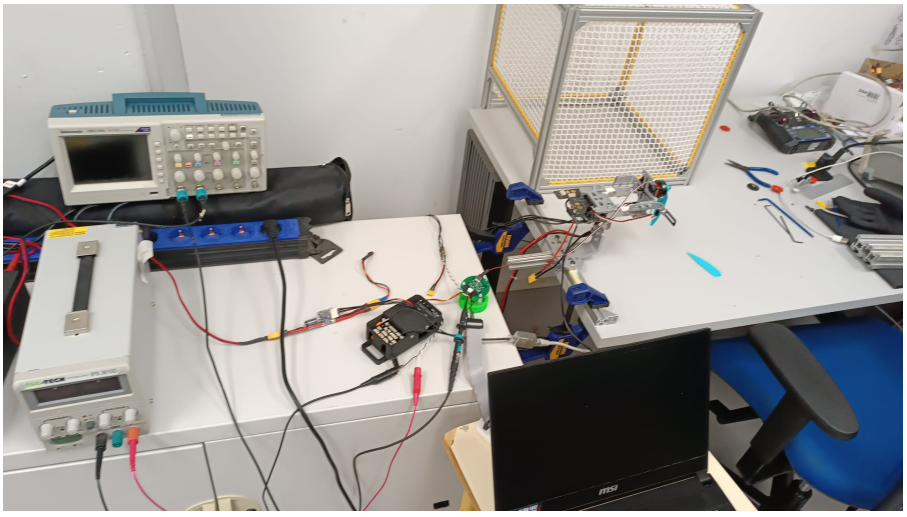
Institut de Robòtica i Informàtica Industrial
CSIC
UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Faculty of Mathematics, FME-UPC

Thrust helix force controller construction and programming



May 12, 2023

Internship Report

Pau Sabater Nácher

Supervised by:
Joan Solà, IRI Titular scientist researcher
Hugo Duarte, IRI Support Engineer

1 Abstract

Add your own abstract here.

Thanks

I want to thank all the people who helped me in the realization of this internship.

First, I want to thank my tutor, ***Joan Solà*** *for the time he spent during both the realization of this project and for the knowledge he shared with me all along my internship.*

I also want to thank all the laboratory's mobile robotics team for their welcome and their expertise in particular ***Patrick Grosch*** *who taught me how to use several workshop instruments and* ***Hugo Duarte*** *who helped me a lot both during the internship in the understanding of the way ROS works and making my arrival to the IRI easy and sharing his experience of it.*

Last, I would like to recognize ***Kevin Becquet*** *from Sorbonne Polytech Université, as this internship is based on his previous works. Despite I did never meet him, all his work at IRI has helped me to achieve my objectives.*

Thanks to all of these people I learned a lot during my internship and I'm grateful for all the experience I acquired doing it.

Contents

Vocabulary

PWM: *Pulse Width Modulation*, squared signal with a variable duty cycle.
ESC : *Electronic Speed Controller*, component that allows the conversion between a PWM signal and a three phased signal usable by the motor in order to spin

2 Introduction

I did my internship in the Mobile Robotics team of the Institut de Robòtica i Informàtica Industrial (IRI), a Robotics laboratory located in Barcelona (Spain). In particular, my task was at the department of *High Dynamics Robotics*. The goal of this internship was to implement a system that allows the control of a motor by the thrust it produces.

https://github.com/hidro-iri

Motivations

The laboratory is currently working on a project: the realization of an agile drone. This drone wants to be able to follow complex trajectories and thus react quickly. To make this drone react quickly, they use a model predictive control that sends commands in force to the drone and in particular its motors' force.

For now, the motors are commanded in open loop. The only thing that is controlled is the signal sent to the motor. The force produced by the motor is calculated using the motor speed and models of aerodynamics and of the propeller attached to the motor.

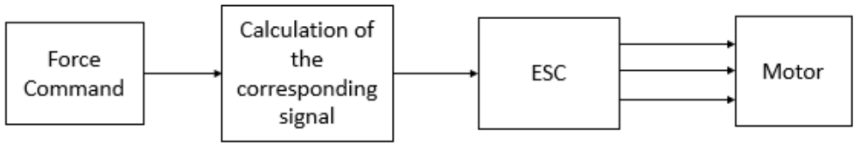


Figure 1: Previous "open loop" design sketch

However, there could be air perturbations or other things that the models do not take in consideration making the force determination not accurate enough.

Thus, to bypass those calculations, we want to create a closed loop controller that will make the motor produce the desired force. By putting a sensor right under the motor, we allow the possibility of a feedback of the forces applied to it.

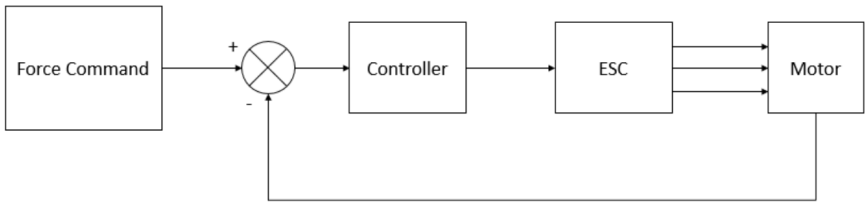


Figure 2: Desired "close loop" design sketch

Objectives

Hence, the goal is to close the loop from the current system to control force. This loop must have a faster dynamic than the global system that sends the force commands not to be the part that limits the drone's overall performance.

Moreover, as we want a fine control of the drone's trajectories, we don't want any error between the command and the real thrust.

Kevin Becquet previously worked on this project, tried to find a solution and implemented it in a standard Arduino board. He find some problems and limitations with that board, so that a full custom PCB was crutial to be designed and printed with some capabilities from the previous board and all the complements needed to achieve the objective. We will now implement this solution in the custom PCB, we will calibrate the PCB by component assembling so as to work in the range of values allowed by the hardware limitations, and we will implement a new improved algorithm solution for the controller system with STM32.

Methodology

This project can be split in two parts: the hardware conception and the software implementation.

For the hardware part, we can use a more detailed version of the figure (2) by putting in parallel the hardware components that will be used to build this controller.

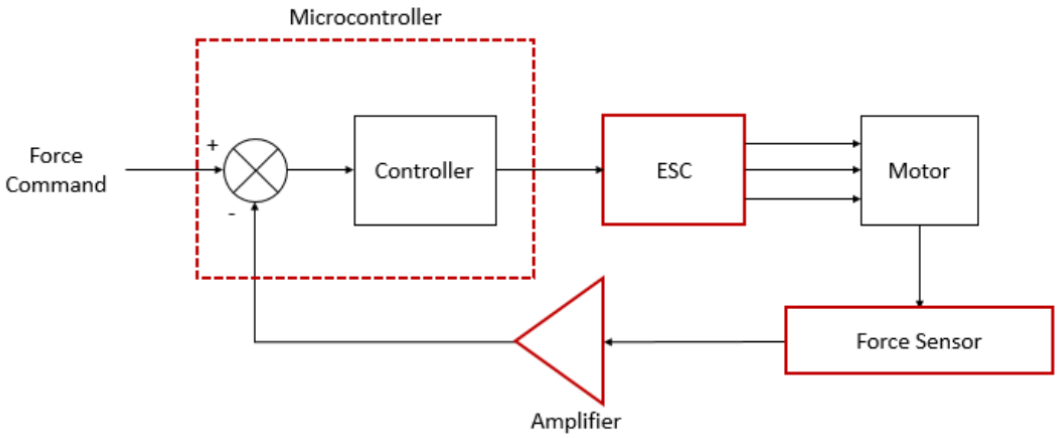


Figure 3: Hardware design sketch

Concerning the software part, we need the microcontroller to implement these for tasks:

1. Receive force commands from the UAV (*Unmanned Aerial Vehicle*) main controller.
2. Get the thrust values from the sensor and filter them.
3. Implement the controller and tune it.
4. Send the corresponding signal to the ESC to make the motor produce the desired thrust.
5. Send the data to the UAV main controller to confirm the results we get. Finally, the tests were realized in the laboratory the motor caged for safety reasons and fixed to a benchmark to get an external measurement of the force it produces. To get these measurements we fixed the motor to a benchmark, a tool able to collect several data like the rotation speed, the torque or the thrust of the motor.

With this background, the following internship objective was proposed:

Thrust controller implementation in custom PCB with STM32.

3 Hardware

This part is consecrated to the hardware components used during this project, their explanation and how they are connected together. To make a control in thrust of the motor we will need to put a force sensor to know how much force it produces. Then we will also need an amplifier to make the sensor's signal readable by the microcontroller which will compute the signal to send to the ESC, the component that

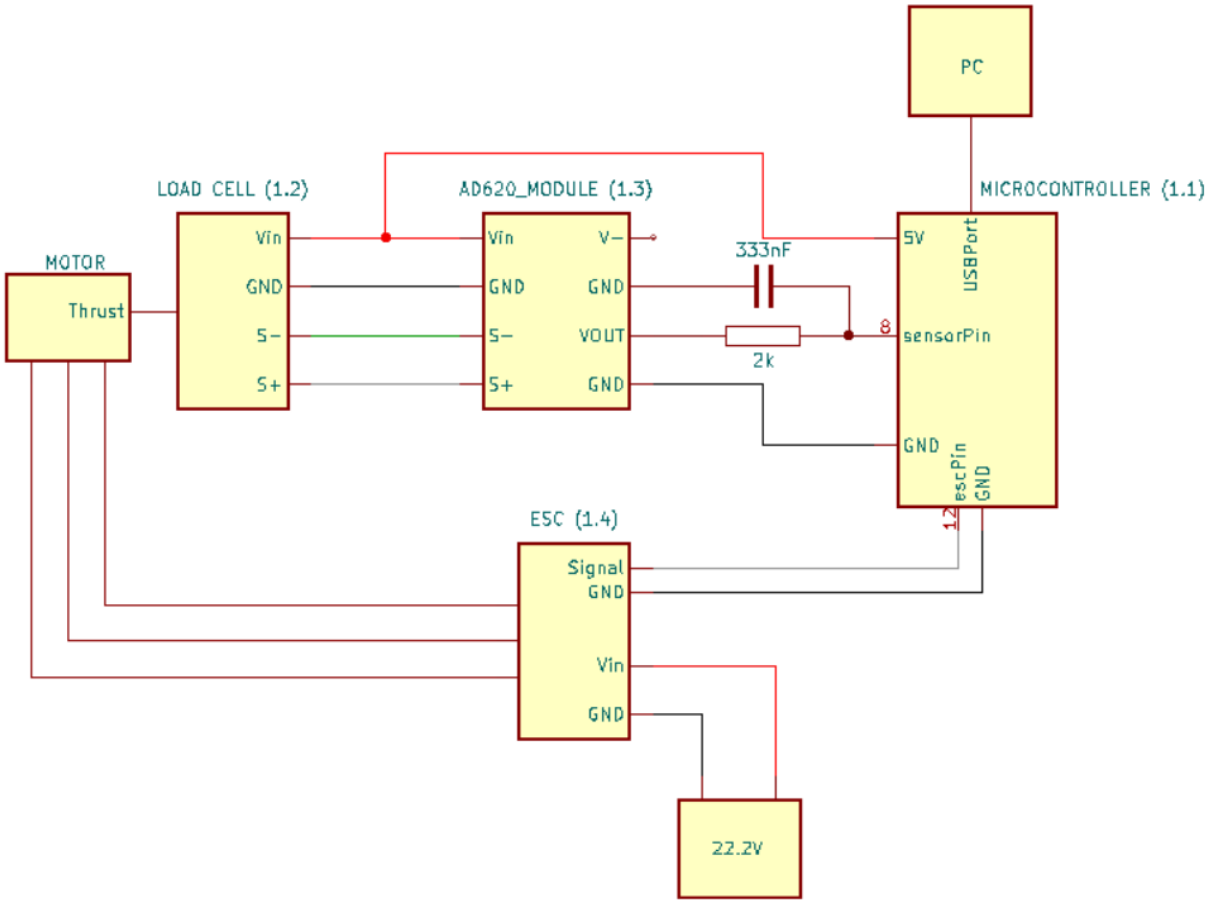


Figure 4: System block schematic

3.1 Microcontroller

The microcontroller used in this project in a STM32L011F4Px mounted on a printed board. It will allow the implementation of the software part and the communication between the other components. The microcontroller is connected to a PC with an USB wire through a communication ST-link/V2 device 5b for debugging.

For the connection with the PCB, we have a 4 pin connector. In the documentation of the ST-link, you can find that the connections are:

(a) Device with welded wires for the PCB connection.

Pin no.	ST-LINK/V2 connector (CN3)	ST-LINK/V2 function	Target connection (JTAG)	Target connection (SWD)
1	VAPP	Target VCC	MCU VDD(1)	MCU VDD(1)
2				
3	TRST	JTAG TRST	JNTRST	GND(2)
4	GND(3)	GND(3)	GND(3)(4)	GND(3)(4)
5	TDI	JTAG TDO	JTDI	GND(2)
6	GND(3)	GND(3)	GND(3)(4)	GND(3)(4)
7	TMS_SWDIO	JTAG TMS, SW IO	JTMS	SWDIO
8	GND(3)	GND(3)	GND(3)(4)	GND(3)(4)
9	TCK_SWCLK	JTAG TCK, SW CLK	JTCK	SWCLK
10	GND(5)	GND(5)	GND(4)(5)	GND(4)(5)
11	Not connected	Not connected	Not connected	Not connected
12	GND	GND	GND(4)	GND(4)
13	TDO_SWO	JTAG TDI, SWO	JTDO	TRACESWO(6)
14	GND(5)	GND(5)	GND(4)(5)	GND(4)(5)
15	NRST	NRST	NRST	NRST
16	GND(3)	GND(3)	GND(3)(4)	GND(3)(4)
17	Not connected	Not connected	Not connected	Not connected
18	GND	GND	GND(4)	GND(4)
19	VDD(3)	VDD (3.3 V)(3)	Not connected	Not connected
20	GND	GND	GND(4)	GND(4)

1. The power supply from the application board is connected to the ST-LINK/V2 debugging and programming board to ensure signal compatibility between the boards.
2. Connect to GND for noise reduction on the ribbon.
3. Available on ST-LINK/V2 only, not connected on ST-LINK/V2-ISOL.
4. At least one of these pin must be connected to the ground for correct behavior, it is recommended to connecting all of them.
5. GND on ST-LINK/V2, used by SWIM on ST-LINK/V2-ISOL (see *Table 3*).
6. Optional: for Serial Wire Viewer (SWV) trace.

(b) JTAG/SWD cable connections.

Note that we used then pins 1 and 2 for the voltage supply, 6 for GND, and 7 and 9 for SWDIO and SWCLK, respectively, the debugging connections.

3.2 Force Sensor

To build a thrust controller, we need, in a first part, to know what force the motor produces. The motor (with the propeller) can produce a maximum thrust of 20N. This motor cannot produce negative forces. Therefore, we want our system to be able to measure precisely the forces applied to the motor in a range from -5N to 25N.
The sensor used to do so is a load cell. It is built fixing a Wheatstone full bridge over an aluminium structure.

The load cell is fixed at the base of the motor by a 3D-printed fixation piece and some screws:

Figure 6: Fixation between the motor and the load cell

3.2.1 Wheatstone bridge

A Wheatstone bridge is an electronic circuit composed by 4 strain gauges acting like variable resistors.
When no force is applied on the sensor the 4 strain gauges have the same resistive value. However, when

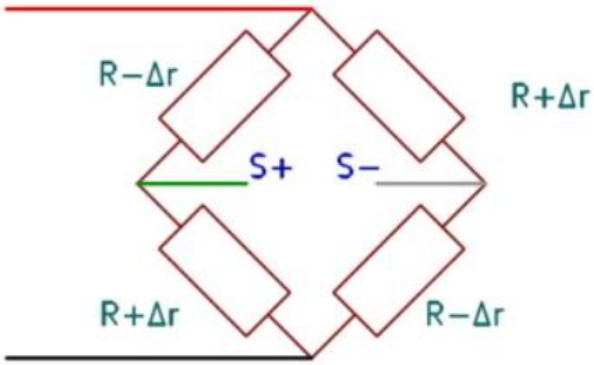


Figure 7: Wheatstone bridge schematic

a force is applied to the sensor, the aluminium plaque will bend under this force causing two of the strain gauges to be in compression, decreasing their resistivity while the two other will be in tension, increasing their resistivity.
The output signal can be determined using Kirchoff's laws:

$$V_{S^+} = \frac{R+\Delta r}{2R} \cdot V_{IN} \ ; \ V_{S^-} = \frac{R-\Delta r}{2R} \cdot V_{IN} \Longrightarrow V_S = V_{S^+} - V_{S^-} = \frac{\Delta r}{R} \cdot V_{IN} \tag{1}$$

We have now a linear relation between the output of the sensor and the force applied to it.

3.2.2 Pros and cons

This sensor provides a voltage that varies linearly according to the force applied to it.

Some experimental measurements with the force sensor were made. Given some weights, the applied force in the sensor due to this mass satisfies $F = m \cdot a$, so then I measured the produced output voltage of the season, using a high precision multimeter.

m(kg)	$F = m \cdot g(N)$	$\Delta V(mV)$
-2.532	-24.82626	-1.733
-2.032	-19.92376	-1.474
-1.532	-15.02126	-1.194
-1.032	-10.11876	-0.9914
-0.532	-5.21626	-0.683
0	0	-0.378
0.532	5.21626	-0.1312
1.032	10.11876	0.0904
1.532	15.02126	0.317
2.032	19.92376	0.551
2.532	24.82626	0.802

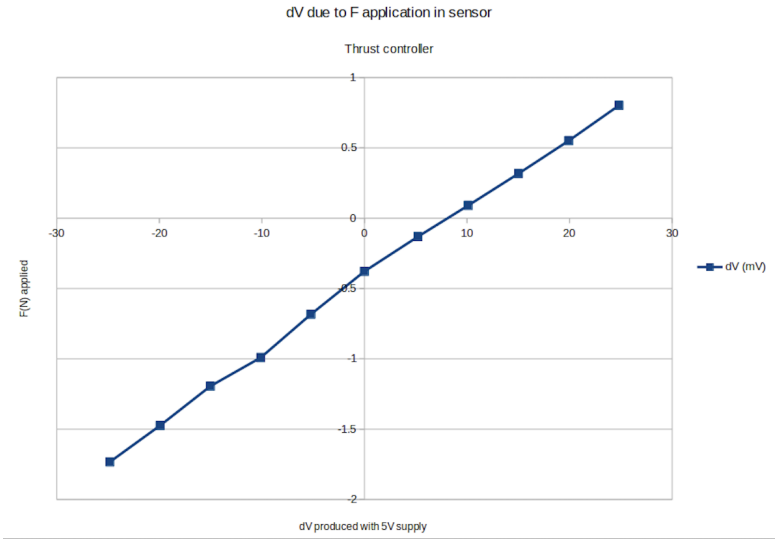


Figure 8: ΔV measured (OY) after F application (OX).

Then the average rope is:

$$a = \frac{\Delta V_{MAX} - \Delta V_{min}}{F_{MAX} - F_{min}} = 0.051 \frac{mV}{N} = 0.50031 \frac{mV}{kg}$$

However, when powered with a 5V supply, the scale of the voltage variation is around 1mV each 10N. This sensor will need to be amplified to be useful in our system.

Therefore, this offset must be calibrated for each cell, which motivated us to create a calibration protocol/method.

3.3 Amplifier

The purpose of this component is to make the signal from the load cell usable to the microcontroller.

This amplifier is an analog voltage amplifier. So the problem of sampling rate is solved as the digital conversion will now be done by the microcontroller, which is able to sample up to a maximum frequency of 10kHz.

