Project2 - Speed controller

Emilie TRAN
François BARNOUIN
Link to the GitHub: https://github.com/Franz1234567/Project2/tree/main

Part 1

Signed motor speed measured (PPS or RPM):

We needed to count how many pulses there were in one second so we added another timer interrupt of 1 second. For this we used timer0. Since this timer is 8 bits, we could not have a clock count above 255. We used a prescaler of 1024:

16000000.0/1024.0 = 15625Hz and 1/15625 = 64us

Thus, one clock tick was of 64us so we chose to count to 125 in the timer (= 8ms). Finally, we waited until this timer was triggered 125 times. When it was reached, it meant that 1 second (= 8ms * 125).

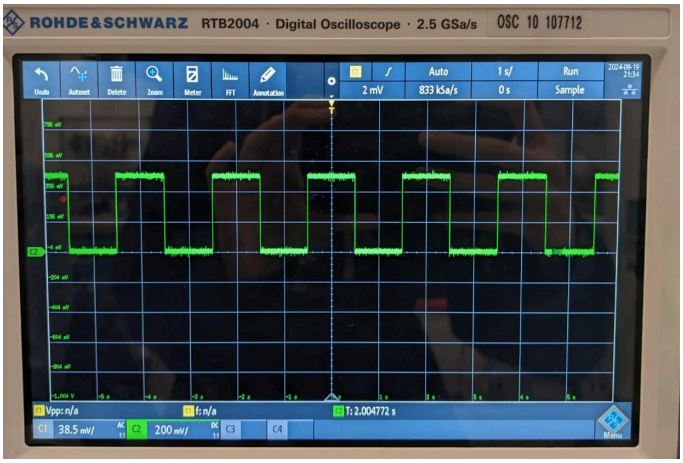
With this, we measured a maximum speed of 2860pps

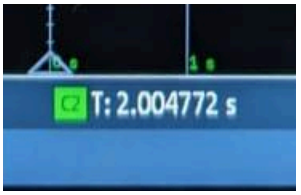
Encoder driver hides hardware:

See the encoder.h and encoder.cpp files for this question.

Stable time base for speed:

In order to check that this time base was stable, we toggled the LED every second and used the oscilloscope to check if everything was all right. Here s a picture of that:





Note: We can read a period of 2s which means that we the LED do the cycle on-off-on in 2 seconds which means that there is indeed a toggle change every second.

Part 2

Max motor speed measured correctly and Time constant measured correctly:

The maximum motor speed that we measured while driving the load (we measured 10 times and did an average) was: 2840pps and the time constant was (doing the same average) of 0.64s and the corresponding speed of 1718pps.

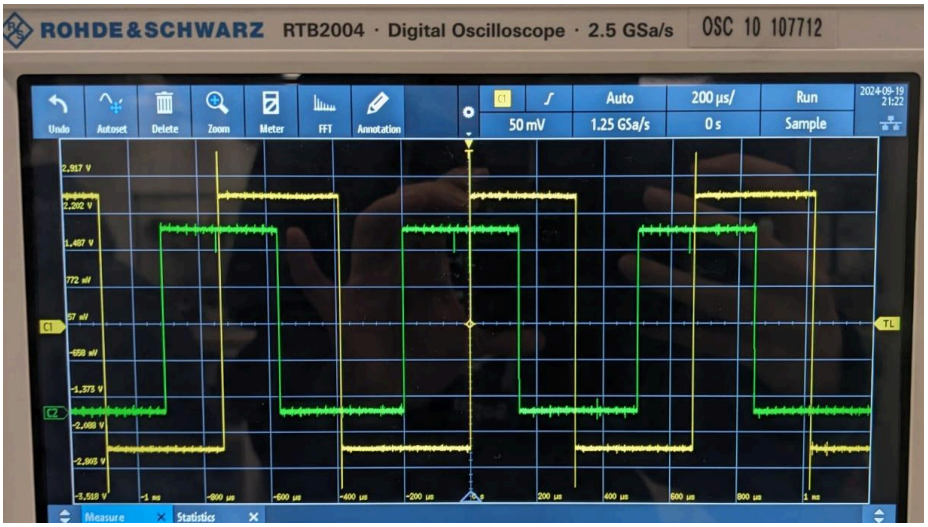
We can notice that when the wheel is attached, the maximum is slightly lower. (but it's not as significant as we were expecting).

Note: To measure the time at 63% we computed the theoretical speed velocity and put in a variable the number of counts when it was equal or higher.

Part 3

Timing requirements, how set them up and validation methods:

task: counting encoder pulses
Timing Requirement: The pulses must be sample at every pulse change
Sampling Rate: Based on the previous project, we set a sampling rate of 280 µs
Method: Using an interrupt with Timer2 to count encoder pulses at 280 µs intervals.
Validation: The oscilloscope shows that the interrupt is triggered accurately every 280 µs and that we don't miss any pulses (see picture below)



Yellow: Encoder - Green: LED

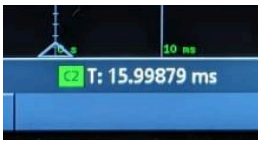
task: computing the motor speed
We have already spoken about this part earlier at part 1.

