DC motor controller: speed control

Embedded Systems CSEE 4840

Alexandre Msellati (am5692) , Ayman Talkani (aat2193)

12 May 2022

Contents

1 Introduction

Through a quick google search, it is pretty easy to come across microcontroller based projects that claim to perform "motor control". These projects often utilize a small permanent magnet direct-current (PMDC) motor connected to a variable voltage source at its terminals, and prove that they can vary the speed by modifying the voltage. However, while these projects work perfectly fine for controlling DC motors with smaller ratings that are often used by hobbyists, they cannot be applied to motors with higher ratings that are often used in the industry.

In order to mitigate this issue, we will be making use of the DE1 SoC with an FPGA in order to design our own Motor Controller. While motor control designs are well researched, there are many additional advantages when these are designed on FPGAs, such as the ability to use custom pulse-width modulation (PWM), easier component integration, fewer components, higher control-loop bandwidth, and higher reliability. The purpose of our project was to design an actual DC motor controller that takes full advantage of the FPGA's capabilities and can be implemented in actual industrial level applications, where the system and control laws have been thoroughly studied before implementation.

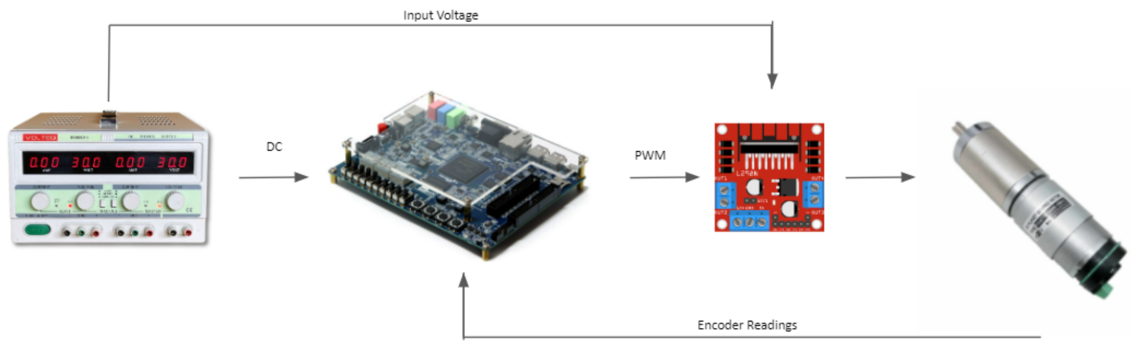


Figure 1: Simplified overview of system componenets

2 Overview of the project

The global system that aims to control the speed of the motor is shown figure 2.Here, the GPIO pins of the FPGA are connected to the pins of the dual full bridge L298N motor driver that controls the output voltage to the gearbox PMDC motor am-2971 by acting as an H-bridge circuit. We also make use of the 8-channel 12-bit A/D LTC2308 Converter that is built into the DE1-SoC in order to read input DC voltage values supplied by the user in mV. The ADC pins are also connected to a voltage divider.

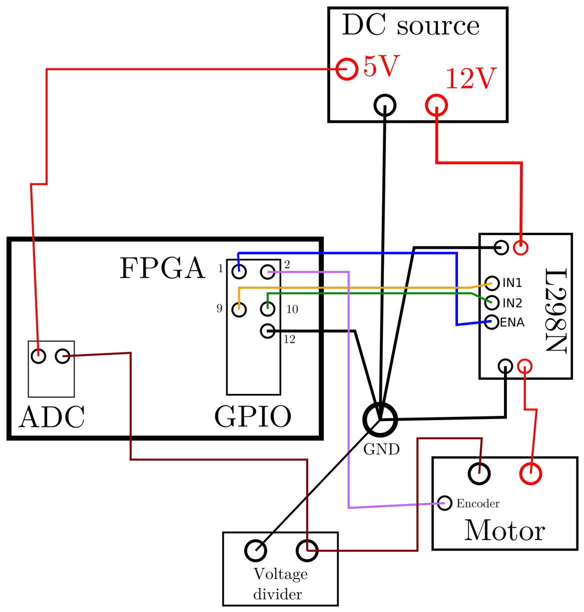


Figure 2: Schematic of the global system.