The module possesses two potentiometers. One tunes the gain between the input and the output while the other tunes the offset of the value. Indeed, a calibration of the component is necessary to use it. But before the calibration, we can notice a noise non-negligible in the output signal.

3.3.1 Potentiometer Calibration

The microcontroller has analog input pins that can handle voltage from 0V to 3.3V. So we need the amplifier to provide the biggest gain possible while staying in that range for all the force values between -5N and 25N that the sensor receives.

Lots of work was done before in this aspect by Hugo Duarte, but we found its value overcame the 3.3V microcontroller limit. I had to modify its value so as to make the amplifier output stay inside 0 and 3.3V, the output voltage of the module follow this formula:

$$\begin{cases} 0 = G \cdot \Delta V_{min} + V_{Ref} \\ 3.3V = G \cdot \Delta V_{max} + V_{Ref} \end{cases} \Rightarrow \Delta V \cdot G = V_{\text{meas}} - V_{\text{ref}} \tag{2}$$

where G is the gain of the amplifier , as 3.3V is the maximum voltage that the controller is able to work with, and R_1 and R_4 can be identified in the schematic. Therefore, and as V_{ref} is a voltage divisor:

$$(\Delta V_{MAX} - \Delta V_{min}) \cdot G + V_{\text{ref}} = 3.3V \qquad V_{ref} = 5V \cdot \frac{R_4}{R_1 + R_4}$$

$$G \leq \frac{3.3 - V_{ref}}{\Delta V_{MAX} - \Delta V_{min}} \tag{3}$$

Moreover, reading the documentation of the AD620 amplifier, we can compute the value of the resistor as:

$$R_G = \frac{49.4}{G-1} k\Omega \tag{4}$$

It can be find in the sensor TAL107F documentation, fig (43), that the sensibility of the sensor is:

$$(1 \pm 0.2) \frac{mV}{V \cdot 10kg} \tag{5}$$

But it actually works nicely accurate. As our voltage supply is $5V$, hence the rope of the measurements taken yesterday (shown in picture (26)) must be:

$$\Delta V = am + b \quad \text{such that} \quad a \in \left[\frac{(1-0.2)mV \cdot 5V}{V \cdot 10kg}, \frac{(1+0.2)mV \cdot 5V}{V \cdot 10kg} \right] = [0.4, 0.6] \frac{mV}{kg} \simeq [0.04, 0.06] \frac{mV}{N} \tag{6}$$

Which holds, let's plot it: h(red): $a_{\text{min}} = 0.04 \frac{mV}{N}$, g(blue): $a_{\text{MAX}} = 0.06 \frac{mV}{N}$, f(green): $a_{\text{measured}} = 0.05 \frac{mV}{N}$

Following this argument, as we can receive a sensor with sensibility with all this values, our range of data $G \cdot (\Delta V_{MAX} - \Delta V_{min})$ must be between 0 and $3.3V - V_{ref}$ even in the worst case possible, $a = 0.6$, in which:

$$0.6 \frac{mV}{kg} = a = \frac{\Delta V_{MAX} - \Delta V_{min}}{25N - (-5N) \cdot \frac{1kg}{9.805N}} \Longrightarrow \Delta V_{MAX} - \Delta V_{min} = \frac{0.6 \cdot 30}{9.805} mV \simeq 1.85mV \tag{7}$$

And hence our resitor value will be:

$$G = \frac{3.3V - 1.65V}{1.85mV} = 891.89 \Longrightarrow R_G = \frac{49.4}{G-1} k\Omega = 55.45\Omega \tag{8}$$

In case we cannot find of this value, we will choose one with the value **over, bigger than** the one computed,

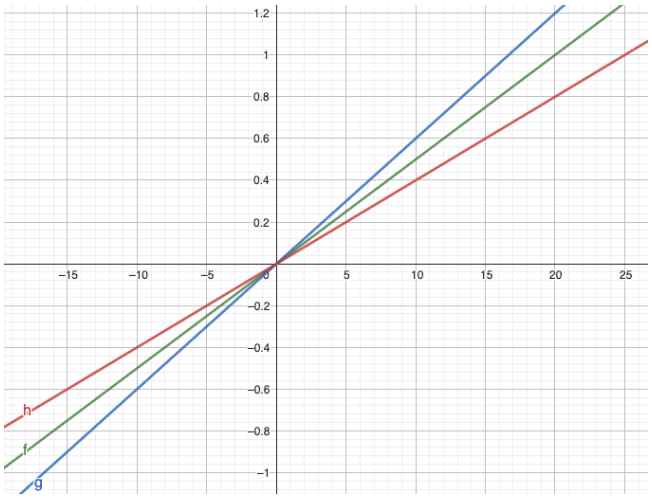


Figure 9: OY: $\Delta V(mV)$, OX: $F(N)$

as in other cases we might produce a too much big gain coefficient (G), for which our range of voltage could be out of the working interval of the controller.

With this new configuration of the resitor, the values of the measured voltage, and $\Delta v = v_{S+} - v_{S-}$ might have changed. With some experimental measurements, we noticed that:

m(kg)	-0.5	0	0.5	1	1.5	2	2.5
$\Delta v(mV)$	-0.645	-0.388	-0.144	0.100	0.357	0.600	0.837
V_{meas}(V)	1.1	1.34	1.55	1.78	2	2.22	2.43
$\Delta V = V_{\text{meas}} - V_{\text{ref}}$(V)	0.558	0.318	0.098	0.120	-0.328	-0.562	-0.837

Then notice that the average rope and the dV values are the same as before, but check that the amplified values are now inside our range. We will plot it for visual understanding. In blue, the values measured on 23/3/2023, in orange, today's ones. We know that $V_{\text{ref}} = 1.68 \simeq 1.65V$, plotted in purple, and the values for V_{meas} are in green. The diference is plotted in yellow. Now $V_{\text{meas}} \in (0.7, 3.3)$, not saturating!

By using an external tool measuring the force and a voltmeter, we will start by tuning the potentiometer (see schematic) dealing with the gain. The goal is to have a voltage range the as close to 3V as possible when we apply a force range of 30N to the sensor.

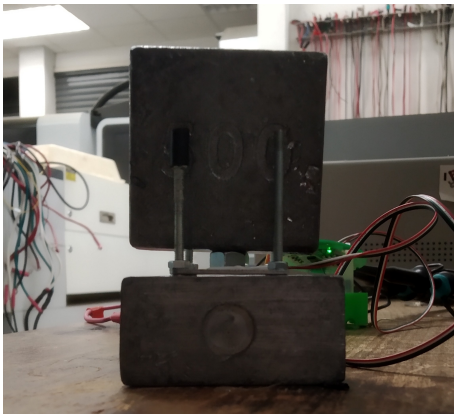


Figure 10: Front view of the weight measurement.

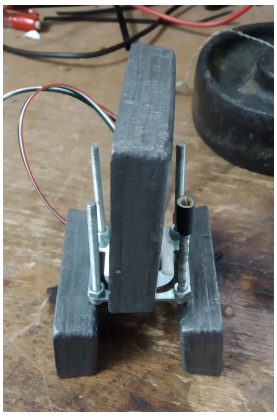


Figure 11: Side view.

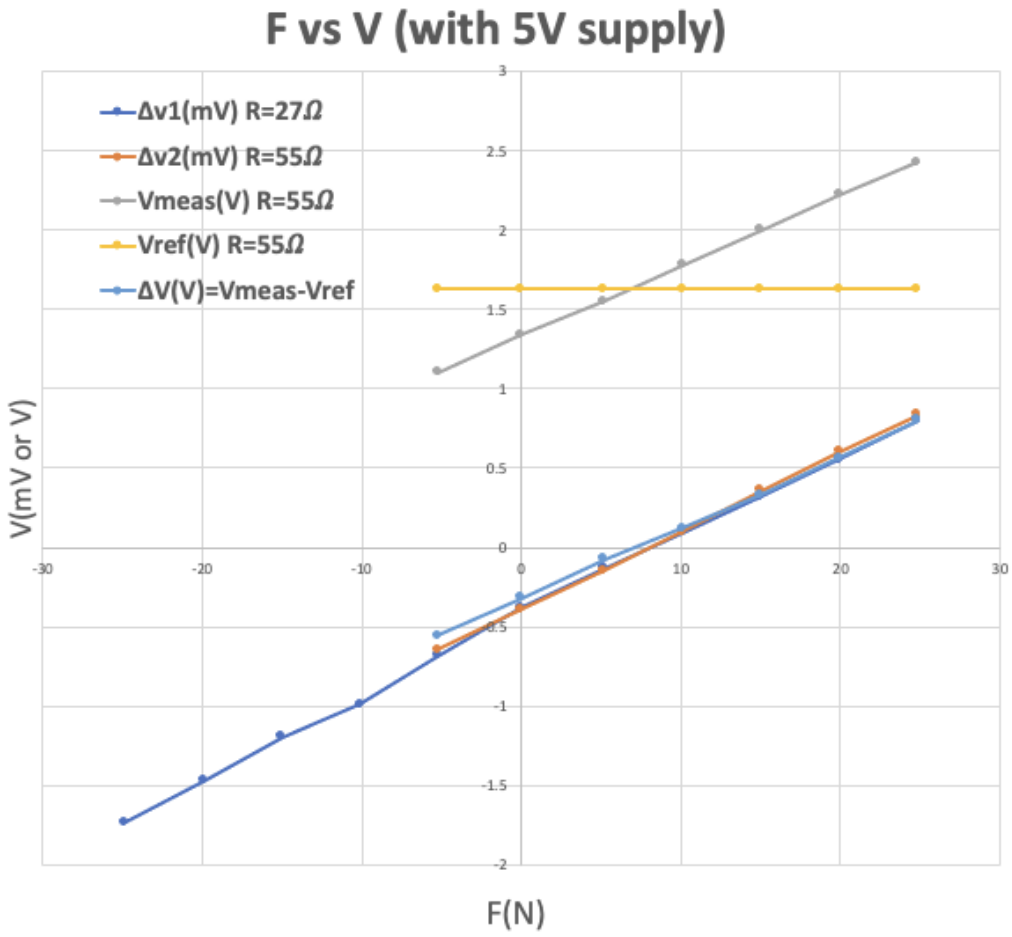


Figure 12: ΔV measured (OY) after F application (OX).

Once the gain is tuned, we need to tune the offset potentiometer (R7 on the schematic) so the corresponding signal for -5N gives a value over 0V to allow the microcontroller to read it.

Due to the second amplifier connected, we know that the $V_{Ref} = \frac{R_4}{R_1+R_4}$ and must be $\leq 3.3V$. Moreover, it will finally hold that:

You can check that $\boxed{\Delta v = aF + b}$ and then that $\boxed{\Delta V = G \cdot \Delta v + V_{\text{ref}}}$, $\begin{cases} 0 = G \cdot \Delta V_{min} + V_{Ref} \\ 3.3V = G \cdot \Delta V_{max} + V_{Ref} \end{cases}$, where a, b, G have been computed the previous days.

Also notice that there's the GAP between Δv at $F = 0$, of $-0.388mV$, which means the resistors in Wheatstone bridge are not exactly the same. So as to correct that gap, we can introduce a new extra resistor (R1 in the figure) thousand times bigger than the other (R1,2,3) the just after the sensor (to avoid analog noise), to reduce the voltage, as following:

Notice that R1 could be parallel to R2 or R4 if S- is bigger, or parallel to R3 or R5 if S+ is the bigger one.

Else we can manipulate the value in the software part, with the risk that the values of the amplified voltage could achieve out-of-range values, misleading into fatal errors.

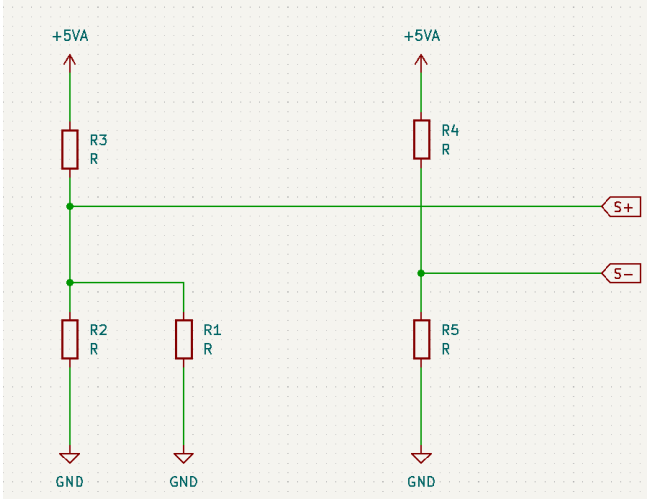


Figure 13: Wheatstone bridge modified for offset elimination.

3.4 Comunications

For our drone system, the comunicating protocols were and are, respectively:

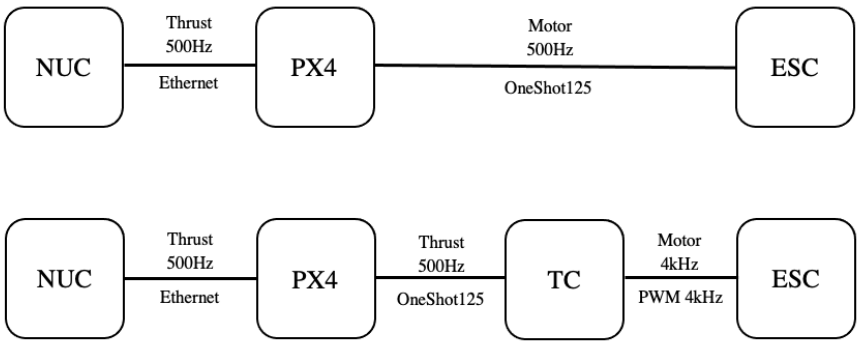


Figure 14: Previous and current comunication protocols sketch

Where NUC, is the global controller of the drone, which sends the commands for thrust through the PX4, and now we are implementing a thrust controller TC between the PX4 and the ESC.

3.5 Electronic Speed Controller

The ESC is in charge of making the motor spin according to the command we send it.

Its input signal is a pulse length signal and it generates an output three-phase signal from it that is usable by the motor to spin.

The PWM protocol used by the ESC is the OneShot125.

3.5.1 OneShot125

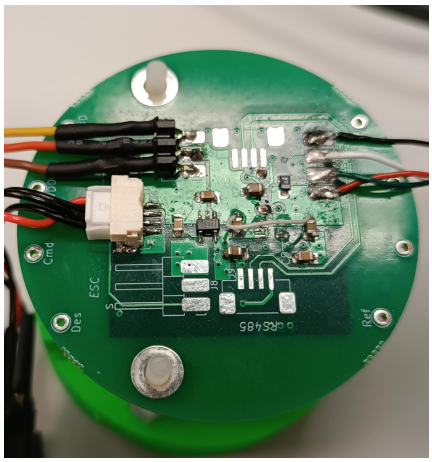
The microcontroller will have to generate a PWM signal that respects this protocol to make the motor spin.

OneShot125 is defined by the length of the throttle of one PWM period. This throttle has to be between $125\mu s$, the minimum pulse that can be sent corresponding to a non-spinning motor, and $250\mu s$, the maximum pulse, that corresponds to a motor spinning at full power.
As the maximum pulse length is $250\mu s$, we can send signals to the ESC at a rate of 4kHz.

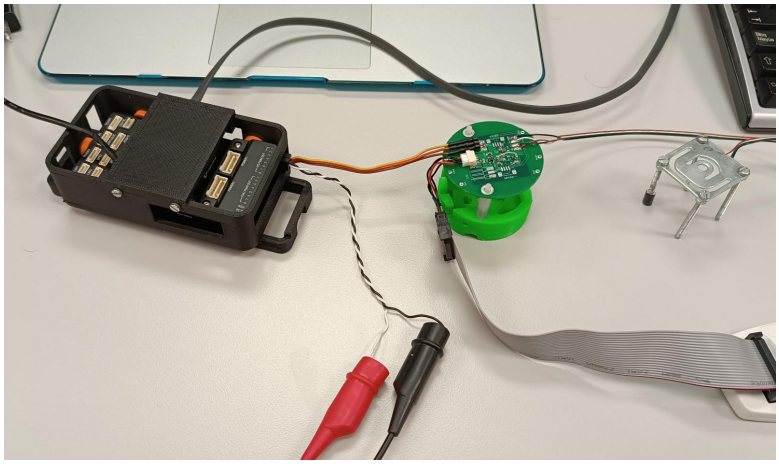
3.6 Wiring and supply

The different components are powered up in a certain way.

- The load cell and the amplifier are powered up by the 5V supply that can provide the microcontroller.
- The ESC powered up by a 22.2V power supply (replicating the $6 \cdot 3.7$V provided by the drone's battery).



(a) PCB source connections welding



(b) Set-up update for PWM reading.

Figure 15: Set-up update for PWM input configuration.

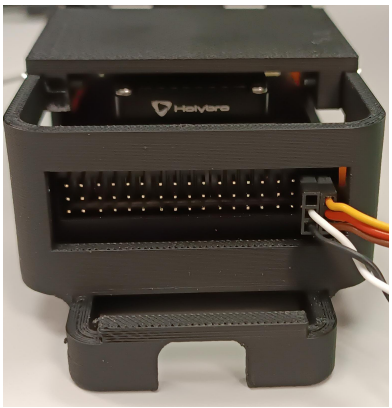
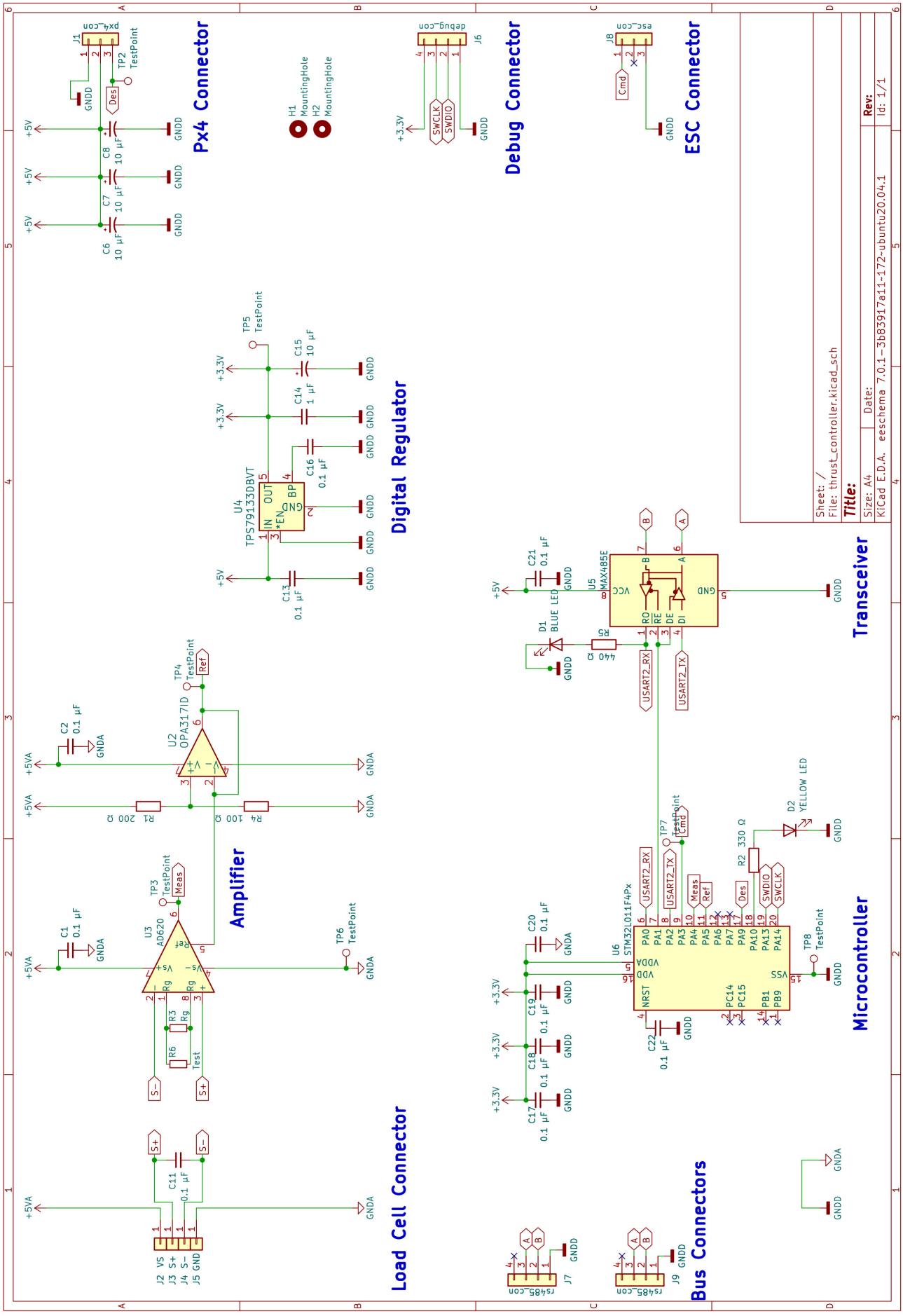


Figure 16: PX4 wire connection to receive signal, power and ground.

