its lowest position (below 1050 μs), all four motors stop (e.g. the value of ThrottleCut-Off is 1000 μs or 0% power). Usually you would do this after landing the quadcopter. The PID parameters need to be reset in case you want to have a bumpless restart.

Now you are finally ready to sent the commands to each of the four motors.

The last step in the iteration is to wait until the 4000 μs or 0.004 s have passed using a while loop. When this condition is met, reset the timer to the actual time and the program can proceed to the next iteration. Congratulations, you created a 250 Hz control loop!

```
140     if (MotorInput1 > 2000)MotorInput1 = 1999;
141     if (MotorInput2 > 2000)MotorInput2 = 1999;
142     if (MotorInput3 > 2000)MotorInput3 = 1999;
143     if (MotorInput4 > 2000)MotorInput4 = 1999;
```

Limit the maximal power commands sent to the motors

```
144     int ThrottleIdle=1180;
145     if (MotorInput1 < ThrottleIdle) MotorInput1 =
146       ThrottleIdle;
147     if (MotorInput2 < ThrottleIdle) MotorInput2 =
148       ThrottleIdle;
149     if (MotorInput3 < ThrottleIdle) MotorInput3 =
150       ThrottleIdle;
151     if (MotorInput4 < ThrottleIdle) MotorInput4 =
152       ThrottleIdle;
```

Keep the quadcopter motors running at minimally 18% power during flight

```
153     int ThrottleCutOff=1000;
154     if (ReceiverValue[2]<1050) {
155             MotorInput1=ThrottleCutOff;
156             MotorInput2=ThrottleCutOff;
157             MotorInput3=ThrottleCutOff;
158             MotorInput4=ThrottleCutOff;
159             reset_pid();
160     }
161     analogWrite(1,MotorInput1);
162     analogWrite(2,MotorInput2);
163     analogWrite(3,MotorInput3);
164     analogWrite(4,MotorInput4);
```

Make sure you are able to turn off the motors

Sent the commands to the motors

```
165     battery_voltage();
166     CurrentConsumed=Current*1000*0.004/3600+
                    CurrentConsumed;
167     BatteryRemaining=(BatteryAtStart-
              CurrentConsumed)/BatteryDefault*100;
168     if (BatteryRemaining<=30) digitalWrite(5, HIGH);
169     else digitalWrite(5, LOW);
```

Keep track of battery level (project 9)

```
170     while (micros() - LoopTimer < 4000);
171     LoopTimer=micros();
172 }
```

Finish the 250 Hz control loop

Before you fly... radiotransmitter failsafe

Imagine you are flying your quadcopter and suddenly, your radiotransmitter loses power or signal. What will happen to your quadcopter? Well, you did not program any return to home function, so it will just keep flying until it runs out of battery or crashes. To avoid this, your radiotransmitter can tell the receiver to give a throttle command of 1000 µs when it loses connection. This means that the motors of your quadcopter will stop turning an it will fall to the ground - not the ideal solution but better to have some damage then a quadcopter on the loose.

Power on the transmitter → hold the OK button for two seconds → choose Sytem → go down and choose "RX setup" → go down and choose "Failsafe". Choose channel 3 (the throttle channel) then click UP or DOWN to activate the failsafe of channel 3 (ON). Now lower the throttle stick to its lowest position and press and hold the CANCEL function to tell the receiver it should sent a throttle command of 1000 µs or 0% when it loses contact with the radiotransmitter. Return to the previous screen on the transmitter, which should show -100% at channel 3. The failsafe is now set.

Start-up and flying your quadcopter

When you have set the radiotransmitter failsafe, it is time to start flying. After you connect the battery and turn on the slide switch, the red LED should be lighted indicating the ongoing startup process. Wait a couple of seconds without touching your quad (to avoid calibration errors). When the green LED illuminates, you can move the throttle stick slightly upward and each motor will beep four times indicating that you are ready to go. Increase the throttle stick to 30% power such that the motors are turning but the quad is not yet taking off, **then turn off the radiotransmitter to test the failsafe.** If after a second all motors turn off, the test is successful.

You can now start your first flight. Flying in rate control mode is rather difficult, so be sure to fly outside at a large grass field without any people nearby to minimize any damage to the quadcopter or others in the event of a crash. You can play with the PID values to optimize the quadcopters response to your liking.

This manual helps you to develop, program and construct your own quadcopter with the help of 24 small projects, explaining the essentials on aeronautics, electronics and embedded programming along the way.

All components and code used in this manual are fully hackable and adaptable, giving you the opportunity to create your own unique quadcopter.

Carbon aeronautics