By choosing the voltage issued from the DC source, the user can control the speed of the motor. The voltage divider is used for an estimation of the current, that feeds a current controller defined in software. The GPIO pin 0 is connected to the pin ENA, that turns on or off the output voltage for the DC motor. A user interface, built with the switches of the FPGA, enables us to control the output of this pin. The hexadecimal display indicates whether the motor is turned on or off, and is also used to display the values of the input voltage readings as read by the ADC. One switch allows to determine the rotation direction of the motor and another one, to set the mode to manual (the user decides the direction of rotation) or automatic. The GPIO pins 7, 8 and 9 are connected to the ground, the pin IN1 and IN2 respectively. Once ENA is enabled, IN1 and IN2 can be used to control the H-bridge, as well as to implement pulse width modulation, where the duty cycle is determined by the controller that takes the input voltage supplied to the user, and generates a new signal by using the clock signal as reference. The voltage drop at the output terminals of the L298N is 2V, which is significant considering the maximum input voltage of the motor being 12V.

The controller issues a voltage between -12V and 12V that is converted to a duty cycle and rotation direction command, and sent to the hardware that processes it for the pwm.

There were many challenges that we had faced while making this project, such as accurately reading the correct ADC values from the LTC2308 based on its configuration and timing requirement, reading values accurately from the encoder, generating a PWM to appropriately control the motors based on the output from the controller, as well as the interfacing between the Hardware and Software components.

Following our discussion with Professor Stephens after our design document submission, we have removed the driving simulator component and restructured our project milestones as follows:

1. Getting the correct values from the embedded ADC converter.
2. Designing and discretizing our PI controller to work on the FPGA, while also reading values from the motor encoder.
3. Generating an appropriate PWM signal on the GPIO pins of the board, and creating a hardware software interface for our individual components in order to control the motor.

3 Project Design

Here, we go over the detailed project design for our motor controller, detailing with the design of the Hardware and Software components of our project respectively.

3.1 Hardware implementation

3.1.1 ADC

To start off, we will address the task involved in our first checkpoint, which involves making use of the LTC2308 embedded ADC converter on the De1-SoC board. Our goal in this section was to be able to read ADC values directly from a controlled DC voltage source, and display our values on the Hexadecimal display. We will start off by giving a brief introduction to the LTC2308 converter, followed by a discussion on the timing requirements involved, including as the final implementation in order to read the respective values. While we did face a lot of initial difficulties in order to accurately depict the values of the ADC from our board due to ambiguities in the datasheet while trying to recreate the timing diagram, we were eventually able to run this module on our board without any issues.

3.1.1.1 LTC2308 ADC Converter

The embedded ADC converter—LTC2308 on the DE1 SoC board is a low noise, 500Ksps, 8-channel, 12-bit ADC module with an SPI compatible serial interface. Its internal conversion clock enables the external serial output data clock (SCK) up to operate at the frequency up to 40MHz. It operates from a single 5V supply and draws just 3.5mA at a sample rate of 500ksps. It has internal 2.5V reference with 8-channel multiplexer to select its input channel. We are able to interact with this through our 2x5 header, as shown in 3. We have also included the block diagram

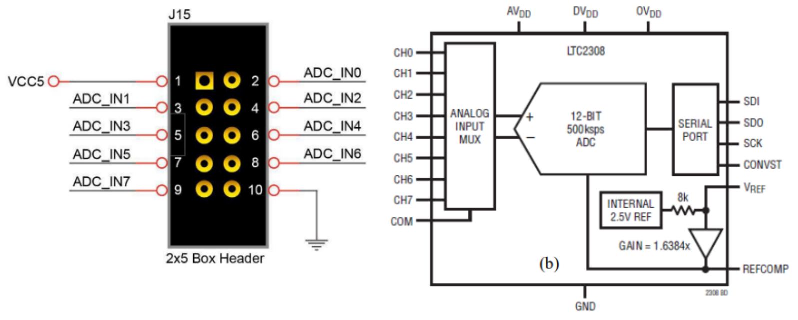


Figure 3: 2x5 Pin header on the De1 SoC, as well as the LTC2308 block diagram

of the LTC2308 in the figure. Note that ADC-IN0-7 refers to the 8 channels 0-7, which are included in our ADC logic. We read the values of the pins of channel 0 for our case. In LTC2308, under the control of configuration signals and internal clock, 12-bit ADC data can be measured within a sample rate 500Ksps. We have also attached an image delineating the connections between the FPGA, header, and the converter in 4. We can decide the appropriate pins to assign, as well as their interactions from this diagram. We will now explain the timing diagrams for each of these pins in the next section.