For the wiring, the following figures describe the electric connection between the components.


Px4 Connector
Debug Connector
ESC Connector
Digital Regulator
Amplifier
Load Cell Connector
Transceiver
Microcontroller
Bus Connectors
Sheet: /
File: thrust_controller.kicad_sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. eeschema 7.0.1-3b83917a11-172~ubuntu20.04.1
Id: 1/1

4 Software

This part describes how the microcontroller manages to send to the ESC the command that will make the motor produce the thrust we want.

To make this happen, the microcontroller, in each iteration, make these tasks:

- Check if a new user set point command has been received.
- Read the thrust signal from the amplifier and treat it to be usable.
- Compute the output force with a PID controller with feedforwarded thrust.
- Send the PWM command signal to the ESC.
- Send several data to the PC (output thrust, command, ...)

But first let's see how the algorithm is built.

4.1 Algorithm structure

As the ESC can receive signal at a frequency of 4 kHz, we want the microcontroller's algorithm go to the same rate to be able to modify the signal sent to the motor as fast as we can.

To do so, we will use one the microcontroller's hardware timer. This timer is able to count a defined amount of time and execute an interruption when the time passed.

4.1.1 Timer interruption

The program uses hardware timers (microcontroller) to make an iteration of the algorithm every 250μs (4kHz).

Once the timer counted 250μs, the program enters the interruption function where it executes one iteration of the code described below.

4.1.2 Interruption function

1. Check for data from PC.
2. Read and treat thrust.
3. Control.
4. Convert force to PWM.
5. Send signal to the ESC.
6. Send data to the PC. End.

This sums up all the task the microcontroller needs to do in an interval of 250μs.

After measuring the time passed in the interruption, we noticed that this function spends around 35μs to execute all its tasks. So, the time passed in the interruption is not problematic.

4.2 Communication with the PC

This part will detail the communication between the microcontroller and the PC: how the command messages are sent and how the data are received.

The communication between the system and the computer is made with the microcontroller through the USB serial port.

4.2.1 Receiving commands

4.2.1.1 Previous Arduino board PWM reading

The force command is sent by writing a message through the USB port. In a first time, this message sent to the microcontroller meant to be only force command.
However, this method showed its limits quickly as we wanted to tune the controller or tare the force sensor frequently. Thus the messages sent were no longer command messages but configuration messages that now contains:

- ✓Mode
- ✓Force comand
- ✓Vibration filter coefficient
- ✓Controller's coeff. (K_P, K_i, K_d, K_{ff})

The mode parameter determines what has to be modified in the parameters:

Mode	Action
0	Send a command to the ESC.
1	Send a controller configuration.
2	Disable the controller and stop the motor.
3	Tare the load cell.

4.2.1.2 Current custom board PWM reading

1. Create a CubeIDE (or MX, etc) project. As always, select the debug serial wire as the interface in *System core/SYS*.

2. Select your *Clock configuration*. As always, for our microcontroller we will modify the preescalers so as to obtain $32MHz$.

3. Now we will configure the timer that will be used for the PWM input. As we want to use the pin $PA9$, we will use the timer *TIM21*. In its confiuration page, select *Internal clock* as the *Clock source*.

4. For a PWM input, we will need to store the period of the total pulse period, but also the period of the duty cycle, so we will use two different channels associated to the only timer for this task. Select the option **PWM Input on CH2** in the *Combined Channels* drop down.

5. In *Parameter Settings*, let the *Counter Period* be the highest amount allowed, in our case a 16 bits value: 65535, so as to avoid overfloating as the values for the time period and the pulse must be inside 0 and ARR(Counter Period).

6. At the end of the configuration, when executing the program and programming the value for the PWM time, you could find some weird values. This happens due to the overfloat of the value of the *Counter Period* that we have configured previously. We can fix this problem by prescaling the internal clock frequency, in other words, by taking a value different from 0 at the *TIM21/Parameter Settings/Prescaler*. **Example:** we had the problem that our period was around 80000 and our Counter Period was 65535, by putting a Preescaler of $16-1$, our time dropped to 4000, which nicely worked.

7. Now it´s time to set the channel configuration. As explained before, our input will be on the *Channel 2*, so this will be our **Direct** *IC Selection*, with **Raising Edge** *Polarity*. Therefore, for our duty channel, we will select **Falling Edge** *Polarity* and **Indirect** *IC Selection*. The other values will be the default ones.

8. Don't forget to tick the box for the global interrupt in the TIM21 *NVIC Settings*.

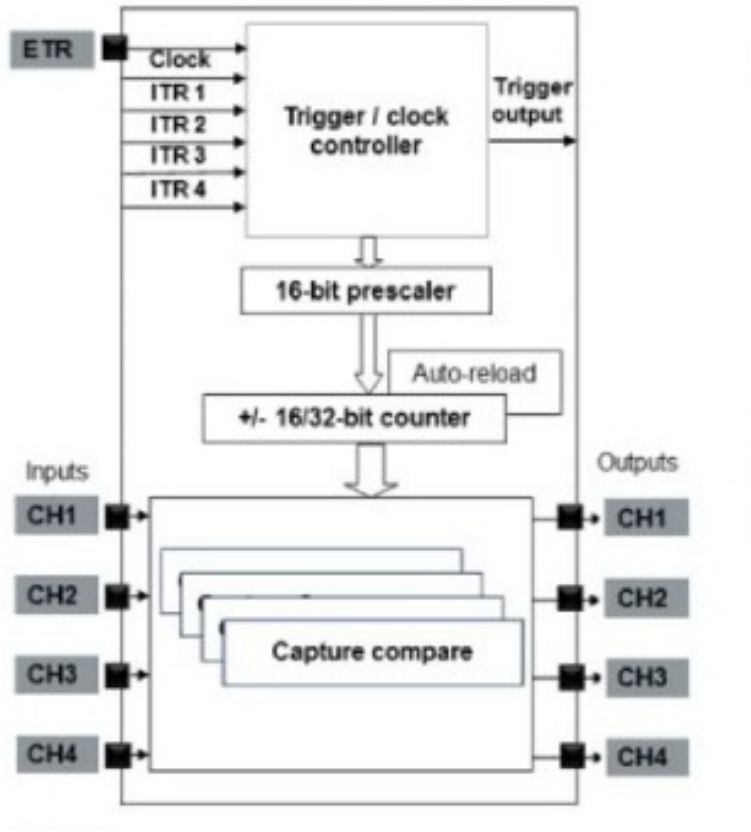


Figure 17: Timer work

9. Finally, after generating the code we will set the software section in *main.c* file:

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
//Global variables declaration.
    //############################## PWM INPUT#################################
    float Duty_input = 0; //Duty cycle coefficient: 0->0%, 1->100%
    uint32_t PWM_input_period = 0; //Period of the PWM function (counts timer clicks)
    uint32_t PWM_input_pulseON = 0; //Pulse width, of the positive part, on.
    uint32_t Frequency_input = 0;//Frequency of the PWM hole function (Hz)
/* USER CODE END 0 */
```

Inside the main function, initialize the input capturing channels:

```
/* USER CODE BEGIN 2 */
    //IC means Input Capture
    //IT menas Callback Interrupt
    //##############################PWM INPUT#################################
    // Main channel (Function Period Measure):
```

```
    HAL_TIM_IC_Start_IT(&htim21, TIM_CHANNEL_2);
    // Indirect channel (Pulse Width Measure):
    HAL_TIM_IC_Start(&htim21, TIM_CHANNEL_1);
/* USER CODE END 2 */
```

Finally, create the callback function for capturing the data and computing the values:

```
/* USER CODE BEGIN 4 */
//####################PWM INPUT INTERRUPT FUNCTION ####################
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim){
// If the interrupt is triggered by channel 2
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2){
        // Read the IC value
        PWM_input_period = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
    //If it´s not the initial 0 capture (=>no pulse registered)
        if (PWM_input_period != 0){
            PWM_input_pulseON = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
            // Compute the Duty Cycle
            Duty_input = (float)(PWM_input_pulseON)/(float)PWM_input_period;
            //Compute the frequency of the PWM. 32000000 is the timer frequency
            Frequency_input = 32000000/(PWM_input_period);//Divide also by prescaler if used
        }
    }
}

/* USER CODE END 4 */
```

Now you are ready to read an PWM input at *live variables* in STM32CubeIDE.✓

4.2.1.3 Current custom board sensor input and ADC

1. Create new project in STM32CubeIDE, activate the *Debug Serial Wire*, as always.
2. In *Analog/ADC*, tick the IN peripherical conection. Enable Continuous conversion Mode in *Parameter settings*, this will make the microcontroller to be constantly measuring the input voltage. In *NVIC Settings*, enable the *NVIC ADC, COMP1 and COMP2 interrupts* tick box. The *Sampling time* represents how many cycles does microcontroller count between ADC conversions.
3. You can check in *Clock configuration* that there´s the ADC own clock prescaled from the internal clock.
4. Generate the code. Then create a variable for the input, in our case:

```
/* USER CODE BEGIN PV */
//################################ADC INPUT SENSOR################################
uint16_t V_meas = 0;
/* USER CODE END PV */
```

5. Also

```
/* USER CODE BEGIN 2 */
//##################################ADC INPUT SENSOR#################################
HAL_ADC_Start_IT(&hadc);
/* USER CODE END 2 */
```

6. And finally:

```
/* USER CODE BEGIN 4 */
//##################ADC INPUT SENSOR INTERRUPT FUNCTION##########################
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc1)
{
    V_meas = HAL_ADC_GetValue(hadc1);
}
/* USER CODE END 4 */
```

And it´s ready to be compiled, debugged, executed and checked in the *Live Expressions* interface that V_meas shows the Voltage input in milivolts(mV).

4.2.2 Sending data to the PC

As for the command, sending measurements started with only the measured thrust. But quickly, we wanted to see more data than that. The data sent by the controller ended up being:

- The measured thrust.
- The command.
- The output thrust of controller.
- Its different part (proportional, integral, derivative and feedforward).
- The pulse length encoded over 10bits sent to the ESC.

These data have been useful to check if the results of the controller were coherent and how should the controller be tuned according to its responses. This still isn´t implemented in the current custom board as USART pin was not welded, but would be nice to be implemented in the future.

4.3 Implementation of the controller

The controller implemented is a PID controller with a feedforward term.

To reduce the noise, a software signal filter can be implemented by adding:

```
float k = 0.001;//Remainance of the previous value
Thrust_mesured = k*Thrust_mesured + (1-k)*HAL_ADC_GetValue(hadc);
```

The higher the value of k, the higher the filtering but the lower the velocity of response.

4.3.1 Proportional

The proportional part of a PID controller computes the error between the measurement and the command and send it to the output.

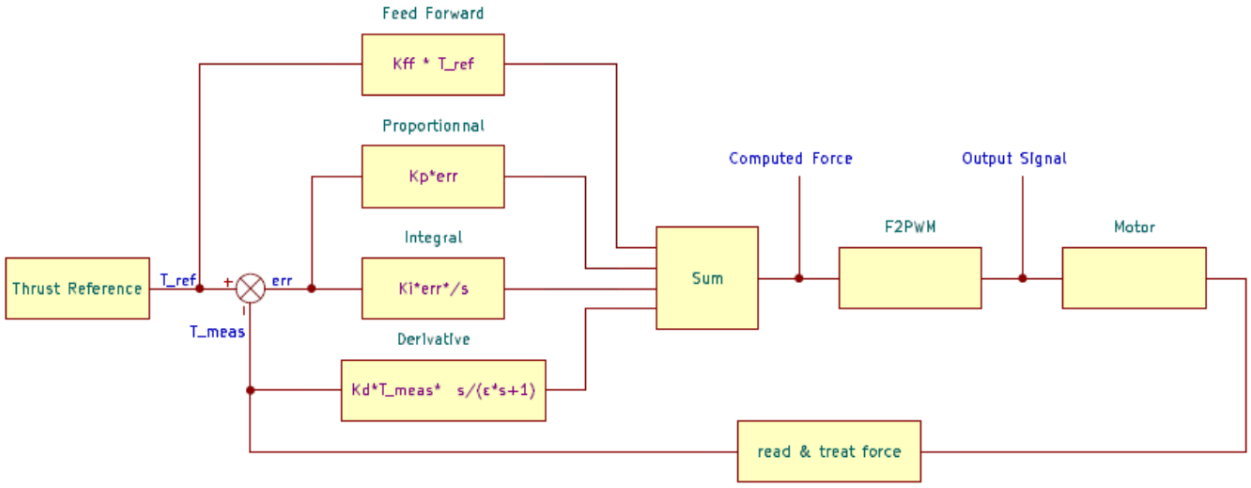


Figure 18: Detailed schematic of the controller

4.3.2 Derivative

The derivative part of a PID controller is used to fasten the stabilization of the force around the command. However, its use increases the noise in the signal. A standard real derivation was implemented. However, here there is a derivative approximation:

Real derivative: Derivative model aproximation:

$$K_d \cdot \frac{df(t)}{dt} \Longrightarrow K_d \cdot s \cdot F(s) \qquad K_d \cdot \frac{df(t)}{dt} \Longrightarrow \frac{K_d \cdot s}{\varepsilon s + 1} \cdot F(s) \quad \text{with} \quad 0 < \varepsilon \ll 1 \tag{9}$$

This approximation is made to limit to increasing of the noise allowing us to have a greater derivative part without increasing too much the noise.
To implement this in the program it is necessary to convert this Laplacian expression in the $\mathbb{Z}$-domain (discrete values). We replace s with $\frac{1-z^{-1}}{T_s}$ where T_s is the sampling period of the system. From (reference): $T_s = 250 \cdot 10^{-6}$s.

Thus, we get:$K_d \cdot \frac{1-z^{-1}}{T_s+\varepsilon-\varepsilon \cdot z^{-1}}$, and defining the thrust measurement, T_m, and converting to recursive form corresponds to:

$$Der_{out}(k) := D_o(k) = \frac{\varepsilon \cdot D_o(k-1) + K_d \cdot (T_m(k) - T_m(k-1))}{T_s + \varepsilon} \tag{10}$$

where $Der_{out}(k)$ is the derivative of $T_m(k)$ at $t = kT_s$

4.3.3 Integral

The Integral part of a PID controller nullifies the error between the command and the output of the system. In digital control the integral is made by adding successively the current error in the integral sum:

$$\int_{out} = \int_{out} + Ki \cdot err \tag{11}$$

In case the syste, is saturated and connot reach the command to send it, an anti-windup has been added: when the computed Thrust passes over 14N, or when it decreases under 0N (the motor cannot provide forces over 14N nor negative forces), the integration is stopped meaning that the program stops adding terms in the integration sum:

```
if ( (*t_c <= A_TU) && (*t_c >= A_TD) ) {
    I_out += Ki / Freq * error; //integration of the error if the output is not saturated.
    *t_c = FFout + Pout + Iout + Dout;
}
```

where **t_c** is the thrust compute and **A_TU**, **A_TD** are the antiwindup thresh up and down,

4.3.4 Feed forward

The feed forward term is the response the system would have given to the command if it was in open loop. It doesn't react with the change in the feedback.

This term is useful in a controller because it shortens the rising time of the system output allowing a faster stabilization of the measurement around the command overall.

4.4 Communication with the ESC

The communication between the microcontroller and the ESC is made by sending a PWM signal, as this way we will send frequential pulses that is what ESC reads.
Thus to send a force command to the ESC we have to know how to send a PWM signal and how to convert a force into it.

4.4.1 Signal sending

As OneShot125 protocol understands pulse lengths between 125μs, and 250μs, the last one corresponding to motor full power, the lower frequency of pulses is 250μs, or equivalently the signal sent to the ESC is a PWM signal at 4kHz.

4.4.1.1 Previous Arduino board PWM generation

We started by using the Arduino function analogWrite() but as its main purpose is to create an analog output using the average of a PWM signal, the result was not as good as expected: the change in the PWM were not synchronized with its period and the results was that some throttles were fused . Moreover, the basic PWM frequency of the analogWrite() is 400Hz. So we needed a way to fix both problems.
To send a PWM signal at this frequency, there are some functions in the OpenCM's board manager that are useful. Indeed, there is a PWM driver implemented in it. So send the signal at the desired frequency and pulse require only one line of code for each:

```
drv_pwm_setup_freq(pinOut, FREQ);
drv_pwm_set_duty(pinOut, PWM_RES, dc);
```

Where:

- **pinOut** is the output pin on which the PWM signal is sent.
- **FREQ** is the desired frequency of the signal.
- **PWM_RES** is the resolution of the duty cycle sent.
- **dc** is the duty cycle encoded over PWM_RES bits.

However, there still was glitches in the PWM signal. This was due to the way drv_pwm_set_duty updated the PWM signal sent: to modify the shape of the PWM signal, we have to modify a register of the microcontroller. This function clears the whole register before rewriting the new value. This action makes a glitch in the signal each time the register is modified. We fixed this by modifying the function to stop prevent the register from resetting but only modify the bits that are responsible of the PWM duty cycle.

4.4.1.2 Current custom board PWM generation

Steps:

1. Create a CubeIDE (or MX, etc) project. As always, select the debug serial wire as the interface in *System core/SYS*.
2. Select your *Clock configuration*. As always, for our microcontroller we will modify the preescalers so as to obtain $32MHz$.
3. Now we will configure the timer that will be used for the PWM generation. As we want to use the pin $PA3$, we will use the timer *TIM2*, as *TIM21* is used for the input. In *TIM2* configuration page, select *Internal clock* as the *Clock source*.
4. For a PWM putput, we will need a pin, and therefore a channel, through which the data will be sent (the period of the total pulse period, but also the period of the duty cycle). In our case we will select *Channel 4*, for which we will select the *PWM Generation CH4*.
5. In *Parameter Settings*, let the *Counter Period* (from now on we will refer to it as the ARR, *Auto-Reload Register*) be whichever value, it usually is 65535 by default, we will modify it in the c file. **Enable *Auto-reload preload***
6. At the end of the configuration, when executing the program and programming the value for the PWM time, you could find some weird values. This happens due to the overfloat (16 bits) of the value of the *Counter Period* that we have configured previously. We can fix this problem by prescaling the internal clock frequency, in other words, by taking a value different from 0 at the *TIM2/Parameter Settings/Prescaler*.
7. Other parameter will be set as default.
8. Finally, after generating the code we will set the software section in *main.c* file:

Firstly, we will need to declarate the variable we will need all along the code, global variables:

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
//################################PWM OUTPUT#################################
    uint32_t Frequency_output = 600; //in Hertz
    float Duty_output = 0.45; //From 0 to 1
    uint32_t ARR4 = 0; //Counter Period(Auto Reload Register) see TIM2 Parameter Settings.

/* USER CODE END 0 */
```

Inside the main function, from documentation, we know that the PWM frequency satisfies:

$$\nu_{PWM} = \frac{\nu_{Clock}}{(ARR+1)\cdot(PSC+1)} \qquad \Rightarrow \qquad ARR = \frac{\nu_{Clock}}{\nu_{PWM}(PSC+1)} - 1 \tag{12}$$

Where v_{PWM} is the desired frequency for the PWM generation, v_{Clock} in our case is $32MHz$ and PSC and ARR are the parameter we will configure in the code to obtain our desired v_{PWM}

```
/* USER CODE BEGIN 1 */
    //################################PWM OUTPUT#################################
    ARR4 = (3200000/Frequency_output)-1;
/* USER CODE END 1 */
```

Moreover, we know also from documentation that:

$$\text{DutyCycle}_{PWM} = \frac{CCR4}{ARR4} \tag{13}$$

Where the coefficient 4 is as our channel used is the *Channel 4*. Then, we will have to initialize the PWM generation and declare the desired duty cycle:

```
/* USER CODE BEGIN 2 */
    //################################PWM OUTPUT#################################
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_4); //Generation initialization.
    TIM2->CCR4 = (Duty_output*ARR4);//Duty Cycle configuration.
/* USER CODE END 2 */
```

Finally, note that if after coding you save from the .ioc file, the coding modifications will be removed and rewritten to default ones. To avoid this, remember to:

```
/* USER CODE BEGIN TIM2_Init 1 */
//################################PWM OUTPUT#################################
        //Remember to change htim2.Init.Period = ARR4;
/* USER CODE END TIM2_Init 1 */
```

Now you are ready to generate a PWM output, and you could check variables work at *live variables* in STM32CubeIDE. Furthermore, it´s easy then to generate a pulse length between 0 and the frequency of the PWM by sending a desired value:

4.4.2 Force-PWM mapping

4.4.2.1 Previous Arduino board PWM generation

To make the motor produce a certain force we have to know what signal we have to send for it to reacts on a coherent way.
The relation between the length of the throttle and the thrust the motor creates is quadratic. The formula describing it has this form:

$$\text{Thrust} = a \cdot (PWM - 511)^2 + b \cdot (PWM - 511) \tag{14}$$

with PWM the length of the throttle encoded over 10bits. 511 is retracted to PWM because it is the minimum throttle that can be sent and the motor it stopped at this value. 511 corresponds to 50% of duty cycle encoded over 10bits.

