We observed very soon into the project that the IR sensor does not give consistent readings above the floor of the Design Studio, where the craft is meant to operate, thus rendering this sensor useless for our application. We checked the sensor against other surfaces, to make sure it functions correctly, and we found that it indeed works accurately and consistently over other surfaces. We would have preferred the IR sensor over the ultrasonic one, because, being an analogue sensor, the reading time of it would have been only limited by the Arduino processing speed, which from testing was found to be around 2-3 ms, while the ultrasonic sensor has a much longer reading time, as sounds travels a lot slower than light and because we have to wait to make sure the previous sound have doesn’t echo back after a new one was emitted as that would give readings that are very far out from reality. This means that we must slow down the reading frequency of the sensor to 10Hz, to ensure enough time has passed before sending another sound pulse.

From the beginning, we decided not to use an Arduino UNO as our autopilot, but to go with a smaller, lighter version of this, which is the Arduino Nano. The boards are very similar in terms of their functionality, both using the same microprocessor. This allows us to code the boards in the very same way, from the same software. This modification presented us with a few other redesigns.

Firstly, we were not able to use anymore the GPS shield to store telemetry data from our aircraft. To solve this, we decided to use Bluetooth to transmit data to our laptops. One Bluetooth module is situated on our craft, which is encoding data containing: current altitude, throttle, autopilot state and grabber state, into a single string of characters, that can be sent all at once, and transmits them to another Bluetooth module, which is hooked up to another Arduino that decodes the data, before sending the data to a laptop, where it is store in an Excel file. the functions that encode and decode the data can be seen in the Figure 1 and Figure 2 in appendix 1.1. This way the data can be viewed in real time, and graphs that describe the behavior of the craft can be plotted with ease. Encoding and decoding the data into a single string of characters is the easiest way we have found to synchronize the transmission and the reception of the correct data points at any given time. Currently data is being transferred from the craft to the laptop at a frequency of 2Hz, which is enough for analysis, as the PID primary loop, which includes the PID loop that controls the autopilot, only runs at 10Hz. As a back-up to our Bluetooth solution, we are able to fit a SD card module onto the craft to store data onboard, which can be processed later.

Secondly, Arduino Nano’s are designed to be mounted on an either a copper board or on a breadboard. We chose a copper board as the circuit can be minimalized, both in terms of size and in terms of weight, to save space on the craft. The circuit includes all connections required with the receiver, flight controller, ultrasonic sensor and Bluetooth module. It also includes a relay, whose function is to switch between autopilot and manual flight. A relay was chosen as it provides a physical connection between the receiver’s throttle channel and the flight controller, meaning that if the Arduino fails, in any way, or if the power to the Arduino is lost, then control of the craft can still be maintained as the relay falls back into it’s normal state, which is designed to be manual mode. At the same time, if the Arduino losses signal from the receiver, the relay falls into the same position, providing manual control of all channels. Although a transistor was considered to switch the relay, the datasheet of the part indicated that the coil only draws 30 mA at 5V, which is the Arduino voltage, while the I/O pins of the board can provide up to 35 mA, indicating that the relay can be powered directly from the Arduino, thus saving more space and simplifying the circuit more. In retrospective, a fly-back diode should have been added to cope with the surge in voltage that occurs when the relay is powered off. An LED is also present on the board that lights up to confirm Bluetooth connection between the craft and the other Arduino. A schematic of the circuit can be seen in Figure 4, together with a copper board layout.

For the firmware of the flight computer, we started by using Cleanflight 2.1.0, which was the latest version at that time, but after having some issues with the signal on one of the channels of the flight controller, we decided to flash it again, this time with another version 2.2.0, which was released meanwhile. The test flight was successful, we managed to see the aircraft working, but the telemetry that was recorded by the flight controller, was incompatible with any software that could have helped us to interpret it. The telemetry data is very important to us, as the autopilot needs some values for the throttle, which it can slightly adjust to maintain a constant altitude. In the end, we changed the firmware from Cleanflight 2.2.0 to Betaflight 3.5.2, which is a firmware that allows more advanced control of the craft’s parameters but allows us to record and read the telemetry to get the values we need for the throttle.