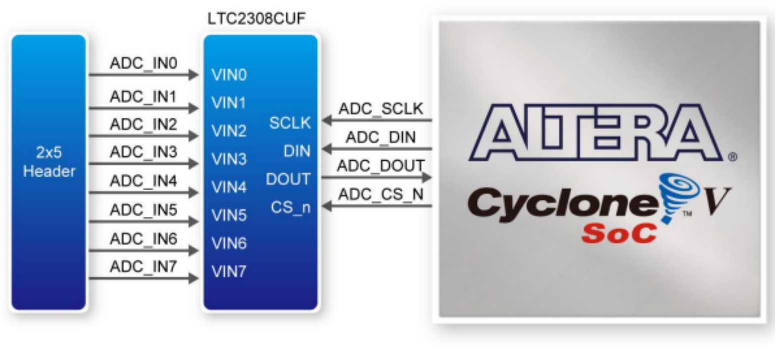


Figure 3-31 Connections between the FPGA, 2x5 header, and the A/D converter

Figure 4: Connections between the FPGA, 2x5 header, and the A/D converter

3.1.1.2 LTC2308 Timing diagram and implementation

We have written our code according to the timing diagram5, as shown in the documentation of the LTC2308. While the LTC2308 has many configurations for

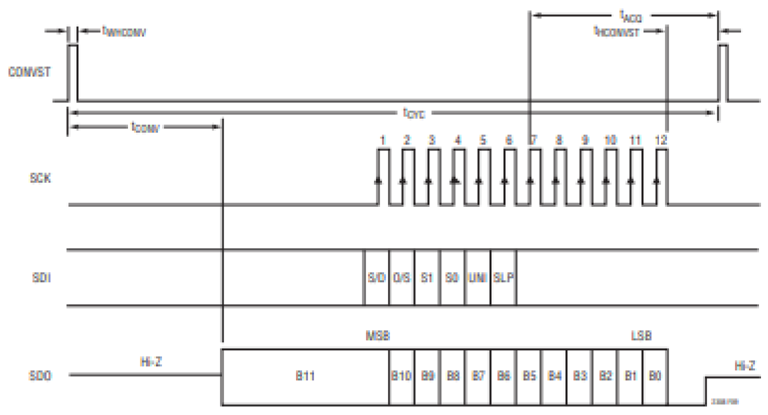


Figure 5: Timing diagram of LTC2308 with short CONVST pulse

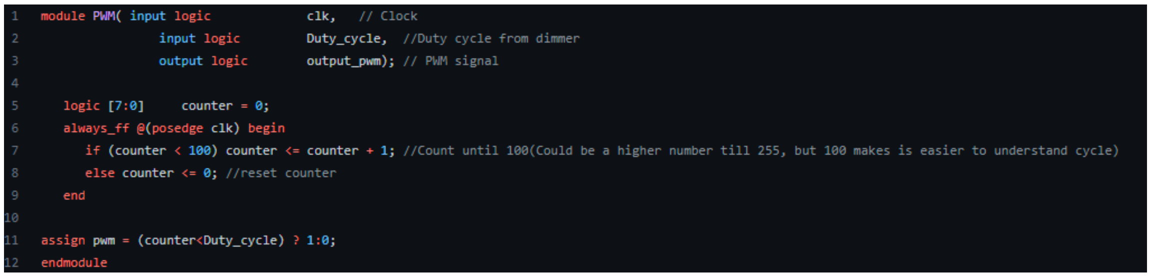
its timing diagram, we have opted for one that uses a short CONVST pulse for a higher sampling rate. Rather than trying to create a separate SCK signal for our ADC, we will be sampling it directly with the input 50 MHz clock signal. Note here that a rising edge at CONVST begins the conversion, and the SCK synchronizes the serial data transfer. The SDI serial bit stream configures the ADC to the appropriate settings for our use case such as the pin that needs to be read and the mode. This is latched on the falling edge of the SCK. On the other hand, the SDO pin outputs the data from the previous conversion, and is shifted out serially on the falling edge of each SCK pulse. Note that we use AJ4, AK4, AK3, and AK2 for the CONVST, SDI, SDO and SCK signal respectively, as described in the De1 SoC schematics. We have selected our timing values such that we can sample our reading at around a 100kHz as this would suffice for our application and conforms to the reuirements of the ADC. We then proceed to code our logic using this same timing diagram and requirements, and display our final 12 bit data on the hexadecimal display. As shown in our presentation, this is able to show the accurate input DC voltage with a precision of 1mV on the board, from 0V to around 3.85V. This value can then be fed to the motor controller to determine the appropriate duty cycle for our PWM signal. We have included comments in our code for reference regarding the logic used here. While we had intially opted to try and generate the SCK signal separately using the clock signal, we found this method inefficient, while also facing a great deal of debugging issues. We instead opt to make use of the same input clock signal for our SCK, where we only sample the clock signal for a specific number of cycles based on our timing requirements, and have written the rest of our code, including the configuration bits of the SDI and the output bits from the SDO using this logic. With this, we have successfully completed our first checkpoint.