Then, as we want the corresponding PWM knowing a certain thrust, we have to invert this equation:
Notation: [y = Thrust] and [x = PWM - 511]

$$\begin{gathered} y = ax^2 + bx + c \iff \\ x = -\frac{b}{2a} \pm \sqrt{\frac{y}{a} + \frac{4ac - b^2}{2a}} \end{gathered} \tag{15}$$

As $c = 0$ and x is increasing with y, substituting the notation:

$$PWM = 511 - \frac{b}{2a} + \sqrt{\frac{Thrust}{a} + \frac{b^2}{2a}} \tag{16}$$

Now the formula between the wanted Thrust and the signal to send is determine, we have to find the a and b coefficients.
To do so, we take several couples of values by making the motor spin by sending specific signals to the ESC and measuring the thrust it provides.

4.4.2.2 Current custom board PWM mapping

With the new PID implementation, no mapping should be needed.

4.4.3 Calibrating

Before use, the ESC needs to be properly calibrated.

The calibration of the ESC is made by sending the maximum signal for a little period of time and the minimum signal right after.

When you power up the ESC it will emit three beeps. After these sounds, send the maximum signal and the minimum one right after. It should emit two more sounds. If it is the case, the ESC is calibrated.

5 Testing

5.1 Tuning

To have the desired response from the controller there were five parameters to tune: the four coefficients of the controller and the one of the vibration filter.

5.1.1 Controller tuning

To have the desired response of our system, the controller needs to be tuned accordingly. It is important to know what modify to obtain the result we want: increase the proportional or integral part will decrease the rising time whereas increasing the derivative part will decrease the stabilizing time.

For example, adding a derivative part will increase the noise in the signal, in a such way that in our first try we added a coefficient that was too high causing a such increase in the noise that the signal got out of the limits of what value it can reach. So the motor stopped itself according to the software security implemented

in the microcontroller's code.

After a few tries, we decided that, in the context of this project, it is better to have a signal that is slower but less noisy rather than one that stabilizes faster but is noisier.

5.2 Final

For the final test, we made a command increasing by 2N until reach 16N.

The controller satisfies our expectations until 14N: the output signal has a short rising time (0.1s), it doesn't have a significant overshoot and its oscillations after the stabilization have a small amplitude(±0.1N).

However, the motor can't produce a force over 14N because of the limits of power it receives. Indeed, it was powered up through the ESC by a power supply that can provide a maximum current of 30A at the ESC's nominal voltage (22.2V). This is the power the motor needs to produce a 14N force. This force allows the drone to create a force able to carry 3 times its weight which is enough to permit it to do complex trajectories.

6 Conclusion

6.1 Project's sum-up

We needed in a first time to build a hardware system:

A load cell was needed to measure the forces applied to the motor. This sensor creates a voltage according to the force measured.

We clearly need a microcontroller to compute the force output to send to the ESC, given an acquired force measurement, and a force command.

According to the cell output characteristics, we need an amplifier to make the force measurements readable by the microcontroller. On this amplifier a filter is added to make the measurement less noisy (the noise is created by a component in the amplifier module). Moreover, we replaced the potentiometer of the module to make the experiments repeatable.

Finally, we need an ESC to make the motor spin according to the force we want to produce.

6.2 Critical analysis

The project of this internship was to implement a control in thrust of a drone motor. It had for goal to be a proof of concept in the purpose of checking if we can bypass some complicated models to compute the force in the drone. Knowing that goal, the project is a success: we managed to make the motor stabilize its thrust by using a sensor. Thus we can now totally bypass the models of aerodynamics or the influence of the shape of the propeller.

However, some things should be upgraded for it to be properly used on a drone:

In a first place, it will be needed a bigger flash memory to store the program and debugging information. Indeed, the one used here works fine while reading the cell measurement, and it is able to create the timer interruption and execute its associated function but when mixing the command reading, the signal , the code cannot be stored in debugging mode within the microcontroller memory, despite being not more than 20kB large.

Then, as the different load cells have different offsets and , . A possible solution is to , if lower than 125μs are not allowed because they may conflict with the ESC communication protocol, maybe larger pulses may be used to enter to a calibration mode.

Moreover, another problem with the PCB hardware was found when powering the board. After flashing the program in the board, it is not properly initialized and executed. It takes several resets to achieve the board to execute the flashed program. The microcontroller could be finding some problems to initialize the file, or something might have broke, so a review must be done for this issue.

In particular, I think a study of the PID convergence and

Finally, despite we haven´t installed USART for communication with PC by USB, this kind of system is really not compatible with the use of the mobile drone: the system is not mobile and the communication by USB port is voluminous knowing we have to command 6 motors in one drone. It would be preferable to use another communication protocol that is adapted with the drone's controllers.

7 Internship Day-to-Day Activities

The internship took place in the months of March to May. Most of the work was done either at the laboratory of IRI, FME, Barcelona, or my place of residence when working from home. The first two weeks were spent reading the literature and discussing possible approaches. The next X weeks were spent on dolor sir amet. Finally the model was evaluated with consectetur adipiscing elif in the last Y weeks.

7.1 PHASE 1: Introduction and approaching

13/03/2023

Brief introduction to the project. Assignation of computer and desk in the lab. Installation of KiCad.

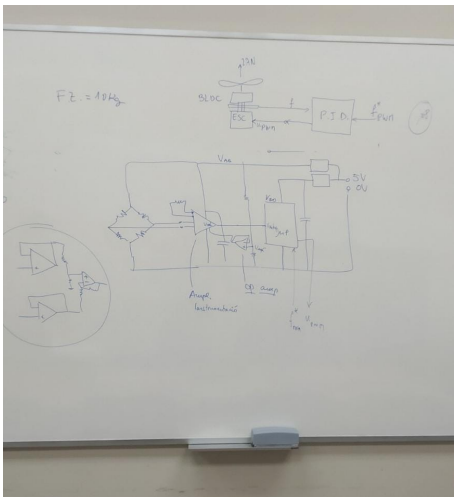


Figure 19: Introduction to the project.

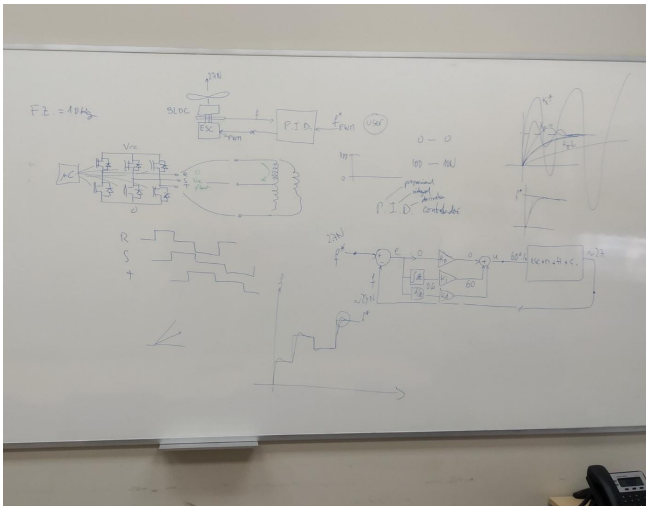


Figure 20: Introduction to the electronics of the project.

14/03/2023

Component welding on the PCB and proper behaviour testing. Installation of Element in IRI computer.

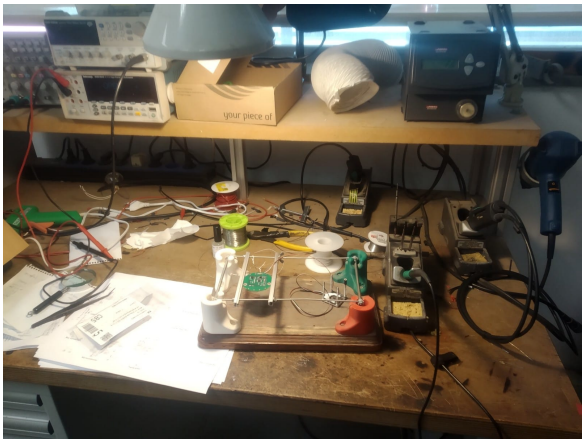


Figure 21: Welding and testing workbench.

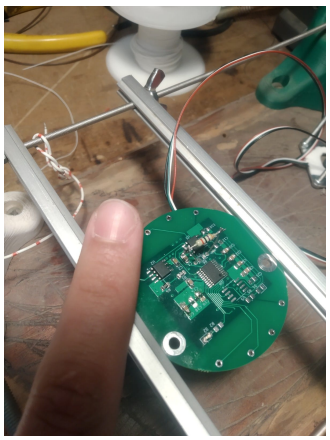


Figure 22: Magnitude of the welded components .

15/03/2023

Received acces to the lab. Welding of components for the ST-Link/V2 connection between the PCB and the computer for programming the microcontroller with STM32 software. STM32 installation and begining of

language sintaxis learning.

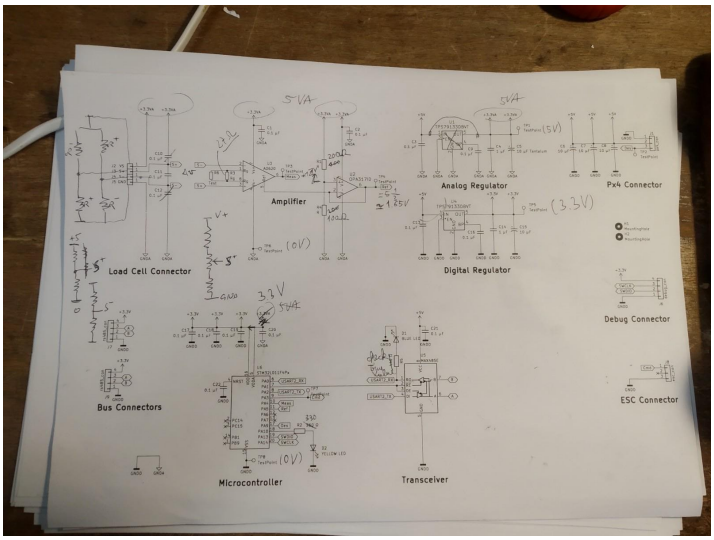


Figure 23: Schematics sketch.

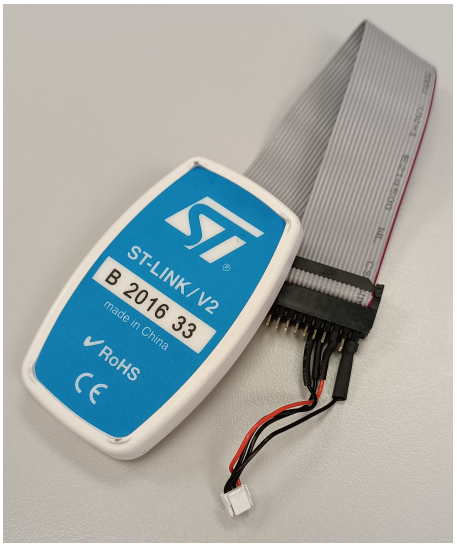


Figure 24: ST-Link/V2 device and conections.

16/03/2023

By following this steps/tutorial I download *STM32CubeMX*, *STM32CubeIDE*, *STM32CubeProg* and used them to configure the connection, create the microcontroller in CubeMX, generate de code (we needed to install some libraries), to finally make an easy code in which we made the PCB LED blink to ensure the PCB was well welded and everything work properly.✓
Location of the project: /home/psabater/thrust/thrust_controller-main/firmware/blink_led If the file is .sh,

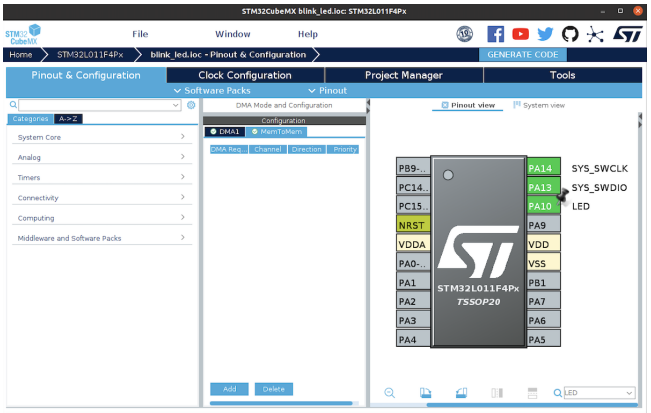


Figure 25: Microprocessor first steps.

*chmod +x ***.sh* and *sudo ./nombre.sh*

7.2 PHASE 2: Software development for PCB microcontroller programming

17/03/2023

Plan: KiCad schematics update. Planification of code implementation in C for STM32 project. Read Kevin Becket previous work.

20/03/2023

Timer example in STM32. You might follow this tutorial (STM32CubeIDE timer TIM2)
On STM32CubeIDE, created a new project. Verified in *Pinout & Configuration/System Core/SYS*, under SYS peripheral, that Debug Serial Wire is selected as Debug interface.

In *Clock Configuration* modified teh operations to obtain $32MHz$
In *Pinout & Configuration/Timers/TIM21*, select *Clock Source: Internal Clock*, and downside in *Configuration/Parameter Settings/PS(Preescaler)* change *Prescaler* value to "32000 – 1" to make the clock feed to the timer $v = \frac{32MHz}{\text{Prescaler value}} = \frac{32MHz}{32000} = 1kHz$. The timer 2' internal counter register will increase by 1 every 1T with $T = \frac{1}{v} = \frac{1}{1kHz} = 1ms$ once the timer starts. **QUESTION: why is TIM2 disabled?** Solved on 21/03/2023

In counter mode, select mode *Up*. Counter is able to count up, which increase its counter value every tick, or count down, which decrease its value every tick.

In counter period (autoreload register), enter "1000-1". This means the counter will increase its internal counter register to 1000 before resetting it to 0 again. Since the duration is 1 ms, the total time to count from 0 to 1000 is 1 second.

Next, we will enable timer interrupt so that an interrupt is generated every time timer's value reaches the counter period. In *NVIC Interrupt Table* or *NVIC Settings*, click on the tick box on the right of TIM21 global interrupt to enable timer 2 interrupt. We will see in the next step how the code is generated for these settings.

Lastly we will configure GPIO pin D10 as an output. Click on the pin in the diagram and select GPIO_Output from dropdown menu. If you click GPIO on the left panel, you will see GPIO D10 is listed in the GPIO Mode and Configuration table.

21/03/2023

7.2.1 Timers management

Today I achieved to control the timer, we used this for toggling a LED given a certain period.✓

Firstly, we noticed the error declaring *static volatile bool timer_expired_flag = false;* was because boolean was not defined well in the framework, and could be easily solved by including the library *#include "stdbool.h"* Moreover, we discovered that in fact that boolean *timer_expired_flag* was not used anywhere in the code so that we removed it.

Secondly, we managed to control the timer by modifying. It would be nice for next days to implement some way to modify the timer interruption with an only variable (which could be the needed **period**). The final procedure to control the timer was:

1. Create the CubeIDE project. Make sure the debug serial wire is selected as the interface (SYS). Also consider selecting a pin in which a LED could be connected to have an output with we could check the proper behavior.

2. Select in the clock configuration the system frequency to be used. The higher the requency is, the more times the controller function will be executed per unit of time.

3. In *Pinout&Configuration/Categories/Timers* selecting *TIM2*. We select the *Clok source*: *Internal Clock*, and the *preescaler* and *Counter period* values. last step is to configure the timer. In With the help of Sergi we understood how the preescale and counter period really worked. Their real meaning is that the preescaler value is the amount of system-clicks that the timer-click is equivalent to. As well, the counter period value is the amount of timer-clicks that are counted before making an interrupt. The preecaler value must include the -1 as the timer escales from 1 to $[preescaler_value + 1]$, and as well for the counter period.

4. IMPORTANT: In TIM2 configuration, at *NVIC Settings*, tick the box which enables *TIM2 global interrupt*.

That way, the generated code will initialize a *systemClock* function, the timer *TIM2* initialization and the LED one. After this considerations the final code additions were:
In the main loop:

```
/* USER CODE BEGIN 2 */
    HAL_TIM_Base_Start_IT(&htim2);
/* USER CODE END 2 */
```

And outside everything:

```
/* USER CODE BEGIN 4 */
    void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_10);//As our LED is in GPIO_PIN_10
        //        timer_expired_flag = true; //comunication issue we weren´t using.
    }
/* USER CODE END 4 */
```

For next day, notice:

```
    htim2.Init.Prescaler = 16000-1;
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 1000-1;
```

And try to simplify the configuration of the period by introducing a new variable, which could be the **period**. For the next day, KiCad Schematics updating as well.

22/03/2023

I already updated the schematics (easy job, not much time). But I must compute the value of the amplifier resistor to so we need the amplifier to provide the biggest gain possible while staying in the range (the microcontroller has analog input pins that can handle voltage from 0V to 3.3V) for all the force values between -5N and 25N that the sensor receives.