For altitude reading, we intended to use an averaging algorithm, that would make several readings from the sensor and calculate the average of those to compute the altitude. Given the long reading time required by the ultrasonic sensor, this was not possible. Researching data filtering algorithms, we discovered an algorithm, called exponential smoothing [1]. The advantage of using this algorithm is that it only requires one parameter to be changed to adjust the amount of filtering and it only stores one additional altitude reading, making it very efficient for the Arduino. Implementing it in the code is not challenging as it only has one equation that needs implementing:

Figure 3 presents the algorithm used to smooth the altitude. On line 56, the actual smoothing function is seen. “W” is the weight of the new data coming in from the sensor and (1-W) is the weight of the old data.

Graph 1 High weight

The amount of smoothing can easily be adjusted by changing the W parameter. Line 59 in figure 3 adds a fail-safe to the algorithm, because if the sensor times out, i.e. the sound pulse doesn’t echo back in the given time for example of the aircraft is too high, the reading on the altitude would be equal to 0, which would make the autopilot think that the craft is very low to the ground, thus applying more throttle. This could cause the craft loose control, and constantly apply more and more throttle. This way, if the sensor goes out of range, it publishes the previous altitude reading, which would make the autopilot apply the correct level of throttle. The only way the sensor would not echo back, would be if the altitude is too high. In our case, the sensor is set to time out at an altitude of about 150 cm, which is well over the 80 cm altitude we are required to hover. Through data smoothing we aim to eliminate the spikes that can occur due to imperfect readings but to not introduce a long settling time. A high weight on the data smoothing, doesn’t introduce a long settling time, but it doesn’t eliminate the jagged appearance of the altitude, as can be seen in Graph 1.

Graph 2 Low weight

Testing with a low sensor weight, the jagged aspect of the readings disappeared, but a long settling time was introduced, which made us loose data regarding the peak altitude as it can be seen in Graph 2. Testing has been performed on a different set-up than the one on the aircraft, as the only interested was the smoothing function. The set-up was moved by hand at different speeds, but all weights have been tested over a range of movement speeds, but only the relevant speeds for our decision have been presented. Our decision towards the weight has been made bearing in mind that we want to eliminate the very large spikes created by wrong readings and we want to eliminate as much as possible long settling times, so a weight that favours new data, of W = 33, is being used.

The algorithm that decides when to drop the beanbag implements a voting system, that is constantly casting votes for or against dropping the beanbag (Figure 4). If the number of votes for dropping the beanbag exceed a threshold, a function is called to command the servo to open the grabber. If a set number of votes have been casted and the number of votes for is not enough to command the servo, then the voting system is reset to 0 and the process begins again. With every PID loop, the algorithm checks the altitude and votes if the aircraft is at the required altitude or not. This way, the beanbag will not be dropped if the aircraft passes through the required flight level and keeps climbing or descending. Based on the number the votes required, the craft needs to be hovering in range of 5 cm of the target altitude for 3 out 5 seconds, before the dropper is engaged.

The autopilot uses a standard PID controller (Figure 5) which takes in as an initial guess, the throttle value required to hover at the target altitude that was used under manual flight and it adjusts around this value. The output of the PID is a number between 0 and 999, 0 representing 0 throttle and 999 representing maximum throttle. This number range was selected to help simplify the encoding and decoding functions, as this way, characters 3 through 6 will always be the throttle value in the encoded string. This number is then remapped into the range 1000-2000 to be then sent as a pulse width to the flight controller, where it is interpreted as a throttle value.

The I part of the PID controller uses an anti-windup algorithm to ensure that the error doesn’t oscillate around the target value. Anti-windup uses both clamping (line 106-107, Figure 5), which only triggers the integral part when the error is relatively small, thus the value of the error doesn’t vary at a high rate and it uses boundaries to limit the magnitude of the error integral (line 103, Figure 5). Boundaries are also used on the throttle value as it cannot exceed the maximum pulse length of 2000, but the value is capped to less than to limit the climbing rate of the craft as a high climbing rate can be dangerous. The base values for the maximum climbing rate and the base throttle value are changed based on the grabber position, as the algorithm assumes that of the grabber is closed, then the mass of the aircraft is greater as it has the beanbag attached. The code stores two sets of values for the base and maximum throttle and those sets are chosen based on the grabber position (line 94-96, Figure 5). Those values are determined experimentally, under manual flight, from the telemetry that is recorded by the flight controller.