3.1.2 Encoder

In this section, we will focus on the encoder used in our system, as this is one of the hardware requirements for checkpoint 2. The encoder embedded in the gearmotor sends 7 pulses per turn. Since they are pulses, there is no need to carefully sample the signal like it should for the current, for instance. Instead, a GPIO pin of the FPGA is configured, as the voltage of these pulses can be rated at 3.3V. The hardware will then count these pulses at a frequency of 100 kHz, but only send data at a 100 Hz rate. Indeed, this module could have been implemented in two ways: either waiting for the pulse count to reach seven or setting a frequency at which sending the data. The first method was implemented but did not give satisfying results. First, the controller needs an update of the speed regularly for computations to be accurate. Second, if the speed is null or very low, then it can take too much time to report the speed of the motor. The method that has been implemented, if close to the first one, differs by its regularity: every 10 ms, the pulse count is sent to the software and reset. The sampling frequency, although less than 100 kHz, is not a problem since no signal is being sampled: only pulses are counted. This rate of 100 Hz ensures that for a minimum speed of 1000 RPM, at least one pulse will be counted.

3.1.3 PWM

Here, we proceed to work on the generation of the PWM signal, as this will be required in our third and final checkpoint. To start off our logic for this section, we try to generate a PWM using the GPIO pins on the De1 SoC board from our preset Duty Cycle values, and have displayed these values on an oscilloscope to ensure that it is able to generate the signals appropriately. For this logic, we make use of a clock counter in order to keep track of our input clock signal cyclesand have generated the PWM signal approriately from this. An early attempt at our generation can be shown in 6. Once we were able to generate the PWM in software, we polished



```
module PWM( input logic            clk,   // Clock
            input logic            Duty_cycle,  //Duty cycle from dimmer
            output logic           output_pwm); // PWM signal

  logic [7:0]     counter = 0;
  always_ff @(posedge clk) begin
     if (counter < 100) counter <= counter + 1; //Count until 100(Could be a higher number till 255, but 100 makes is easier to understand cycle)
     else counter <= 0; //reset counter
  end

assign pwm = (counter<Duty_cycle) ? 1:0;
endmodule
```

Figure 6: Early logic used for PWM signal generation. Note here that Duty cycle could be written as any value out of 100(Example 45 corresponding to 45 percent. We polished this code up during our final implementation, but have included our early logic for reference

up our code further and had our GPIO pins output the signal at around 3V to our oscilloscope. Once we confirmed that our board was generating a PWM signal, we then proceeded to feed this signal into our L298n H-bridge converter to ensure that our motor speed was changing as a result of the PWM signal generated. We will now integrate this module with our controller logic, where the controller would determine the appropriate duty cycle value based on the input voltage, which should complete our final checkpoint of creating a hardware-software interface for our project.

3.2 Software Components

For this section, we will start off by fleshing out the details of our *cascaded control* system, which is a common control framework for motor drive systems in industrial application. It consists of nesting closed-loop controls of different actions. For a speed controller, a current controller is first implemented and the speed controller is added as an additional layer. Figure 7 shows how it is done in practice. A current controller in addition with a speed controller have been studied and implemented in software to control our DC motor.