23(03/2023

Yesterday I did some experimental measurements with the force sensor. Given some weights, the applied force in the sensor due to this mass satisfies $F = m \cdot a$, so then I measured the produced output voltage of the

season, using a high precision multimeter.

m(kg)	$F = m \cdot g(N)$	$\Delta V(mV)$
-2.532	-24.82626	-1.733
-2.032	-19.92376	-1.474
-1.532	-15.02126	-1.194
-1.032	-10.11876	-0.9914
-0.532	-5.21626	-0.683
0	0	-0.378
0.532	5.21626	-0.1312
1.032	10.11876	0.0904
1.532	15.02126	0.317
2.032	19.92376	0.551
2.532	24.82626	0.802

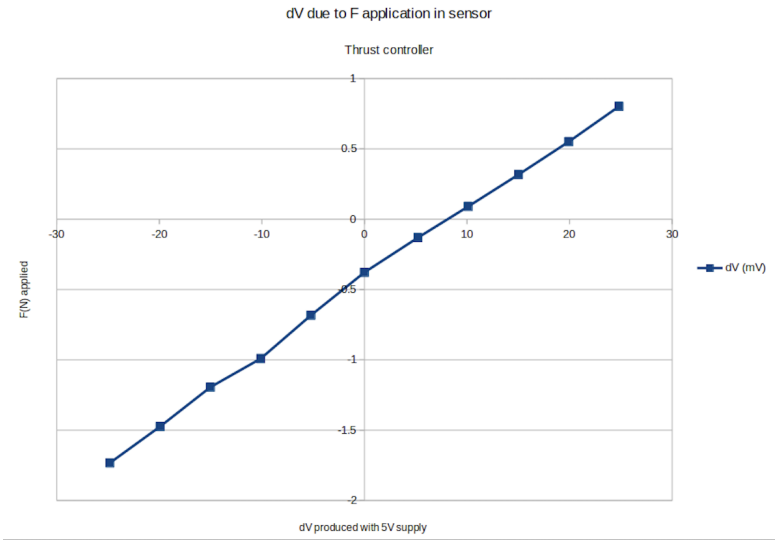


Figure 26: ΔV measured (OY) after F application (OX).

Then the average rope is:

$$a = \frac{\Delta V_{MAX} - \Delta V_{min}}{F_{MAX} - F_{min}} = 0.051 \frac{mV}{N} = 0.50031 \frac{mV}{kg}$$

Next day I will compare the observations with the sensor documentation theoretical values.

NEW PROBLEM: It's easy to notice that as the sensor sensibility is low (see documentation of the sensor TAL107F), fig (43), a calibration phase must be implemented in the software.

24/03/2023

I re-did the calculations because yesterday I was very tired, even having some headache.
We know that the amplifier must satisfy: $\Delta V \cdot G = V_{\text{meas}} - V_{\text{ref}}$

$$(\Delta V_{MAX} - \Delta V_{min}) \cdot G + V_{\text{ref}} = 3.3V \qquad V_{ref} = 5V \cdot \frac{R_4}{R_1 + R_4}$$

where G is the gain of the amplifier , as 3.3V is the maximum voltage that the controller is able to work with, and R_1 and R_4 can be identified in the schematic. Therefore:

$$G \leq \frac{3.3 - V_{ref}}{\Delta V_{MAX} - \Delta V_{min}} \tag{17}$$

Moreover, reading the documentation of the AD620 amplifier, we can compute the value of the resistor as:

$$R_G = \frac{49.4}{G-1} k\Omega \tag{18}$$

Finally, we can find in the sensor TAL107F documentation, fig (43), that the sensibility of the sensor is:

$$(1 \pm 0.2) \frac{mV}{V \cdot 10kg} \tag{19}$$

And as our voltage suppliement is $5V$, hence the rope of the measurements taken yesterday (shown in picture (26)) must be:

$$\Delta V = am+b \quad \text{such that} \quad a \in \left[\frac{(1-0.2)mV\cdot 5V}{V\cdot 10kg}, \frac{(1+0.2)mV\cdot 5V}{V\cdot 10kg}\right] = [0.4,0.6]\,\frac{mV}{kg} = [0.04,0.06]\,\frac{mV}{N} \quad (20)$$

Which is true, let's see that: h(red): $a_{\text{min}} = 0.04\frac{mV}{N}$, g(blue): $a_{\text{MAX}} = 0.06\frac{mV}{N}$, f(green): $a_{\text{measured}} = 0.05\frac{mV}{N}$

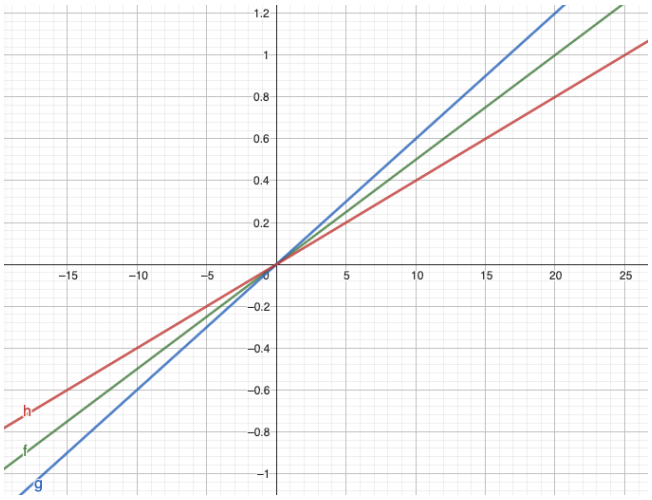


Figure 27: OY: $\Delta V(mV)$, OX: $F(N)$

Following this argument, as we can receive a sensor with sensibility with all this values, our range of data $G\cdot(\Delta V_{MAX} - \Delta V_{min})$ must be between 0 and $3.3V - V_{ref}$ even in the worst case possible, $a = 0.6$, in which:

$$0.6\frac{mV}{kg} = a = \frac{\Delta V_{MAX} - \Delta V_{min}}{25N - (-5N)\cdot\frac{1kg}{9.805N}} \Longrightarrow \Delta V_{MAX} - \Delta V_{min} = \frac{0.6\cdot 30}{9.805}mV \simeq 1.85mV \quad (21)$$

And hence our resitor value will be:

$$G = \frac{3.3V - 1.65V}{1.85mV} = 891.89 \Longrightarrow R_G = \frac{49.4}{G-1}k\Omega = 55.45\Omega \quad (22)$$

In case we cannot find of this value, we will choose one with the value **over, bigger than** the one computed, as in other cases we might produce a too much big gain coefficient (G), for which our range of voltage could be out of the working interval of the controller.

Now it's time to weld the resistor (I will wait for Hugo to check), and start to read data from the sensor. Firstly, I will learn how to display messages in the computer from the PCB.

28/03/2023

So as to configure to communication process, you might follow this tutorial (linux_commands_serial_console)
But we will not consider this problem now as *UART* connecting port is not welded yet.
After checking every computation was OK, we replaced the resistor by one of 55Ω, and check the values obtained:

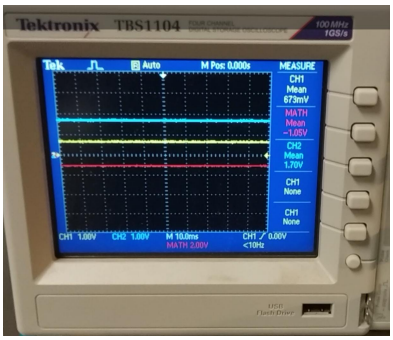


Figure 28: Lower saturation.

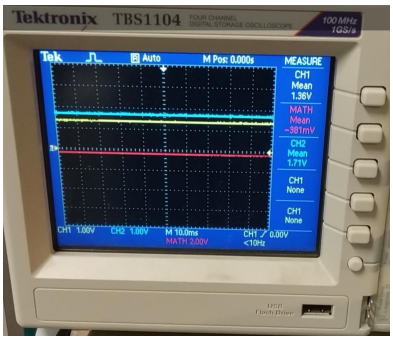


Figure 29: Free of charge.

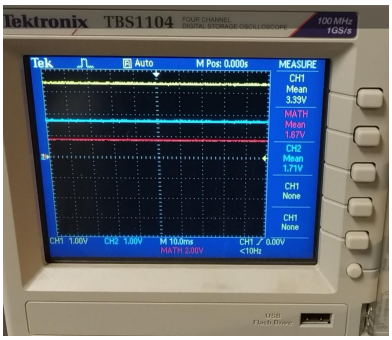


Figure 30: Upper saturation.

	Lower saturation	Sensor without weight	Upper saturation
$V_{measurement}(V)$	0'673	1'36	3'39

Obvserve that in all three cases, $V_{\text{ref}} = 1'71V$ is the same, as we know, its aproximately $1'65V$ with an small error due to the power voltage source precision. Moreover, notice that there´s a gap between V_{meas} and V_{ref} even when there´s no weight. We must check that.

For next day, analyse the range of the voltage output (V_{meas}, V_{ref}), check all this and blink a LED with intensity depending on the force applied on the sensor. I will make the maximum intensity for the maximum force applied stored in the memory with an own variable, i_{max}.

30/03/2023

With this new configuration of the resitor, the values of the measured voltage, and $\Delta v = v_{S+} - v_{S-}$ might have changed. With some experimental measurements, we noticed that:

m(kg)	-0.5	0	0.5	1	1.5	2	2.5
$\Delta v(mV)$	-0.645	-0.388	-0.144	0.100	0.357	0.600	0.837
V_{meas}(V)	1.1	1.34	1.55	1.78	2	2.22	2.43
$\Delta V = V_{\text{meas}} - V_{\text{ref}}$(V)	0.558	0.318	0.098	0.120	-0.328	-0.562	-0.837

Then notice that the average rope and the dV values are the same as before, but check that the amplified values are now inside our range. We will plot it for visual understanding. In blue, the values measured on 23/3/2023, in orange, today's ones. We know that $V_{\text{ref}} = 1.68 \simeq 1.65V$, plotted in purple, and the values for V_{meas} are in green. The diference is plotted in yellow. Now $V_{\text{meas}} \in (0.7, 3.3)$, not saturating!

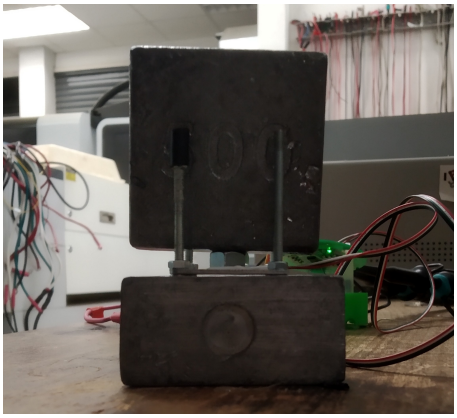


Figure 31: Front view of the weight measurement.

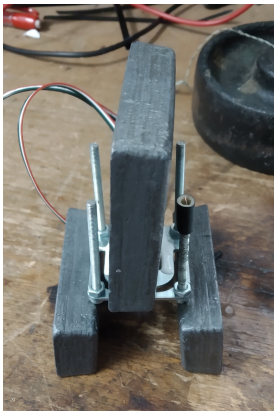


Figure 32: Side view.

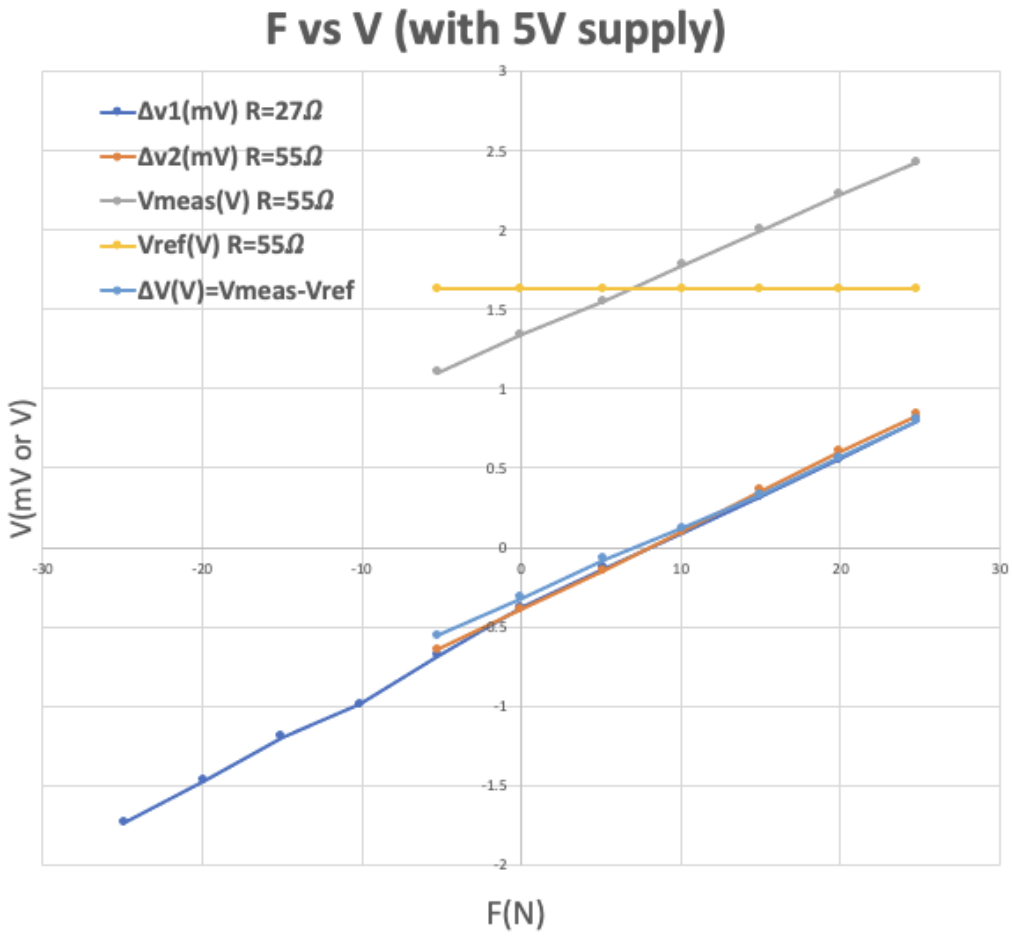


Figure 33: ΔV measured (OY) after F application (OX).

You can check that $\boxed{\Delta v = aF + b}$ and then that $\boxed{\Delta V = G \cdot \Delta v + V_{\text{ref}}}$, where a, b, G have been computed the previous days.

Also notice that there's the GAP between Δv at $F = 0$, of $-0.388mV$, which means the resistors in Wheatstone bridge are not exactly the same. So as to correct that gap, we can introduce a new extra resistor (R1 in the figure) thousand times bigger than the other (R1,2,3) the just after the sensor (to avoid analog noise), to reduce the voltage, as following:

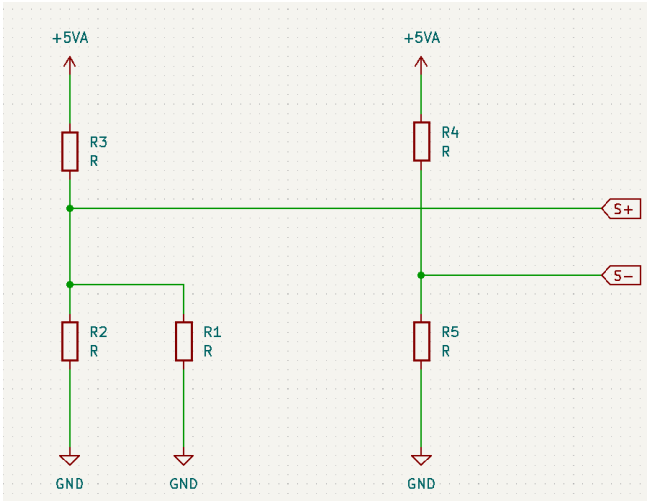


Figure 34: Wheatstone bridge modified for offset elimination.

Notice that R1 could be parallel to R2 or R4 if S- is bigger, or parallel to R3 or R5 if S+ is the bigger one.

Else we can manipulate the value in the software part, with the risk that the values of the amplified voltage could achieve out-of-range values, misleading into fatal errors. ✓

So as to configure analog input pins, go to *Pinout&Configuration/Analog/ADC* (Analog to Digital Converter) and tick the pins you need to use, in our case $IN4$ and $IN5$.
Following this tutorial (ADC) o exemple and for multiple channels.

So now we are ready to begin with the PID algorithm, following the next mapping:

$$\Delta V = (V_{\text{MAX}} - V_{\text{min}}) \cdot \frac{F - F_{\text{min}}}{F_{\text{MAX}} - F_{\text{min}}} \tag{23}$$

31/03/2023 and 11/04/2023

Following previous links and this tutorial and the previous knowledge, I made some trials of how to show the input in the microcontroller to be able to work with it.

12/04/2023

7.2.2 ADC convertion for sensor data input

https://www.youtube.com/watch?v=aqtr1epDoSI

1. Create new project in STM32CubeIDE, activate the *Debug Serial Wire*, as always.
2. In *Analog/ADC*, tick the IN peripherical conection. Enable Continuous conversion Mode in *Parameter settings*, this will make the microcontroller to be constantly measuring the input voltage. In *NVIC Settings*, enable the *NVIC ADC, COMP1 and COMP2 interrupts* tick box. The *Sampling time* represents how many cycles does microcontroller count between ADC conversions.
3. You can check in *Clock configuration* that there´s the ADC own clock prescaled from the internal clock.
4. Generate the code. Then create a variable for the input, in our case:

```
/* USER CODE BEGIN PV */
//################################ADC INPUT SENSOR#################################
uint16_t V_meas = 0;
/* USER CODE END PV */
```

5. Also

```
/* USER CODE BEGIN 2 */
//################################ADC INPUT SENSOR#################################
HAL_ADC_Start_IT(&hadc);
/* USER CODE END 2 */
```

6. And finally:

```
/* USER CODE BEGIN 4 */
//#################ADC INPUT SENSOR INTERRUPT FUNCTION##########################
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc1)
```

```
        {
            V_meas = HAL_ADC_GetValue(hadc1);
        }
        /* USER CODE END 4 */
```

And it´s ready to be compiled, debugged, executed and checked in the *Live Expressions* interface that V_meas shows the Voltage input in milivolts(mV). Otherwise, as the conversion is linear, you should multiply by a empyric constant.

13/04/2023

If configurating the timer, the instructions for the ADC conversion can be moved to the interrupt function.

```
/* USER CODE BEGIN 4 */
    void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
           t = t + 1;
           HAL_ADC_PollForConversion(&hadc, HAL_MAX_DELAY);
           V_meas = HAL_ADC_GetValue(&hadc);
    }
/* USER CODE END 4 */
```

Now it's time for the PWM input. You can also check this tutorial.