Timing Requirement: We need to compute motor speed every 1 second to provide a stable time base for the speed controller
Speed Calculation Period and method: This is done over 1 second, with an 8 ms interrupt interval (from Timer0) and counting 125 interrupts in our main to achieve 1 second for a stable velocity measurement

Validation: We confirmed the accuracy of this timing by measuring the 1 second period with an oscilloscope (see picture in part 1) by toggling a LED pin every second.

task: updating the control output

Timing Requirement: The update rate of the speed controller should be no less than 10/t. We measured earlier=0.64 s.
Update Rate: Using Timer0, you've set an update rate of 125 Hz (8 ms period), which is faster than the required minimum of 15.625 Hz (64 ms period).
Method: Using the Timer0 interrupt at 8 ms to trigger control updates.
Validation: Oscilloscope measurements with a LED that toggles every 8ms to confirm that the interrupt is triggered every 8 ms. (see picture below)



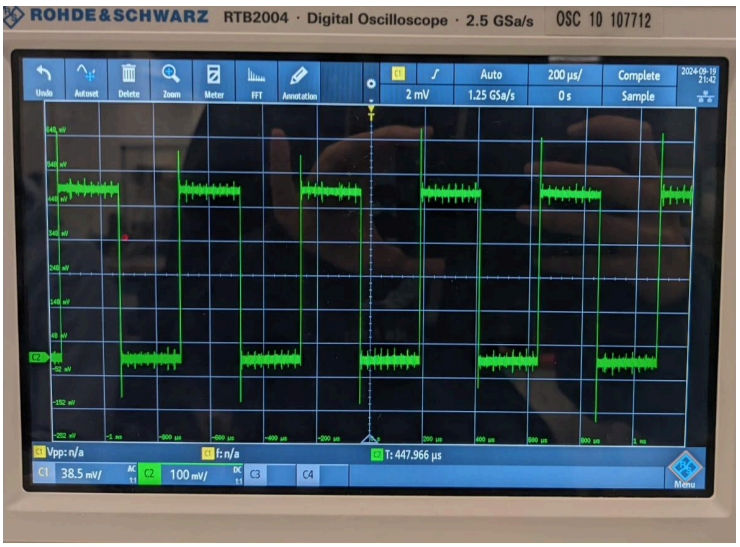
task: Storing and transmitting the response to the laptop

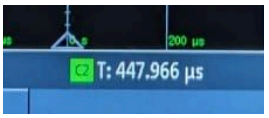
Timing Requirement: As seen in one of the assignments, we need to have a slow frequency for the Serial prints otherwise it can ruin the other interrupts
Transmission Period: We decided to print every second the reference speed, the actual speed and the PWM value. Indeed, doing it faster prevented everything from working
Method: As explained, we used Serial transmission using the Serial.print() function in Arduino
Validation: We ensured that the values were indeed printed in the serial monitor (see video)

task: Stable PWM output at a suitable rate, observing $\frac{1}{f_{\text{pwm}}} \ll \frac{1}{f_{\text{update}}} \leq \frac{\tau}{10}$

Timing Requirement: The equation above
Our values: 1/f_pwm = 450us
1/f_update = 8ms
t/10 = 64 ms

Validation: If we consider that 450us << 8ms, the requirement is met. It was experimentally hard to choose a lower f_pwm because the max speed of the motor was significantly impacted (lower value).





Here is a summary of what we used for this part:

Task	Timing Requirement	Actual Timing	Method Used	Resource Allocation (Timer/ Interrupts)
Counting Encoder Pulses	Not miss any encoder pulse (done in Project 1)	**280 µs** sampling period	Using **Timer2** to generate an interrupt every **280 µs**	**Timer2 (8-bit)**, Prescaler: **1024**, Interrupt every **280 µs**
Computing Motor Speed	Compute the speed velocity every second	**1 second** (calculated using **125 * 8 ms**)	**Timer0** generates interrupts every **8 ms**, and after **125 counts**, speed is calculated	**Timer0 (8-bit)**, Prescaler: **1024**, Interrupt every **8 ms**
Updating Control Output	Respect the given equation: $\frac{1}{f_{PWM}} \ll \frac{1}{f_{update}} \leq \frac{t}{10}$	**8 ms** (actual **125 Hz** update rate)	**Timer0** interrupt triggers the control law update every **8 ms**	**Timer0 (8-bit)**, Prescaler: **1024**, Interrupt every **8 ms**
Storing/ Transmitting Data	Every second	**1 second** (after each speed calculation)	**Serial.print()** transmits data after speed calculation	**UART** for serial communication to the laptop
Generating Stable PWM Output	Respect the given equation: $\frac{1}{f_{PWM}} \ll \frac{1}{f_{update}} \leq \frac{t}{10}$	**450 µs** PWM period (**2.22 kHz**)	**Timer1** generates PWM at **450 µs** period (duty cycle changes based on control output)	**Timer1 (16-bit)**, Prescaler: **64**, PWM period **450 µs**

Part 4

Controller class implemented: See code for this part.