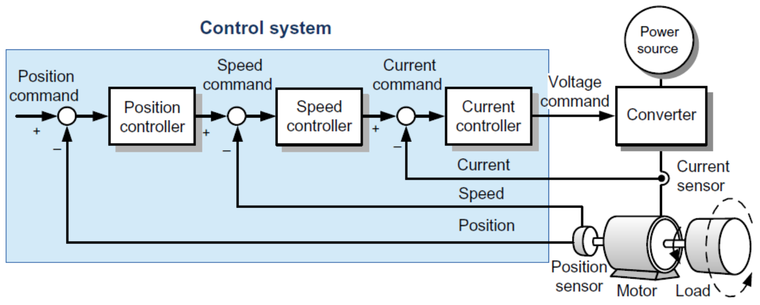


Figure 7: Cascade control diagram for a motor.

3.2.1 Design of the controllers

Before designing the controller, the parameters of the motor needs to be derived. The motor used is the gearmotor am-2971 by AndyMark. It embeds a PMDC motor with an encoder and a gearbox. The specifications given are not as relevant as they should for our application: it only refers to the stall current and torque (when the shaft is locked) and the maximum no load speed and current. Indeed, to build the controllers, the first step is to model the plant, here the DC motor. Figure 8 shows the components of a DC motor and figure 9, the conventional model, where:

- R and L are respectively the resistance and inductance of the armature windings (electric characteristics);
- k the machine constant;
- J the inertia of the shaft;
- B the friction coefficient that exercises a torque proportional to the speed.

These parameters cannot be derived from the given specifications. A extensive set of experimental tests need to be carried out in order to determine these parameters. It was not the object of this project, so approximate values have been chosen based on actual values found in books (see table 3.2.1).

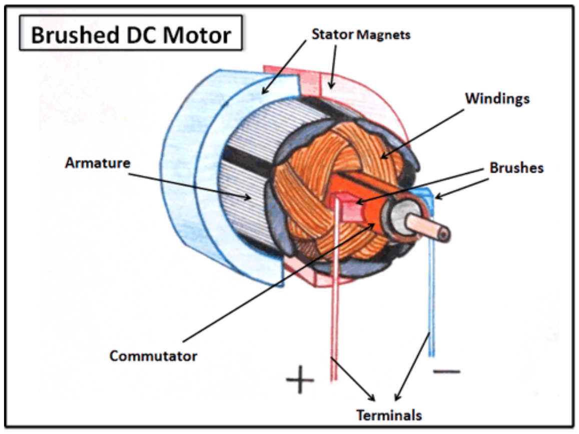


Figure 8: Sketch of a PMDC motor.

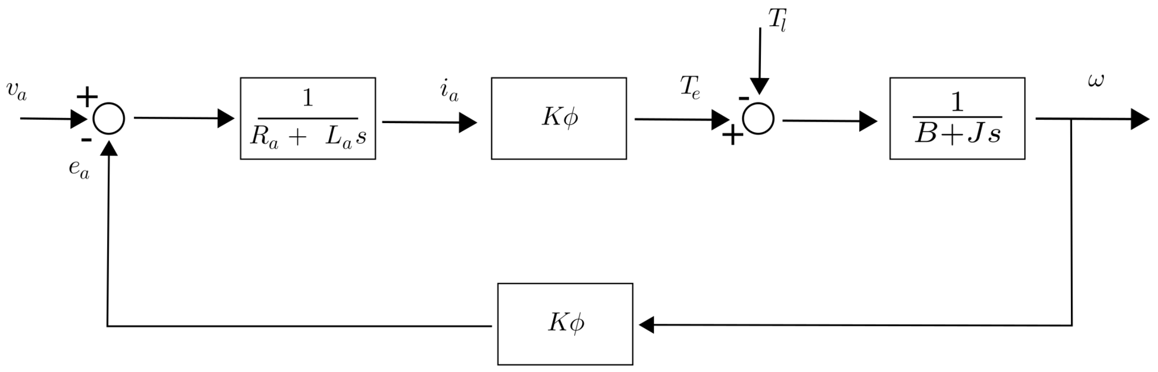


Figure 9: Diagram of the model of a DC motor.

Parameter	Value
R	0.2 Ω
L	0.002 H
J	0.02 kg.m^2
B	1e-5 N.m.rad/s

The current controller purpose is to limit the current flowing in the motor. Indeed, as detailed in the specifications, the current can reach 15 A when the rotor is locked, an intensity that can damage the windings. A PI controller forms the basis of the current controller, to which is added an anti-windup integrator to compensate for the overtime error of the first integrator, a feedforward to make up for the back e.m.f. of the DC motor, and a limiter to limit the output to the rated values (figure 10).