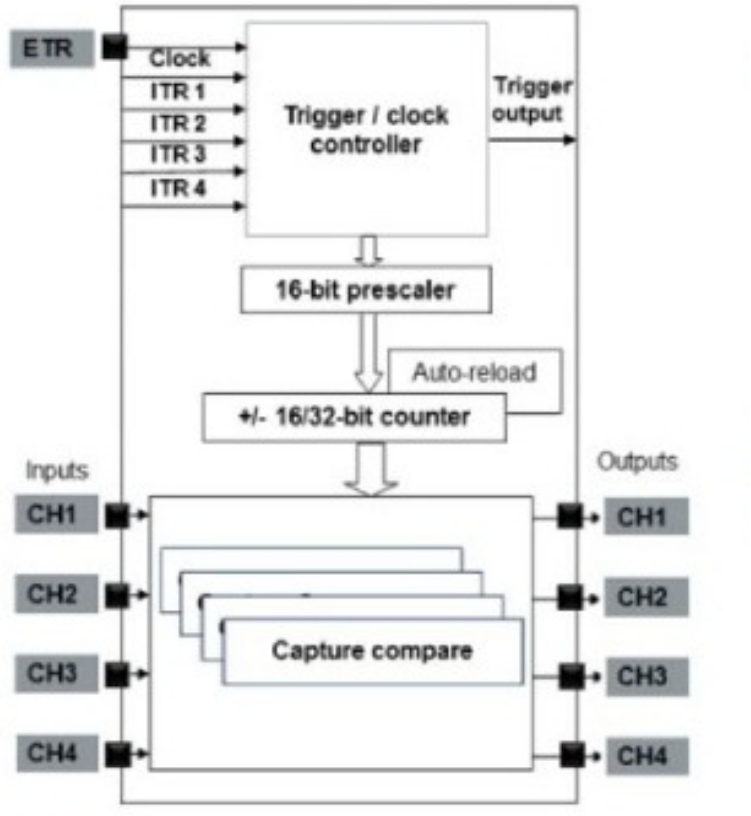


Figure 35: Timer work

14/04/2023

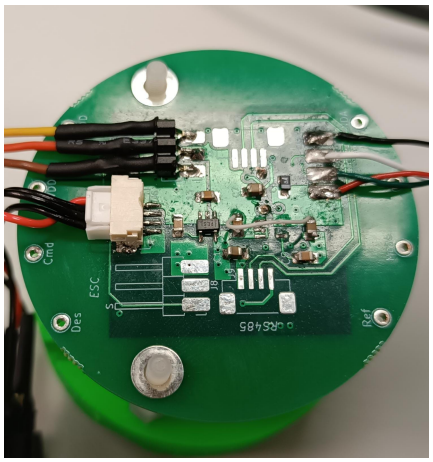
Following this tutorial, or alternatively, or another option.

17/04/2023

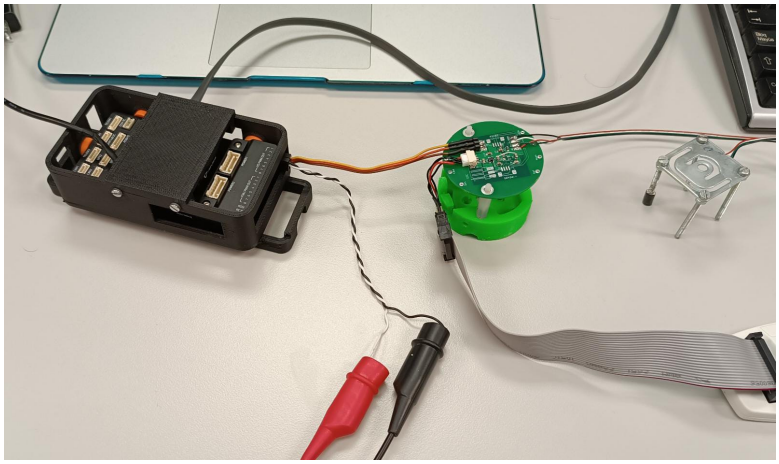
tutorial

18/04/2023

Today I configured the PCB so as to be able to input PWM.



(a) PCB source connections welding



(b) Set-up update after PWM input configuration.

Figure 36: Set-up update for PWM input configuration.

Notice the wires in the left upside corner of the PCB changes, they were previously connected with an adapter which is now used to connect with the PX4 (I reversed the cable and welded it to the PCB connection, and put a thermoretractile protection). Now the power supply comming for the voltage source passes through the PX4. Identify in the image that the superior line of connections is for the **signal**, the lower one is for the ground and the middle one is for the voltage supply (5V).

Figure 37: PX4 wire connection (apart form Ethernet and C-type)

For the software part, I installed a drone controlling app (*QGroundControl*) which lets me control the PWM input. Steps to generate a PWM:

1. Check the PCB set-up is like the previous photos show.
2. Check C-type wire and Ethernet one are conected from the PX4 to the USB ports of the computer with the app
3. Open the app, disconnect any possible device and wait until the app connects with the PX4.

4. Click on the icon of the app located at the left, upside corner. Select *Vehicle Setup* tool, and in *Motors* option at the left list, you will be able to set the desired PWM input.

5. Make sure the lower checkbox is ticked so that *Motor sliders are enabled*.

By modifying the first vertical handler, you can specify the PWM input. ✓

21/04/2023

7.2.3 PWM reading

After lots of work, documentation, and search, I achieved to *input PWM*. Steps:

1. Create a CubeIDE (or MX, etc) project. As always, select the debug serial wire as the interface in *System core/SYS*.

2. Select your *Clock configuration*. As always, for our microcontroller we will modify the preescalers so as to obtain $32MHz$.

3. Now we will configure the timer that will be used for the PWM input. As we want to use the pin *PA*9, we will use the timer *TIM21*. In its confiuration page, select *Internal clock* as the *Clock source*.

4. For a PWM input, we will need to store the period of the total pulse period, but also the period of the duty cycle, so we will use two different channels associated to the only timer for this task. Select the option **PWM Input on CH2** in the *Combined Channels* drop down.

5. In *Parameter Settings*, let the *Counter Period* be the highest amount allowed, in our case a 16 bits value: 65535, so as to avoid overfloating as the values for the time period and the pulse must be inside 0 and ARR(Counter Period).

6. At the end of the configuration, when executing the program and programming the value for the PWM time, you could find some weird values. This happens due to the overfloat of the value of the *Counter Period* that we have configured previously. We can fix this problem by prescaling the internal clock frequency, in other words, by taking a value different from 0 at the *TIM21/Parameter Settings/Prescaler*. **Example:** we had the problem that our period was around 80000 and our COunter Period was 65535, by putting a Preescaler of $16-1$, our time dropped to 4000, which nicely worked.

7. Now itś time to set the channel configuration. As explained before, our input will be on the *Channel 2*, so this will be our **Direct** *IC Selection*, with **Raising Edge** *Polarity*. Therefore, for our duty channel, we will select **Falling Edge** *Polarity* and **Indirect** *IC Selection*. The other values will be the default ones.

8. Don't forget to tick the box for the global interrupt in the TIM21 *NVIC Settings*.

9. Finally, after generating the code we will set the software section in *main.c* file:

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
//Global variables declaration.
    //############################## PWM INPUT#################################
    float Duty_input = 0; //Duty cycle coefficient: 0->0%, 1->100%
    uint32_t PWM_input_period = 0; //Period of the PWM function (counts timer clicks)
    uint32_t PWM_input_pulseON = 0; //Pulse width, of the positive part, on.
```

```
    uint32_t Frequency_input = 0;//Frequency of the PWM hole function (Hz)
/* USER CODE END 0 */
```

Inside the main function, initialize the input capturing channels:

```
/* USER CODE BEGIN 2 */
    //IC means Input Capture
    //IT menas Callback Interrupt
    //################################PWM INPUT#################################
    // Main channel (Function Period Measure):
    HAL_TIM_IC_Start_IT(&htim21, TIM_CHANNEL_2);
    // Indirect channel (Pulse Width Measure):
    HAL_TIM_IC_Start(&htim21, TIM_CHANNEL_1);
/* USER CODE END 2 */
```

Finally, create the callback function for capturing the data and computing the values:

```
/* USER CODE BEGIN 4 */
//####################PWM INPUT INTERRUPT FUNCTION ####################
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim){
// If the interrupt is triggered by channel 2
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2){
        // Read the IC value
        PWM_input_period = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
    //If it´s not the initial 0 capture (=>no pulse registered)
        if (PWM_input_period != 0){
            PWM_input_pulseON = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
            // Compute the Duty Cycle
            Duty_input = (float)(PWM_input_pulseON)/(float)PWM_input_period;
            //Compute the frequency of the PWM. 32000000 is the timer frequency
            Frequency_input = 32000000/(PWM_input_period);//Divide also by prescaler if used
        }
    }
}

/* USER CODE END 4 */
```

Now you are ready to read an PWM input at *live variables* in STM32CubeIDE.✓

Tutorial of PWM generation and alternatively, and specially, this one.

24/04/2023

7.2.4 PWM generation

With this previous introduction to PWM management, it was easy to achieve the *PWM generation*. Steps:

1. Create a CubeIDE (or MX, etc) project. As always, select the debug serial wire as the interface in *System core/SYS*.

2. Select your *Clock configuration*. As always, for our microcontroller we will modify the preescalers so as to obtain $32MHz$.

3. Now we will configure the timer that will be used for the PWM generation. As we want to use the pin $PA3$, we will use the timer *TIM2*, as *TIM21* is used for the input. In *TIM2* configuration page, select *Internal clock* as the *Clock source*.

4. For a PWM putput, we will need a pin, and therefore a channel, through which the data will be sent (the period of the total pulse period, but also the period of the duty cycle). In our case we will select *Channel 4*, for which we will select the *PWM Generation CH4*.

5. In *Parameter Settings*, let the *Counter Period* (from now on we will refer to it as the ARR, *Auto-Reload Register*) be whichever value, it usually is 65535 by default, we will modify it in the c file. **Enable *Auto-reload preload***

6. At the end of the configuration, when executing the program and programming the value for the PWM time, you could find some weird values. This happens due to the overfloat (16 bits) of the value of the *Counter Period* that we have configured previously. We can fix this problem by prescaling the internal clock frequency, in other words, by taking a value different from 0 at the *TIM2/Parameter Settings/Prescaler*.

7. Other parameter will be set as default.

8. Finally, after generating the code we will set the software section in *main.c* file:

Firstly, we will need to declarate the variable we will need all along the code, global variables:

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
//################################PWM OUTPUT#################################
    uint32_t Frequency_output = 600; //in Hertz
    float Duty_output = 0.45; //From 0 to 1
    uint32_t ARR4 = 0; //Counter Period(Auto Reload Register) see TIM2 Parameter Settings.

/* USER CODE END 0 */
```

Inside the main function, from documentation, we know that the PWM frequency satisfies:

$$\nu_{PWM} = \frac{\nu_{Clock}}{(ARR+1)\cdot(PSC+1)} \qquad \Rightarrow \qquad ARR = \frac{\nu_{Clock}}{\nu_{PWM}(PSC+1)} - 1 \tag{24}$$

Where ν_{PWM} is the desired frequency for the PWM generation, ν_{Clock} in our case is $32MHz$ and PSC and ARR are the parameter we will configure in the code to obtain our desired ν_{PWM}

```
/* USER CODE BEGIN 1 */
    //################################PWM OUTPUT#################################
    ARR4 = (3200000/Frequency_output)-1;
/* USER CODE END 1 */
```

Moreover, we know also from documentation that:

$$\text{DutyCycle}_{PWM} = \frac{CCR4}{ARR4} \tag{25}$$

Where the coefficient 4 is as our channel used is the *Channel 4*. Then, we will have to initialize the PWM generation and declare the desired duty cycle:

```
/* USER CODE BEGIN 2 */
    //################################PWM OUTPUT#################################
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_4); //Generation initialization.
    TIM2->CCR4 = (Duty_output*ARR4);//Duty Cycle configuration.
/* USER CODE END 2 */
```

Finally, note that if after coding you save from the .ioc file, the coding modifications will be removed and rewritten to default ones. To avoid this, remember to:

```
/* USER CODE BEGIN TIM2_Init 1 */
//################################PWM OUTPUT#################################
        //Remember to change htim2.Init.Period = ARR4;
/* USER CODE END TIM2_Init 1 */
```

Now you are ready to generate a PWM output, and you could check variables work at *live variables* in STM32CubeIDE.✓

25/04/2023

When joining the PWM input and generation, and ADC convertion for the sensor input, I was given this problem:

```
/opt/st/stm32cubeide_1.12.0/plugins/com.st.stm32cube.ide.mcu.externaltools.gnu-
tools-for-stm32.10.3-2021.10.linux64_1.0.200.202301161003/tools/bin/../lib/gcc
/arm-none-eabi/10.3.1/../../../../arm-none-eabi/bin/ld: prova1outputPWM.elf sec
tion `.text' will not fit in region `FLASH'

/opt/st/stm32cubeide_1.12.0/plugins/com.st.stm32cube.ide.mcu.externaltools.gnu-tools-
for-stm32.10.3-2021.10.linux64_1.0.200.202301161003/tools/bin/../lib/gcc/arm-none-eabi
/10.3.1/../../../../arm-none-eabi/bin/ld: region `FLASH' overflowed by 940 bytes

collect2: error: ld returned 1 exit status

make: *** [makefile:64: prova1outputPWM.elf] Error 1

"make -j12 all" terminated with exit code 2. Build might be incomplete.
```

I found some support pages, but as the problem is about memory, the only solution is to remove something from the code.

Otherwise, it is possible to **release** instead of debugging, as this uses the flash memory, but the con is that it obviously doesn't allow to

26/04/2023

7.2.5 ADC conversion and PWM input and output simultaneously

By releasing and establishing relations between PWM input and output, and the ADC converter, I made all three work at the same time. The project makes a mapping from the [minimum to maximum measured values of the Voltage of the sensor] to the PWM generated:

V_min_measured → PWM=0 V_max_measured → PWM=1

1. First configure the Debug serial wire as always. And also configure the clock to be at 32MHz.
2. Configure the ADC conversion. Tick the IN4 box, and in parameter settings set the external trigger Conversion Source as the Regular launched by software, the samplin Time to 1.5 cycles and the clock preescaler to Syncronous clock mode divided by 2. It's important to note that Continuous Conversion must be disabled. Also tick the NVIC interrupt function box.
3. The timer2 will be set for PWM generation. Select the internal clock as the source, and select the channel 4 for PWM Generation on CH4, which will be the pin PA3. Set the prescaler to 10-1 and let the ARR be 65535 as we will configure in the coding part. Enable auto-reload preload.
4. The timer21 will be set for the PWM input on PA9. Select the internal clock as the source and slect PWM INput on CH2 in the Combined Channels. Make sure the output pin selected is the PA9. In parameter settings, let the preeescaler be 16-1 to avoid overfloating on the timer counting, so also let the ARR4 be

maximum value possible, in our case 65535. As the main channel is CH2, let CH1 Polarity Selection to Falling Edge, and note that the IC Selection is by default Indirect. For CH2, Polarity Selection must be Rising edge as it is our main channel, and IC selection must be Direct. Tick the NVIC global interrupt box.

Now we will set the manual software coding part. Make sure the ADC conversion is not in continues mode (in ADC settings, Parameter Settings, disable Continuous Conversion), and also make sure you reinitialize the ADC conversion, in an infinit loop (the *interrupt functions* or the *while true* inside the main)

```
/* USER CODE BEGIN 0 */
//################################ADC INPUT SENSOR#################################
uint16_t V_meas = 0;
uint16_t V_max = 0;
uint16_t V_min = 65535;
//################################PWM OUTPUT#################################
uint32_t Frequency_output = 400; //in Hertz
float Duty_output = 0.57; //From 0 to 1
uint32_t ARR4 = 0; //Counter Period(Auto Reload Register) see TIM2 Parameter Settings.
//############################## PWM INPUT#################################
int bool = 0; //0-> false 1->true
float Duty_input = 0; //Duty cycle coefficient: 0->0%, 1->100%
uint32_t PWM_input_period = 0; //Period of the PWM function (counts timer clicks)
uint32_t PWM_input_pulseON = 0; //Pulse width, of the positive part, on.
uint32_t Frequency_input = 0;//Frequency of the PWM hole function (Hz)
/* USER CODE END 0 */

/* USER CODE BEGIN 1 */
//################################PWM OUTPUT#################################
ARR4 = (3200000/Frequency_output)-1;//So as to make the output frequency be the desired one.
/* USER CODE END 1 */

/* USER CODE BEGIN 2 */
//################################PWM INPUT#################################
// Main channel (Function Period Measure):
HAL_TIM_IC_Start_IT(&htim21, TIM_CHANNEL_2);
// Indirect channel (Pulse Width Measure):
HAL_TIM_IC_Start(&htim21, TIM_CHANNEL_1);
//################################PWM OUTPUT#################################
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_4); //Generation initialization.
TIM2->CCR4 = (Duty_output*ARR4);//Duty Cycle configuration.
//################################ADC INPUT SENSOR#################################
HAL_ADC_Start_IT(&hadc);
/* USER CODE END 2 */

/* USER CODE BEGIN TIM2_Init 1 */
//################################PWM OUTPUT#################################
```

```
    //Remember to change htim2.Init.Period = ARR4
/* USER CODE END TIM2_Init 1 */

 /* USER CODE BEGIN 4 */
float max(float a, float b){if(a>b) return a; else return b;}
float min(float a, float b){if(a>b) return b; else return a;}
float linear_map (float x, float minIN, float maxIN, float minOUT, float maxOUT){
    //LINEAR MAPPING from [minIN, maxIN] to [minOUT, maxOUT]
    return (x-minIN)*(maxOUT-minOUT)/(maxIN-minIN)+minOUT;}

//###################ADC INPUT SENSOR INTERRUPT FUNCTION##########################
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc1)
{
    V_meas = HAL_ADC_GetValue(hadc1);
    if(V_meas > V_max){V_max=V_meas; Duty_output = Duty_input; bool = 1;}
    if(V_meas < V_min){V_min=V_meas; bool = 0;}

    //Rewrite the duty output:
    if(bool==0){Duty_output = linear_map(V_meas, V_min,V_max, 0, 1);}
    TIM2->CCR4 = (Duty_output*ARR4);//Duty Cycle configuration.
}
//####################PWM INPUT INTERRUPT FUNCTION #####################
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim){
    // If the interrupt is triggered by channel 2
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2){
        // Read the IC value
        PWM_input_period = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
        //If it´s not the initial 0 capture (=>no pulse registered)
        if (PWM_input_period != 0){
            PWM_input_pulseON = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
            // Compute the Duty Cycle
            Duty_input = (float)(PWM_input_pulseON)/(float)PWM_input_period;
            //Compute the frequency of the PWM. 32000000 is the timer frequency
            //Frequency_input = 32000000/(PWM_input_period);//Divide also by prescaler if us
        }
    }
    //As we are not in continues conversion mode, need to reinitialize:
    HAL_ADC_Start_IT(&hadc);
}
/* USER CODE END 4 */
```

And if the FLASH memory is overfilled, release instead of debugging.

Next step is to change the control by *Duty cycle* to *Pulse length* as the protocol of comunication of the ESC is this way. For our drone, the comunicating protocols are:

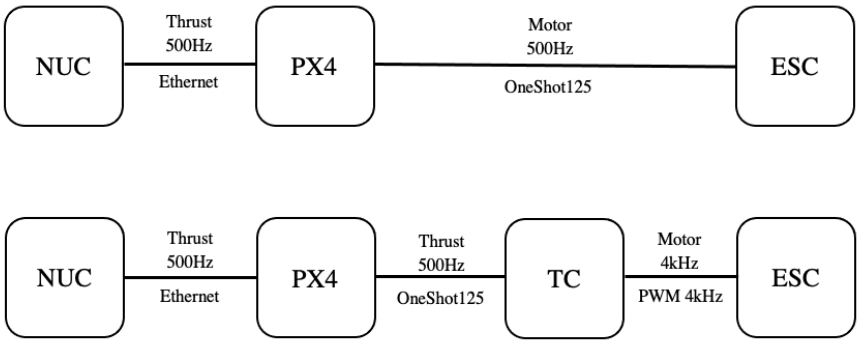


Figure 38: Previous and current comunication protocols sketch

See ESC comunicating protocols. In general, for ESC in drones:

Protocol	Rest Mode - Maximum Thrust input pulse
Standard PWM	1ms-2ms
OneShot125	125μs-250μs
OneShot42	42μs-84μs
MultiShot	5μs-25μs
DShot150	106.8μs
DShot300	53.4μs
DShot600	26.7μs
DShot1200	13.4μs
ProShot	

So we will change our projects to be *pulse-dependent*, instead of *PWM-dependent*.