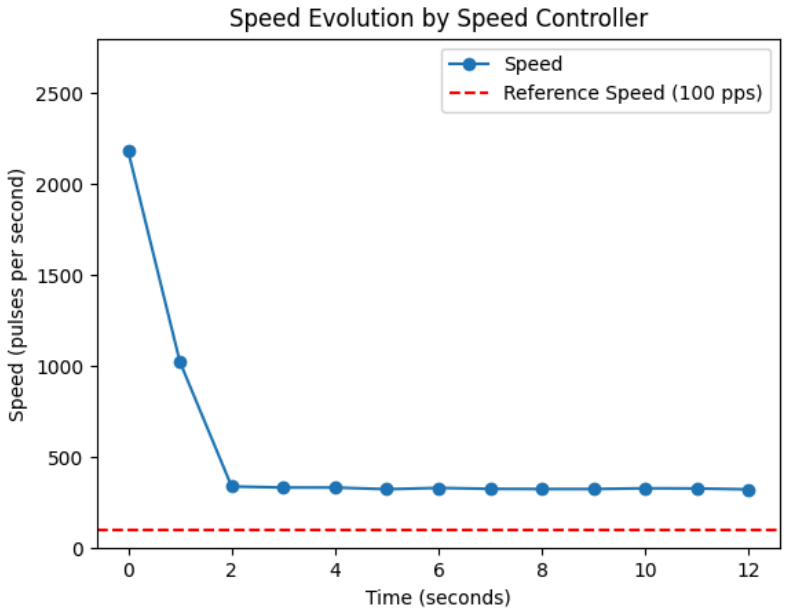
Actual value and control values plotted under load condition (wheel is attached):

To see the values in the serial monitor, please see the video. As mentioned before we were highly restrained in the number of values we could do (doing more prints as you can see in the video resulted in uncontrolled behaviors).

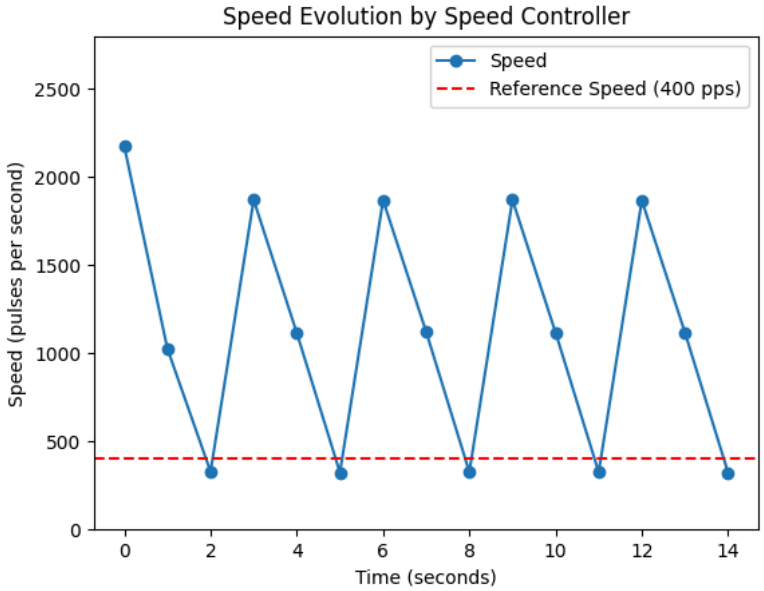
Here are two plots of the motor with a Kp of 0.05. The first one had a reference value of 100pps and the second one a reference value of 400 pps. In both plots, the duty cycle is initially set to 100 then it is changed by the controller.

The two plots show the actual motor speed response over time when subjected to the proportional controller.

The form of the first plot vaguely resembles a step response (an inverted one), however we can obviously see that there are not enough values plotted (for instance to measure a t 63%). Moreover we can see that since the motor does not manage to reach the reference speed, it just tries to go as close to it as possible.



The second plot shows the motor going at 100% at the beginning and then it oscillates between above and under the reference point. So when the speed measured is around 1700 pps, the error slows progressively but when it's close to the reference speed, it goes up to a high speed again. This result is unexpected and it proves that we lacked in the choice of our parameter or in our implementation.


Speed Evolution by Speed Controller
Speed
Reference Speed (400 pps)
Speed (pulses per second)
Time (seconds)
0
500
1000
1500
2000
2500
0
2
4
6
8
10
12
14