The function that turns on or off the autopilot is enabled as in interrupt in the code(Figure 6). This way, even if the code is stuck in an infinite loop, due to a bug, manual control of the craft can still be achieved. The receiver is sending pulses with lengths varying between 1000 and 2000 µs every 20 ms, the length of the pulse depending on the position of the switch on the transmitter. The function is called every time the pin that is connected to the receiver changes state. If the pin is HIGH, then it starts a timer, as it means that the pulse just started. The function is then called again, when the pins falls back to low and the timer stops. A pulse length of over 1500 µs commands the Arduino to switch the relay off, thus enabling manual control of the craft, while a pulse length of under 1500 µs commands the Arduino to power the relay, thus enabling the autopilot. The autopilot is constantly calculating the throttle based on the altitude, but that signal cannot physically be sent to the flight controller, as the Arduino is not in electrical contact with it, except for when the autopilot is switched on.

The code is written as a class (Figure 7) which makes the loop of the Arduino look very intuitive, as only the name of the functions is called, so troubleshooting is very easy to be performed. All constants are stored in a different file (PID constants, pins, throttle values), so changing them can be done very quickly and efficiently (Figure 8).

Appendix 1.1

Code

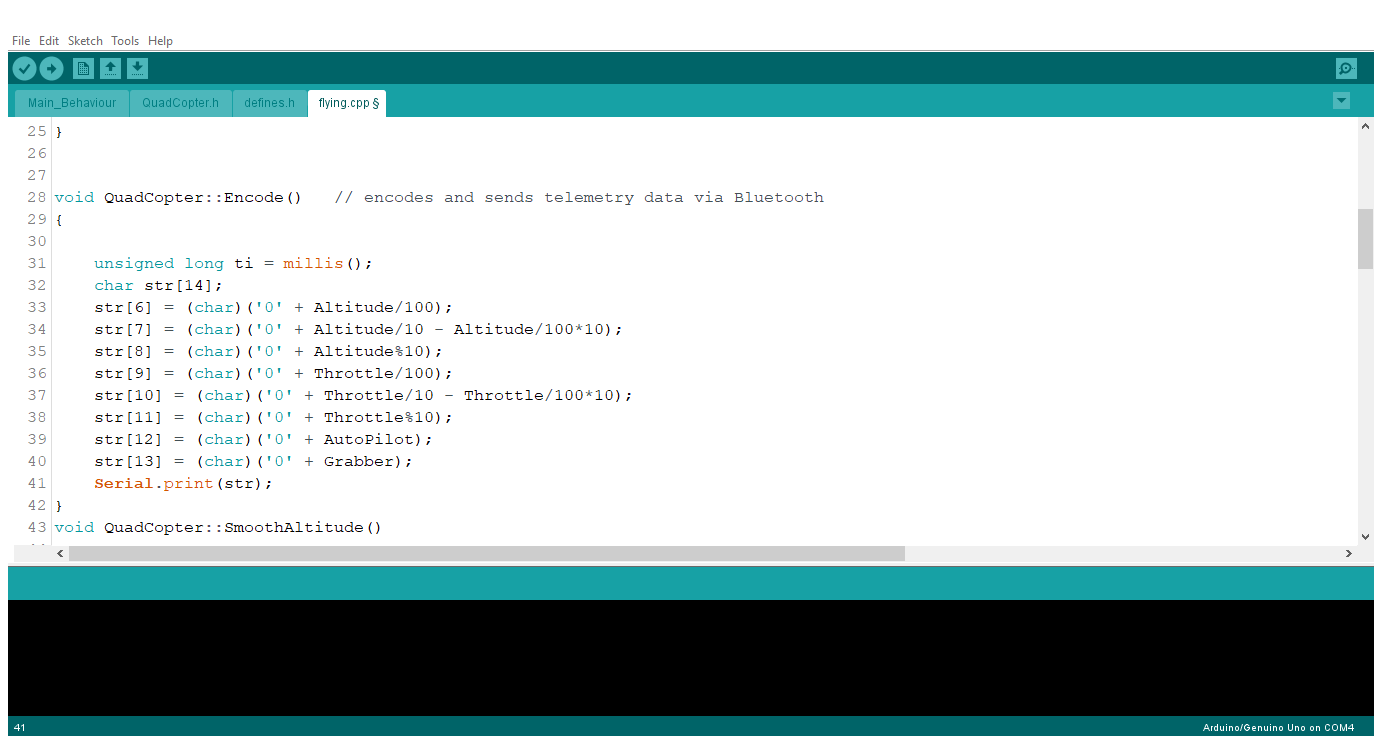


Figure 1 Data encoding function

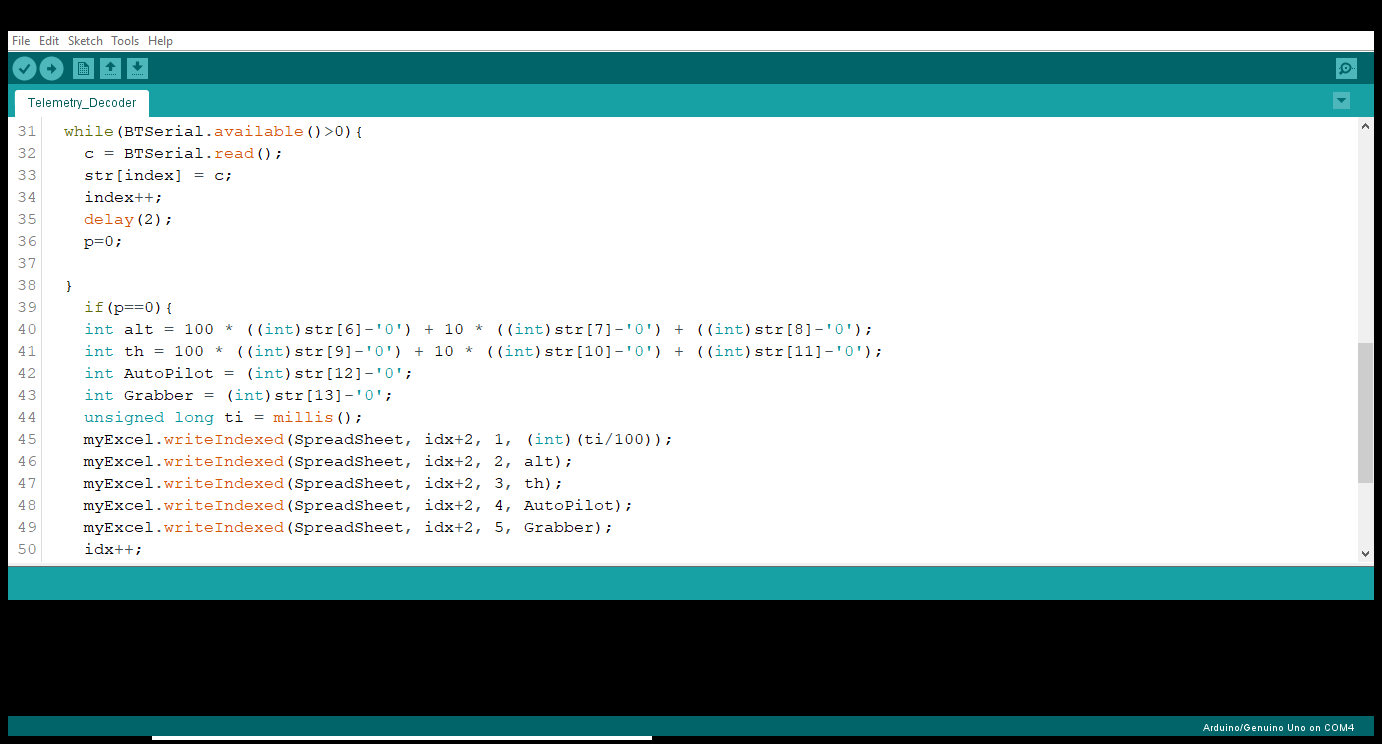


Figure 2 Data decoding function



Figure 3 Altitude smoothing algorithm



Figure 4 Voting algorithm

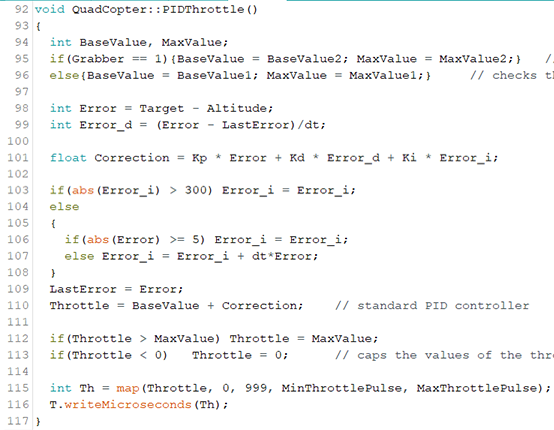


Figure 5 PID controller

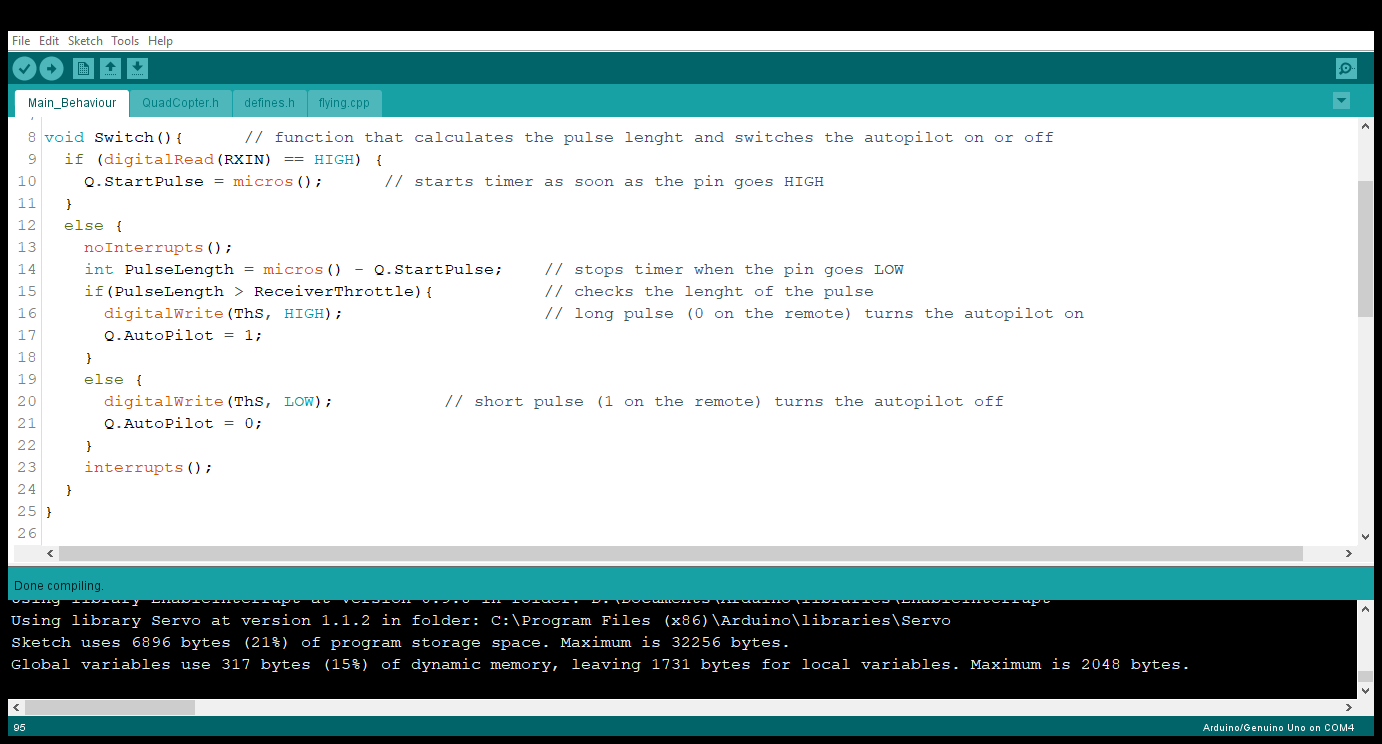


Figure 6 Autopilot Switch

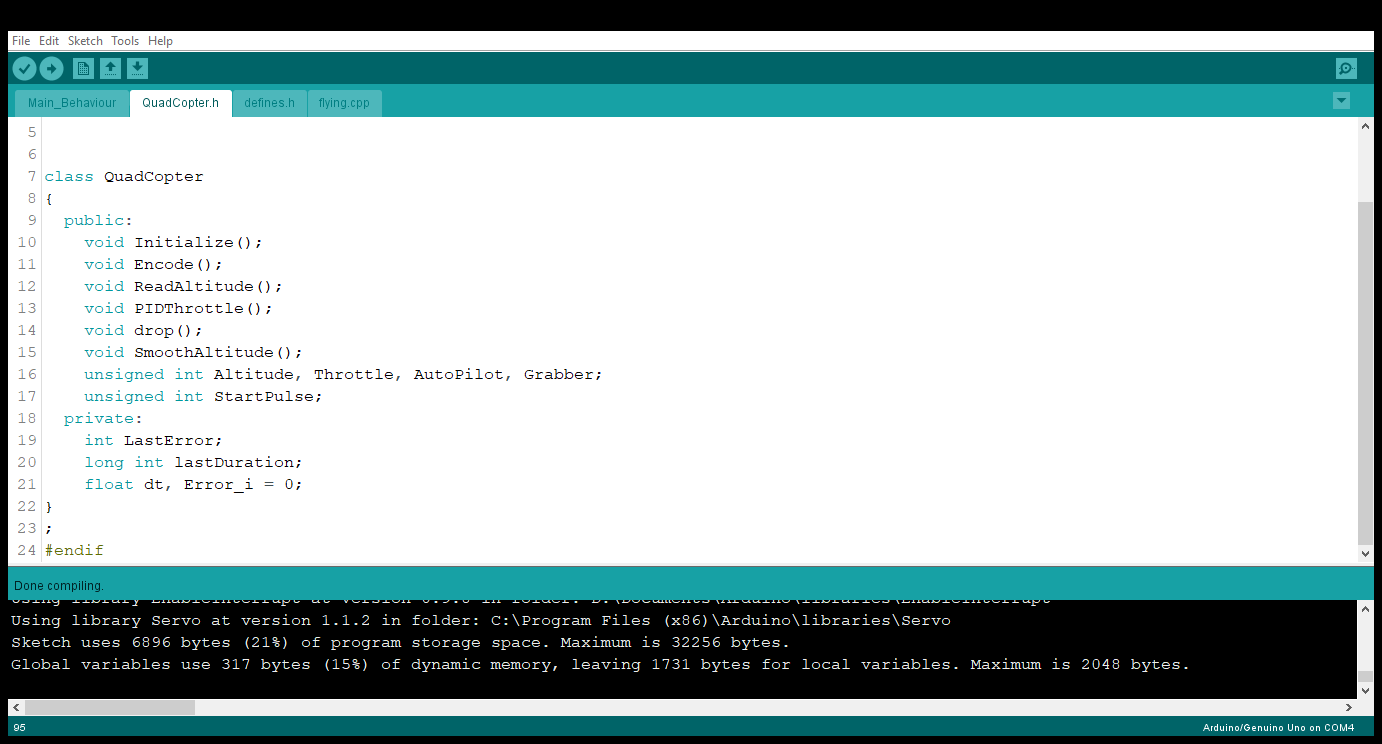
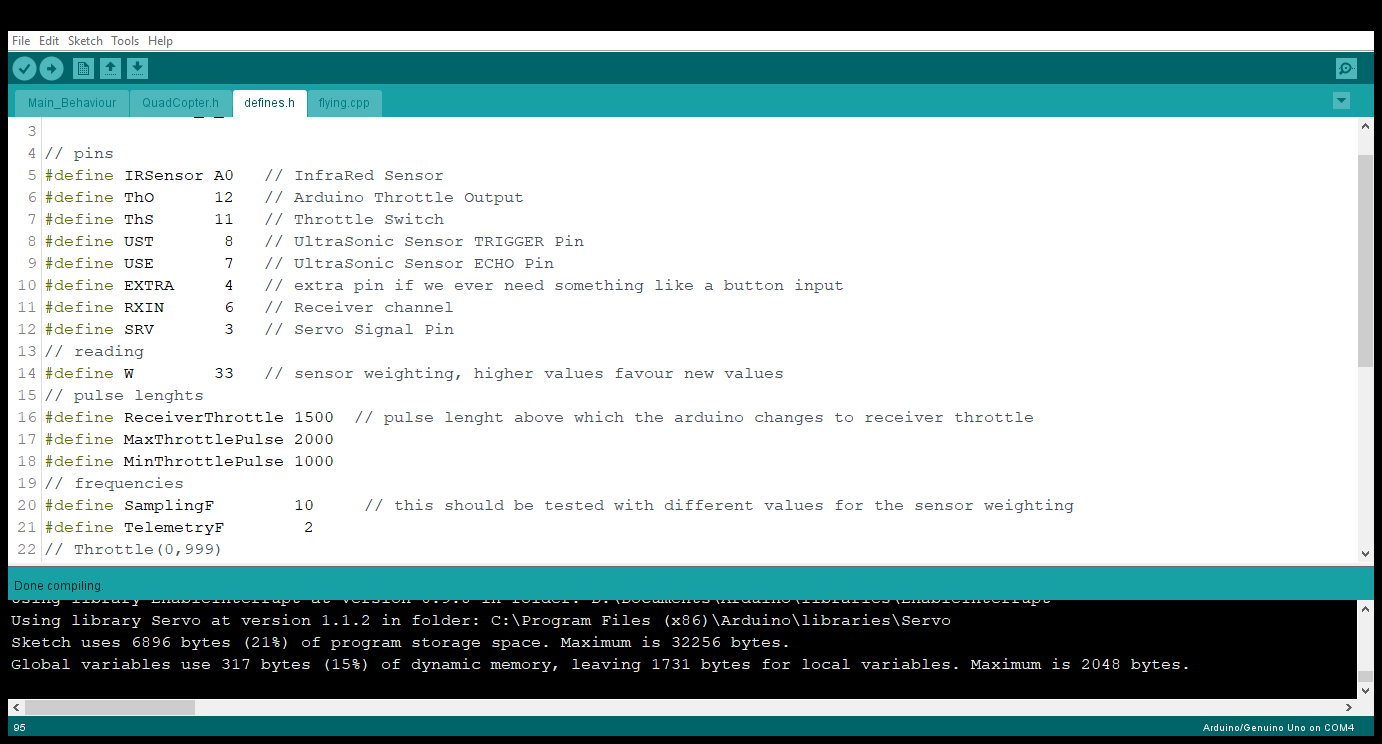
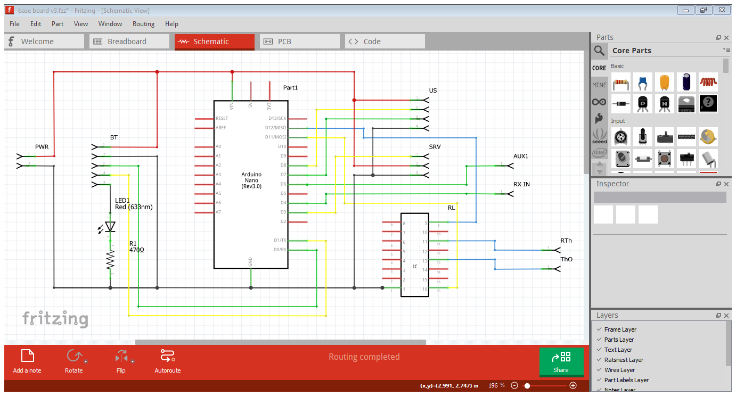