7.2.6 Calibration

As the PWM/pulse used by the drone in flight mode is over the half of the available range of pulses, so they are only positive values, or a region around here, we will set the calibrating mode in some specific part of the other free interval so that by communicating with this pulse-lengths we will be able to make the Thrust-Controller know it is in calibration mode. We will also add an secure interval for safety reasons, in which the drone can't be moving. For example, we could measure the In the following image, you will see this design from 0 to maximum pulse lengths available for the comunication protocol:

Figure 39: Calibration Design Pulse length

As with two only points we can compute the offset and the rope values, and therefore we will be able to map the voltage to the measured thrust. So we can now implement this on the software. We must configure this regions of the total possible length that could arrive through the communication protocol.

7.2.7 Remainance of the programme in the PCB

Once loaded the programme on the PCB, it starts to execute the program. If we disconnect the debugger wire, nothing happens, as the program is stored in the fast memory of the PCB. However, as it is not stored in the flash memory, if we switch off the power supply, and turn it on, the program is obviously lost. It's time to fix this!

First thing I found was this explanation.

The problem was that if you switch the power supply off with the debugging wire (ST-link) connected from the PCB to the computer, and then reconnect the power supply, I think, as the PCB is connected to the computer, the debugger wire makes a reset of theflash memory so you can rewrite on it with the computer.

Another fact is that it is possible when resupplying the PCB with power, if the PWM-input is not connected and working, the program does not initialize properly. The solution is just to connect the PWM input working and switch off and on the power supply, then the program should be executed properly.

Notice global variables modified during the execution will reset when disconnecting the power supply.

7.2.8 PID

Last thing will be to implement the PID. Before programming it, the sketch of the definitive stm32 project for thrust control is:

Configure the settings for PWM input and output and for ADC as shown before. First of all, we will declare all the global variables that will be used along the program. For the PID, we will need the desired thrust or command that will be received from the PX4, and the real thrust measured by the sensor. Once made the PID controller, we will receive a new thrust value that must be sent ot the ESC, for this output, we must define the frequency of the PWM, also the pulse length that will be sent, as the protocol OneShot25 transforms the , and as an auxiliary variable, the duty cycle which matches the frequency of the PWM and the length of the cycle. This last one may be deleted afterwards for memory optimization. The relation of the duty cycle and the intern variable for is the AutoReload-Preload

```
/* USER CODE BEGIN 0 */
    //#####################ADC INPUT SENSOR - PID CONTROL#######################
    float Thrust_measured = 0;//Newtons
    float Thrust_desired = 0;//Newtons
    //################################PWM OUTPUT#################################
    //Frequency in Hertz. As we want a max pulse of 250us(Duty_output=100%)
    uint32_t Frequency_output = 4000;
    float Duty_output = 0; //From 0 to 1
    uint32_t Pulse_length_output = 150;//in us (microseconds)
    //Counter Period(Auto Reload Register) see TIM2 Parameter Settings.
    uint32_t ARR4 = 0;
    //############################## PWM INPUT#################################
    int bool = 0; //0-> false 1->true
    float Duty_input = 0; //Duty cycle coefficient: 0->0%, 1->100%
    uint32_t PWM_input_period = 0; //Period of the PWM function (counts timer clicks)
```

```
    uint32_t PWM_input_pulseON = 0; //Pulse width, of the positive part, on.
    uint32_t Frequency_input = 0;//Frequency of the PWM hole function (Hz)
    //###########################################################################
/* USER CODE END 0 */

/* USER CODE BEGIN 1 */
    //################################PWM OUTPUT#################################
    //So as to make the output frequency be the desired on
    ARR4 = (3200000/Frequency_output)-1;
/* USER CODE END 1 */

/* USER CODE BEGIN 2 */
    //################################PWM INPUT##################################
    // Main channel (Function Period Measure):
    HAL_TIM_IC_Start_IT(&htim21, TIM_CHANNEL_2);
    // Indirect channel (Pulse Width Measure):
    HAL_TIM_IC_Start(&htim21, TIM_CHANNEL_1);
    //################################PWM OUTPUT#################################
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_4); //Generation initialization.
/* USER CODE END 2 */

/* USER CODE BEGIN TIM2_Init 1 */
    //################################PWM OUTPUT#################################
    //Remember to change htim2.Init.Period = ARR4
/* USER CODE END TIM2_Init 1 */

/* USER CODE BEGIN 4 */
    float max(float a, float b){if(a>b) return a; else return b;}
    float min(float a, float b){if(a>b) return b; else return a;}
    float linear_map (float x, float minIN, float maxIN, float minOUT, float maxOUT){
    //LINEAR MAPPING from [minIN, maxIN] to [minOUT, maxOUT]
    return (x-minIN)*(maxOUT-minOUT)/(maxIN-minIN)+minOUT;}
    //FAKE PID function for testing:
    float PID(float Thrust_desired, float Thrust_measured){
        float Thrust_out = -5;//Newtons
        return Thrust_out;
    }

    //#################### ADC CONVERSION and PID control ####################
    //which is the frequency of the call of this function??
    void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc1)
    {
        //ADC conversion:------------------------- at rest: 1650 aprox.
        Thrust_measured = linear_map(HAL_ADC_GetValue(hadc1), 900, 3500, -5, 25);
        //900 and 3000 need calibration, as the ADC input value is not estandard
        //PID control ----------------------------------------------
        float Thrust_PID = PID(Thrust_desired, Thrust_measured);
```

```
        Pulse_length_output = linear_map(Thrust_PID, -5, 25, 100, 250);
        //100 needs calibration (drone weight).
        //Output generation----------------------------------------
        Duty_output=linear_map(Pulse_length_output, 0,250, 0, 1);
        //Duty Cycle configuration. ARR4 as Frequency_out is 4kHz
        TIM2->CCR4 = (Duty_output*ARR4);
    }

    //####################PWM INPUT INTERRUPT FUNCTION ####################
    //It's called each 10*10000/32000000=3.215ms, as the input PWM frequency is 400Hz
    void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim){
        // If the interrupt is triggered by channel 2
        if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2){
            // Read the IC value
            PWM_input_period = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
            //If it´s not the initial 0 capture (=>no pulse registered)
            if (PWM_input_period != 0){
                PWM_input_pulseON = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
                // Compute the Duty Cycle
                Duty_input = (float)(PWM_input_pulseON)/(float)PWM_input_period;
                //Computing the Thrust desired.
                Thrust_desired = linear_map(Duty_input, 0, 1, 125, 250);
                //125 and 250 depend on the comunication protocol.
            }
        }
        //As we are not in continues conversion mode, need to reinitialize:
        HAL_ADC_Start_IT(&hadc);
    }/* USER CODE END 4 */
```

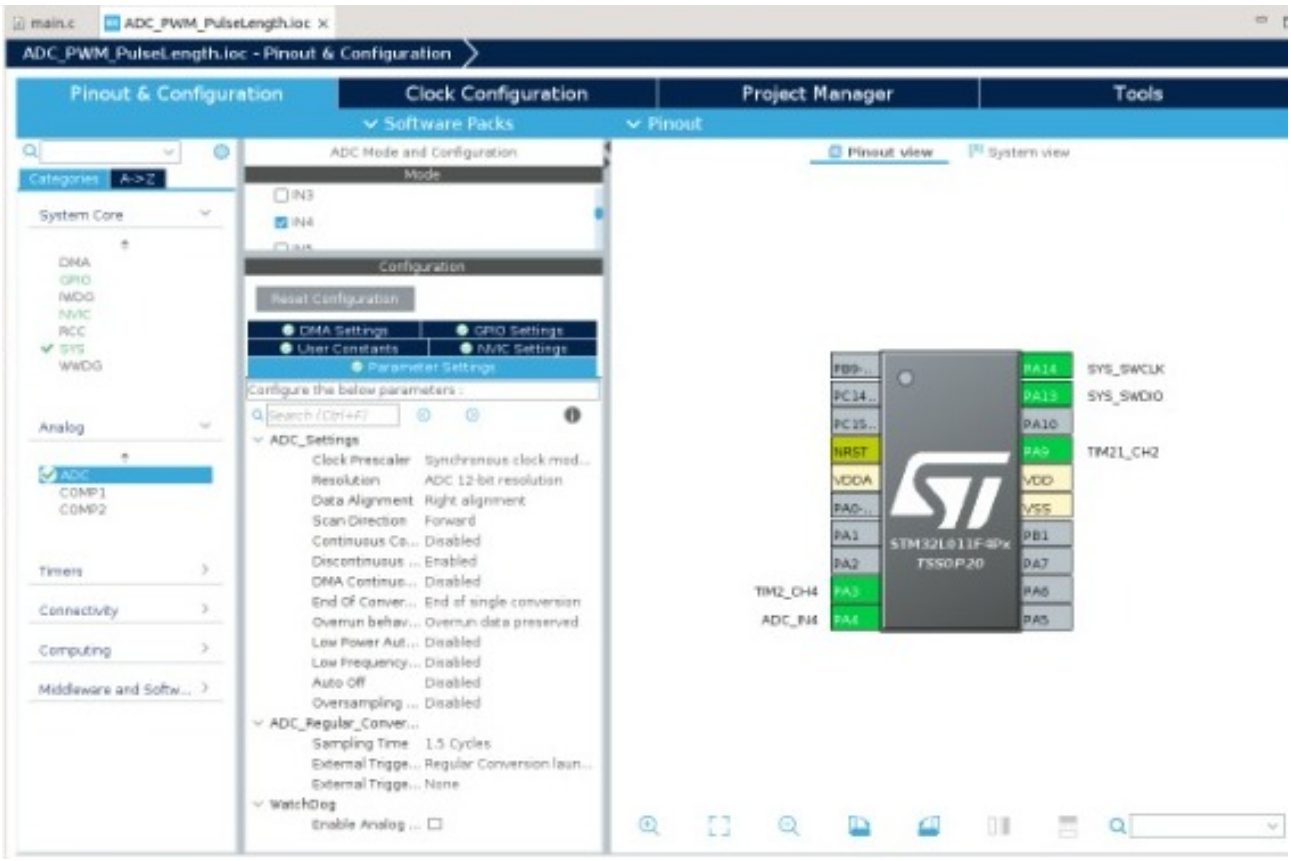
Note that PID function needs to be programmed, and also that there are several values that need to be calibrated due to offsets, ropes, and protocol comunications.

28/04/2023

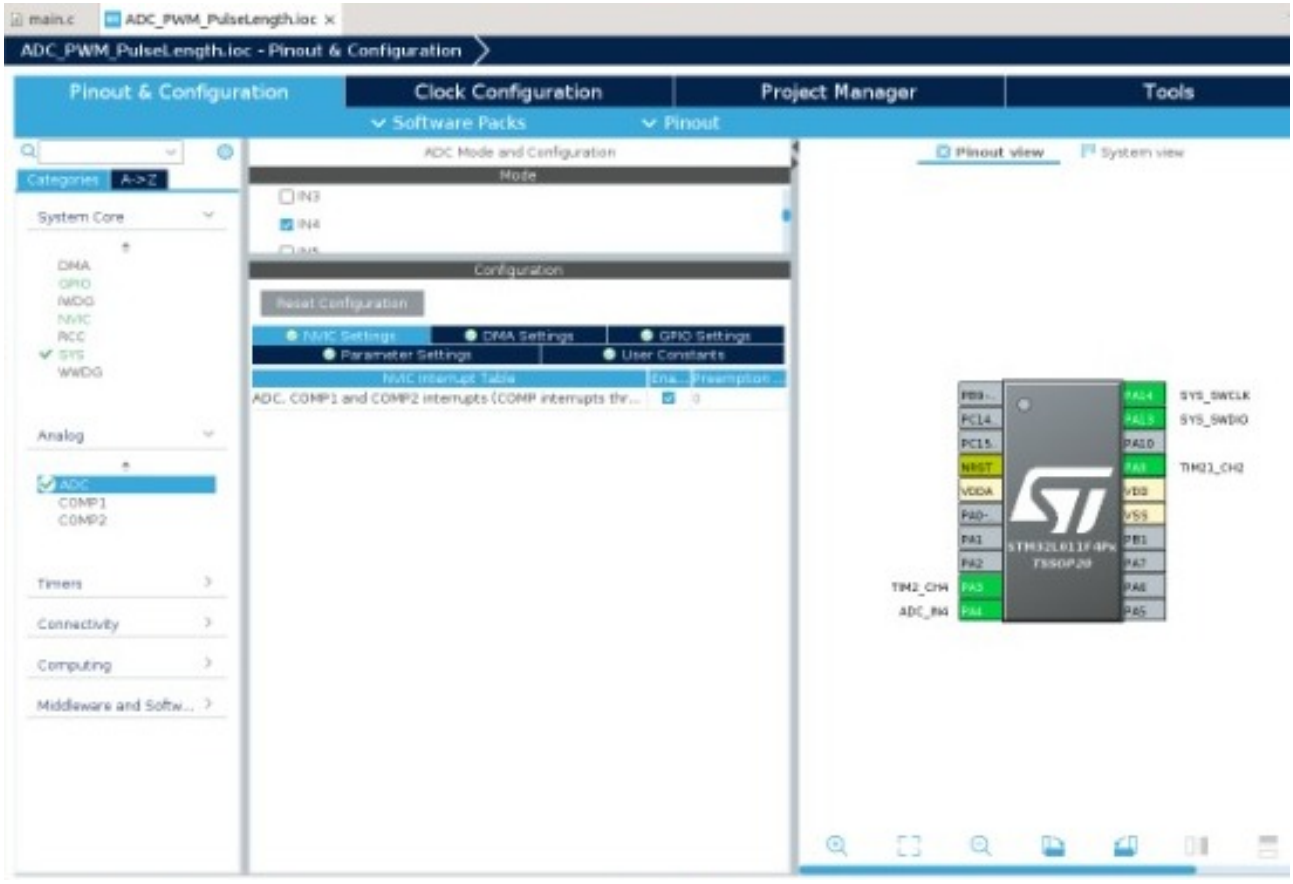
For the development of the PID algorithm you could follow this tutorial. However, it's compulsury to have a good idea first of Control Theory. You could also implement an auto-tunning algorithm. For tunning, we will use Ziegler-Nichols method. For several PIDs comparison, see this article

2/05/2023

So as to read the sensor value at 4kHz (before we were doing it at 400Hz, at timer 21 interrupt function), I set up an extra interrupt function associated for timer 2 which is called at a frequency of 4kHz and calls the ADC interrupt. Probably we could only use the timer one.

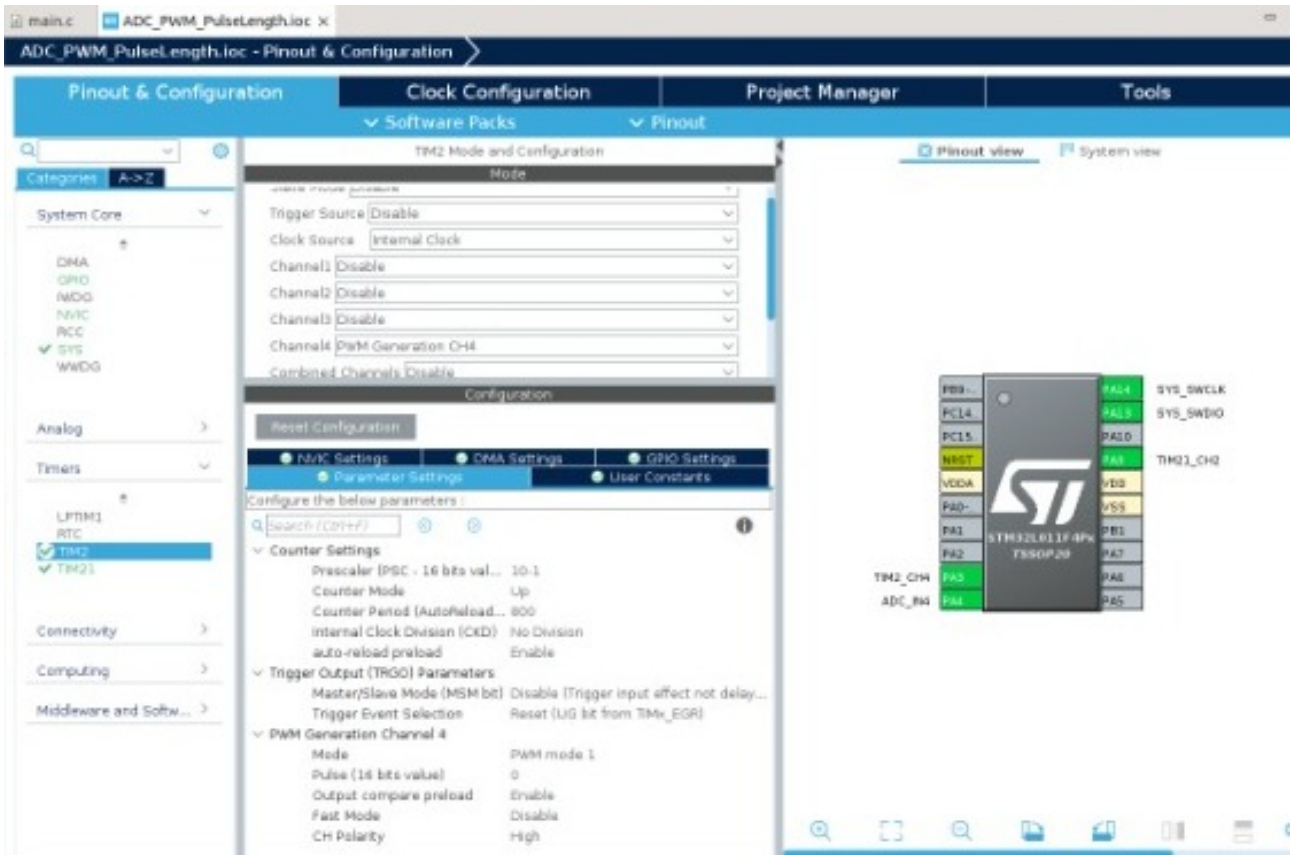


(a) ADC Parameter Settings

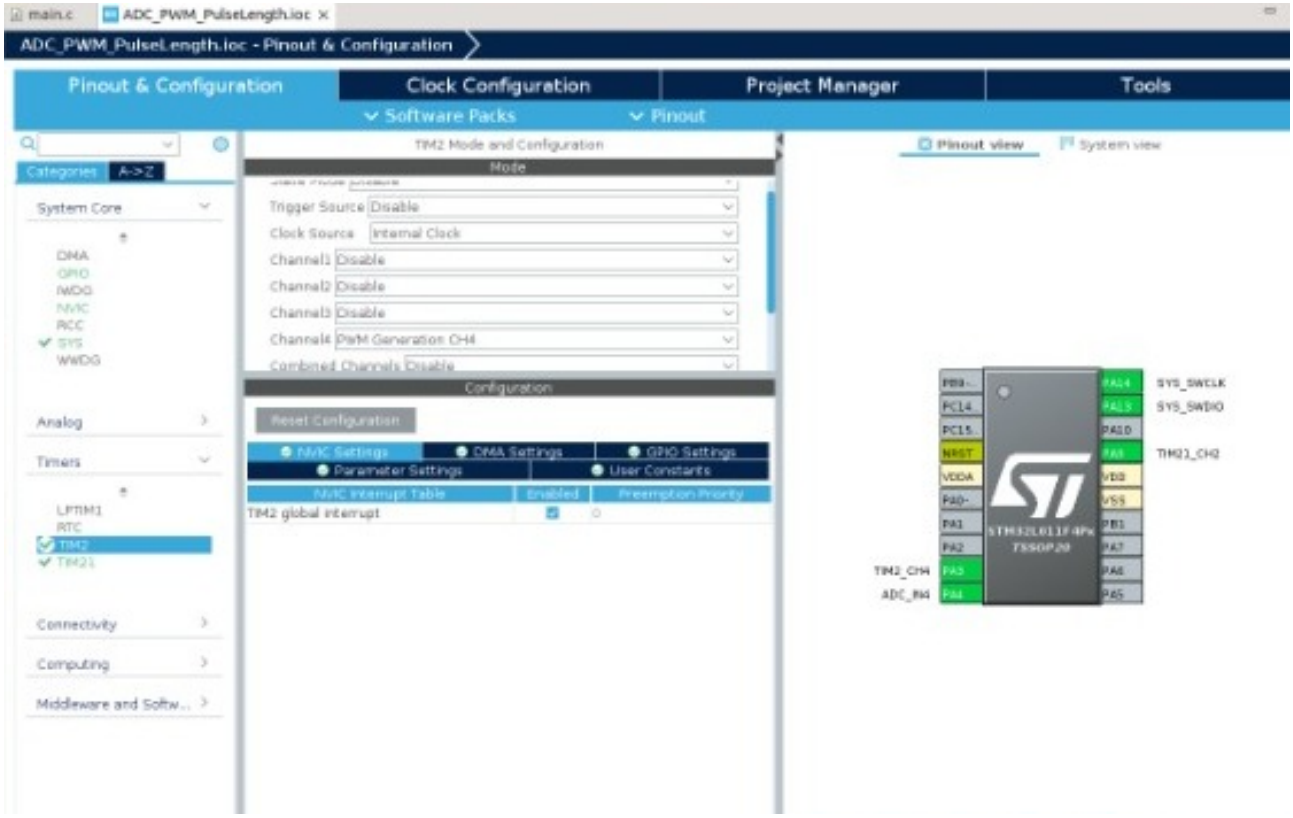


(b) ADC NVIC Settings.