Figure 7 Class definition

Figure 8 Constants

Appendix 1.2

Electronics

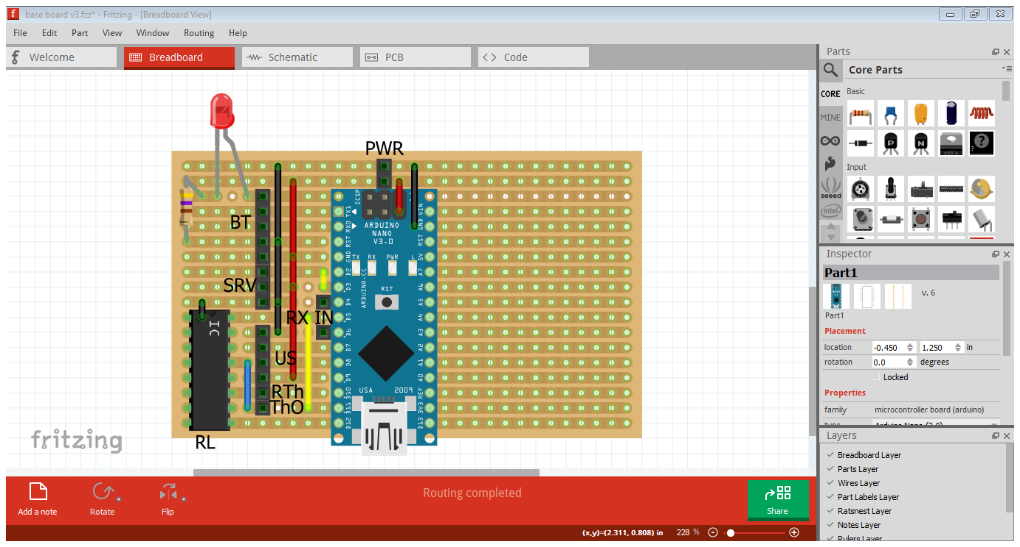


Figure 9 Electronics