Figure 40: Set-up for sensor ADC input.

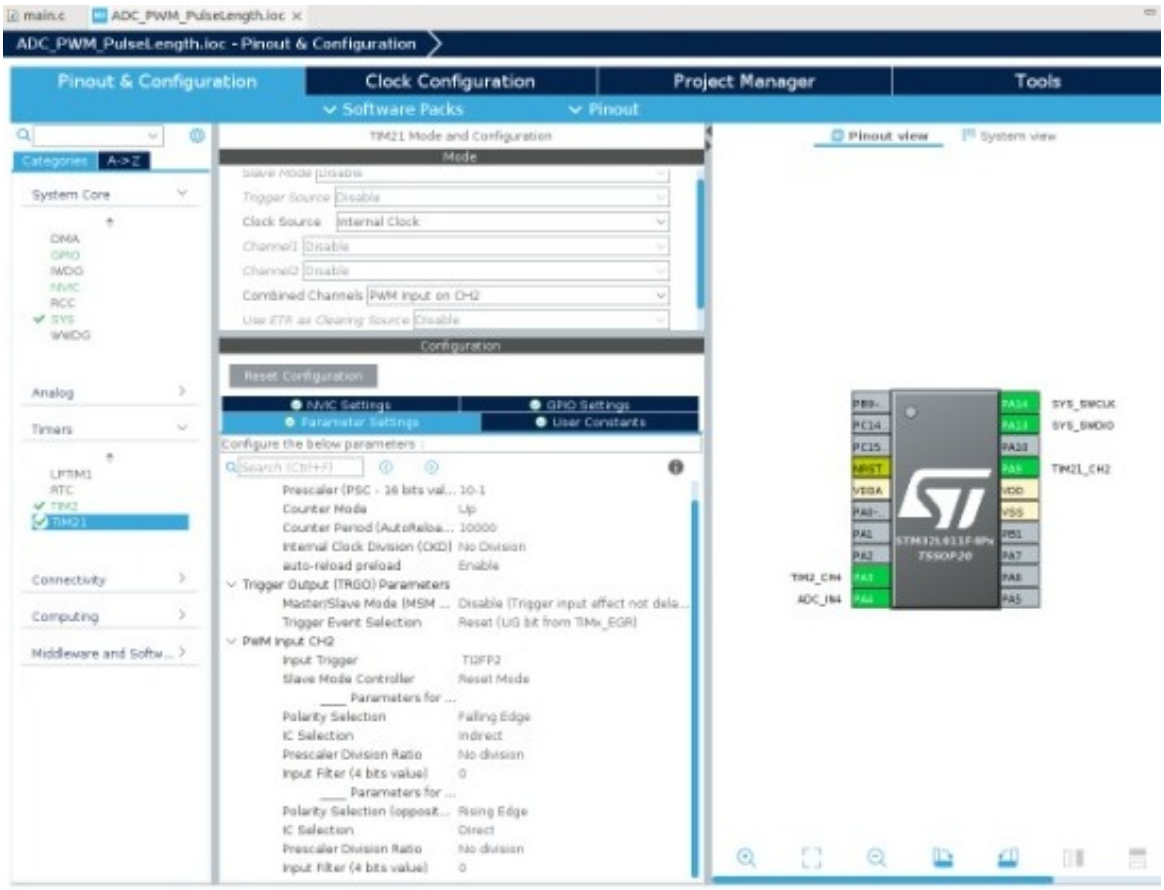


(a) TIM2 Parameter Settings

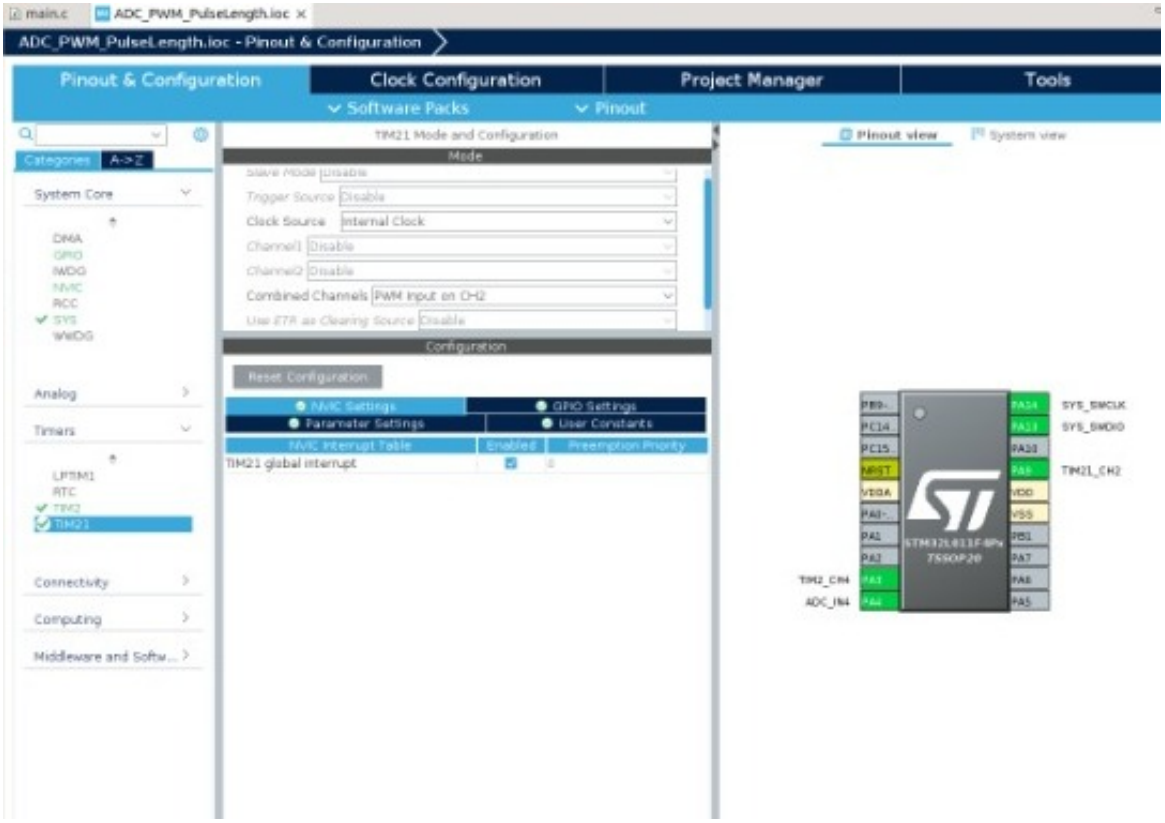


(b) TIM2 NVIC Settings.

Figure 41: Set-up for PWM generation.



(a) TIM21 Parameter Settings



(b) TIM21 NVIC Settings.

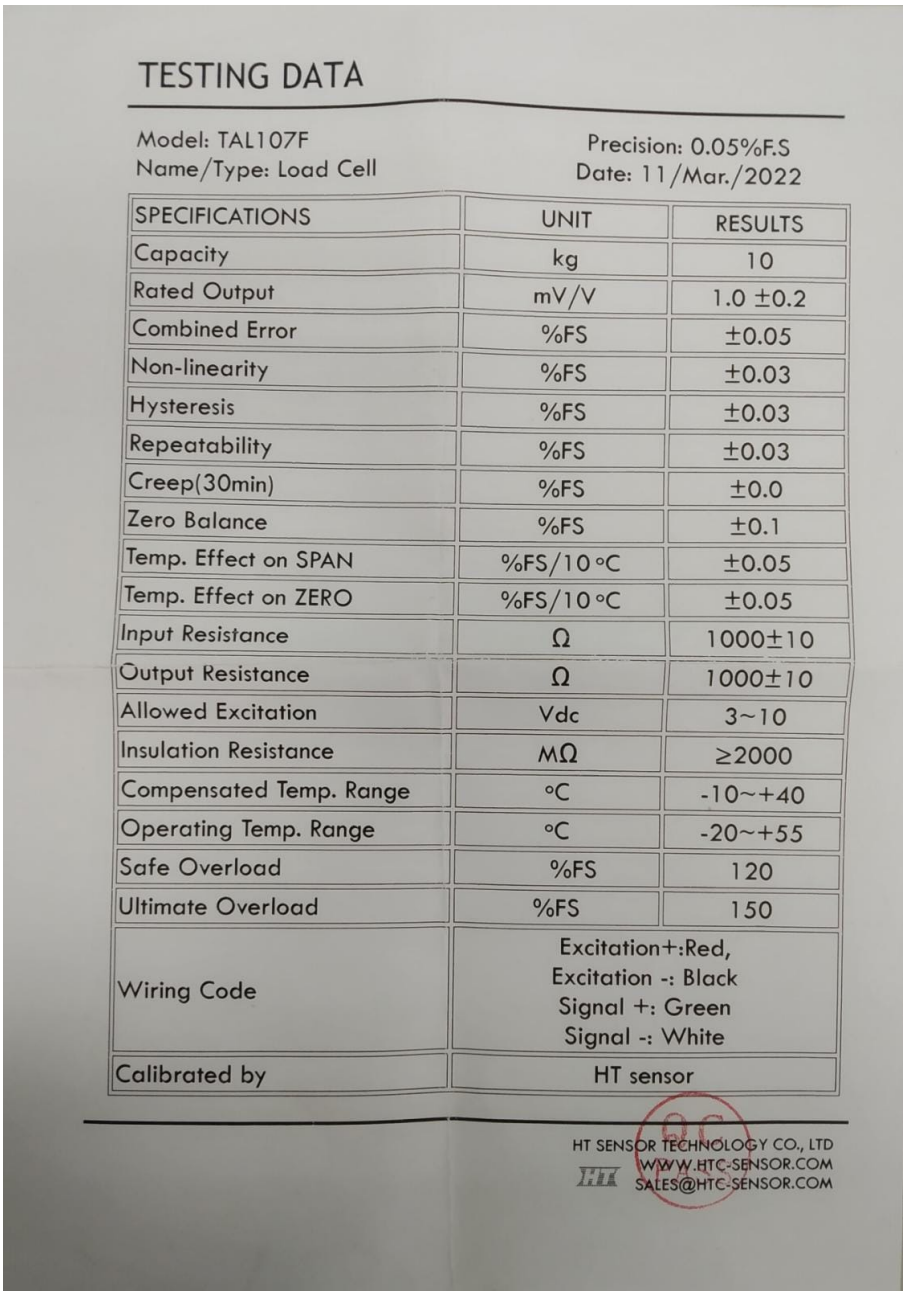
Figure 42: Set-up for PWM reading.

```
    /* USER CODE BEGIN 0 */
        //#########################ADC INPUT SENSOR - PID CONTROL##############
        float Thrust_measured = 0;//Newtons
        float Thrust_desired = 0;//Newtons
        //################################PWM OUTPUT#################################
        uint32_t Frequency_output = 4000;
    //in Hertz. As we want a max pulse of 250us(Duty_output=100%)
        uint32_t Pulse_length_output = 150;//in us (microseconds)
    float Duty_output = 0; //From 0 to 1. Will be computed according to the pulse length
        //############################## PWM INPUT#################################
        float Duty_input = 0; //Duty cycle coefficient: 0->0%, 1->100%
        uint32_t PWM_input_period = 0; //Period of the PWM function (counts timer clicks)
        uint32_t PWM_input_pulseON = 0; //Pulse width, of the positive part, on.
        uint32_t Frequency_input = 0;//Frequency of the PWM hole function (Hz)
        //#########################################################################

/* USER CODE END 0 */

    /* USER CODE BEGIN 2 */
            //################################PWM INPUT#################################
            // Main channel (Function Period Measure):
            HAL_TIM_IC_Start_IT(&htim21, TIM_CHANNEL_2);
            // Indirect channel (Pulse Width Measure):
            HAL_TIM_IC_Start(&htim21, TIM_CHANNEL_1);
          //################################PWM OUTPUT#################################
          HAL_TIM_PWM_Start_IT(&htim2, TIM_CHANNEL_4); //Generation initialization.
  /* USER CODE END 2 */

    /* USER CODE BEGIN 4 */
float max(float a, float b){if(a>b) return a; else return b;}
float min(float a, float b){if(a>b) return b; else return a;}
float linear_map (float x, float minIN, float maxIN, float minOUT, float maxOUT){
//LINEAR MAPPING from [minIN, maxIN] to [minOUT, maxOUT]
return (x-minIN)*(maxOUT-minOUT)/(maxIN-minIN)+minOUT;}


float integration = 0;
float previous_error = 0;
float PID(float Thrust_desired, float Thrust_measured){
        //To be tunned:-----------------------------
        const float Kff = 0; //Constant for Proportionality directly over desired thrust
        const float Kp = 0; //Constant of proportionality over error.
        const float Ki = 0; //Constant of integration
        const float Kd = 0; //Constant of derivation
        //-----------------------------------------------
        float error = Thrust_desired - Thrust_measured;
```

```
        float dt =250/1000000;//250us, microseconds. ADC interrupt function period.
        float derivation = (error-previous_error)/dt;
        integration += error*dt;
        //PID
        float Thrust_output = (Kff * Thrust_desired)+(Kp*error)
        +(Kd*derivation)+(Ki*integration);

        //ANTI-WIND-UP SHIELD. Avoids saturation--------------
        if(Thrust_output > 25){
                integration -= error;
                Thrust_output = 25;//Upper bound
        }
        if(Thrust_output < 0){
                integration -= error;
                Thrust_output = 0;//Lower bound
        }
        //-------------------------------------------------
        previous_error = error;
        return Thrust_output;
}

//################### ADC CONVERSION and PID control ####################
//Frequency of 4kHz as it is initialized in tim2 interrupt HAL_ADC_Start_IT(&hadc);
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc1)
{
        //t_ADC = (t_ADC + 20)%65535;
        //ADC conversion:-----------------------------------------
        Thrust_measured = linear_map(HAL_ADC_GetValue(hadc1), 1650, 3300, 0, 25);
        //1650 and 3000 need calibration, as the ADC input value is not estandard,
    //at rest: 1650 aprox.

        //PID control ---------------------------------------------
        float Thrust_PID = PID(Thrust_desired, Thrust_measured);
        //Thrust_PID = Thrust_measured;//test if ADC reading is working.
        //Thrust_PID = Thrust_desired;//test if PWM generation is working.

        Pulse_length_output = linear_map(Thrust_PID, 0, 25, 125, 250);
    //125 needs calibration (helix weight).

        //Output generation---------------------------------------
        Duty_output=linear_map(Pulse_length_output, 0,250, 0, 1);
        TIM2->CCR4 = (Duty_output*800);//Duty Cycle configuration.
    //ARR4=800 as Frequency_out is 4kHz
}
```

```
//####################PWM INPUT INTERRUPT FUNCTION ####################

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim){
    // Check which version of the timer triggered
    if(htim == &htim21)
    { //It's called each 10*10000/32000000=3.215ms, as the input PWM frequency is 400Hz
        // If the interrupt is triggered by channel 2:
        if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2){
            // Read the IC value
            PWM_input_period = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
            //If it´s not the initial 0 capture (=>no pulse registered)
            if (PWM_input_period != 0){
                PWM_input_pulseON = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
                // Compute the Duty Cycle
                Duty_input = (float)(PWM_input_pulseON)/(float)PWM_input_period;
                //Compute the frequency of the PWM. 32000000 is the timer frequency
                //Frequency_input = 32000000/(PWM_input_period);
                //Divide also by prescaler if used

                //Computing the Thrust desired.
                Thrust_desired = linear_map(Duty_input, 0.5, 1, 0, 25);
                //Depends on the comunication protocol?
           }
        }
        //Duty_output = Duty_input;//Test if PWM read is working.
    }
}
void HAL_TIM_PWM_PulseFinishedCallback(TIM_HandleTypeDef* htim)
{//Every 250us (microseconds) as the frequency is 400kHz
    if(htim == &htim2)
    {//ADC conversion initialization:-------------------------------------------
        HAL_ADC_Start_IT(&hadc);
        //As we are not in continues conversion mode, need to reinitialize
    }
}
/* USER CODE END 4 */
```

8 Documentation



TESTING DATA

Model: TAL107F
Name/Type: Load Cell

Precision: 0.05%F.S
Date: 11/Mar./2022

SPECIFICATIONS	UNIT	RESULTS
Capacity	kg	10
Rated Output	mV/V	1.0 ±0.2
Combined Error	%FS	±0.05
Non-linearity	%FS	±0.03
Hysteresis	%FS	±0.03
Repeatability	%FS	±0.03
Creep(30min)	%FS	±0.0
Zero Balance	%FS	±0.1
Temp. Effect on SPAN	%FS/10 °C	±0.05
Temp. Effect on ZERO	%FS/10 °C	±0.05
Input Resistance	Ω	1000±10
Output Resistance	Ω	1000±10
Allowed Excitation	Vdc	3~10
Insulation Resistance	MΩ	≥2000
Compensated Temp. Range	°C	-10~+40
Operating Temp. Range	°C	-20~+55
Safe Overload	%FS	120
Ultimate Overload	%FS	150
Wiring Code	Excitation+:Red, Excitation -: Black Signal +: Green Signal -: White	
Calibrated by	HT sensor	

HT SENSOR TECHNOLOGY CO., LTD
WWW.HTC-SENSOR.COM
SALES@HTC-SENSOR.COM

Figure 43: Sensor TAL107F documentation

1. ST-link/V2 documentation.
2. AD620 amplifier documentation.
3. STM32L011F4Px microcontroller documentation