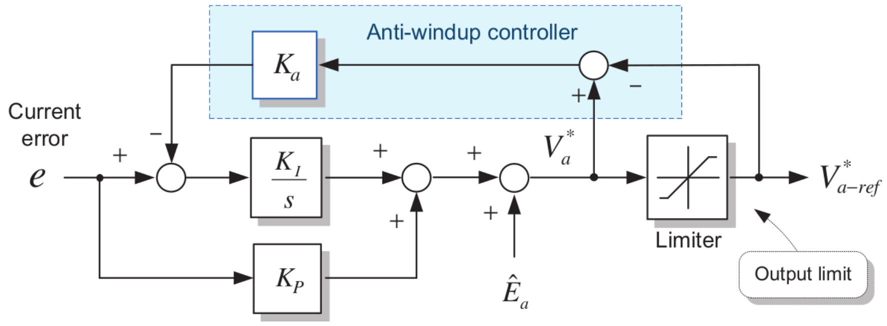


Figure 10: Diagram of the closed-loop current controller.

The same design is used for the speed controller, except that there is no feedforward.

For the controller to be adapted to the plant, the parameters that can be chosen are the different gains:

- $K_{p,i}$, $K_{i,i}$ and $K_{a,i}$ for the current controller;
- $K_{p,s}$ and $K_{i,s}$ and $K_{a,s}$ for the speed controller;

The gain of the anti-windup integrator is chosen to be one over the proportional gain: $K_a = \frac{1}{K_p}$. The proportional and integrator gains are first selected to respect criteria of stability, and then adapted to meet performance criteria. Since the switching frequency of the H-bridge L298N has been chosen to be 25 kHz (common switching frequency for this chip), the crossover frequency needs to be at least 10 times lower for the current controller: $\omega_{cc,i} = \frac{2\pi f_s}{10}$. Better performances have been encountered for a cross-over frequency of 5000 rad/s. Then, the gains for the current controller are defined by:

$$K_{p,i} = L\omega_{cc,i}$$
$$K_{i,i} = K_{p,i}\frac{R}{L}$$

The speed controller should have a lower cross-over frequency compared to the current controller: $\omega_{cc,s} = \frac{\omega_{cc,i}}{5}$. The expression of the gains are:

$$K_{p,s} = J\frac{\omega_{cc,s}}{k}$$
$$K_{i,s} = K_{p,s}\frac{\omega cc, s}{5}$$

With these parameters, the frequency response as well as the step response are shown figure 11.

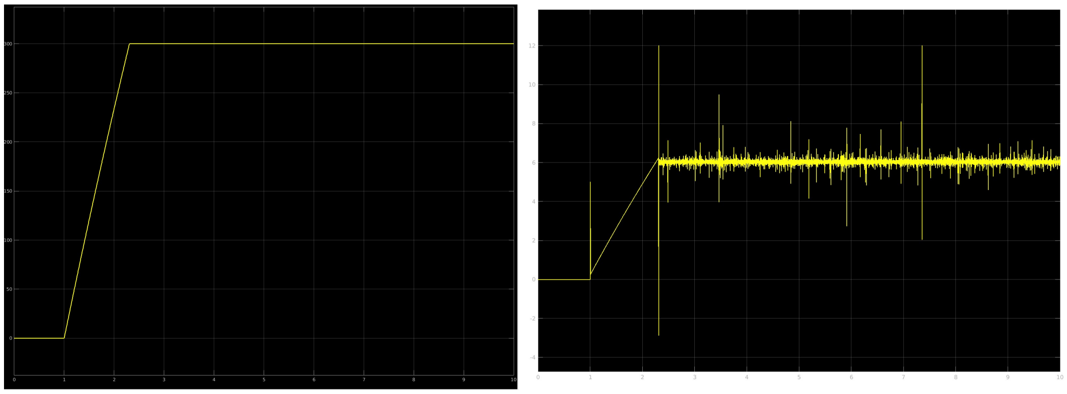


Figure 11: Response of the controller to a step of 300 rad/s at time t = 1s: Output speed of the motor (left) and output voltage command (right).

3.2.2 Discretization of the controller

The continuous-frequency controller built gives satisfying results. However, in a computer, it is impossible to use continuous functions: the functions are discretized. How to discretize the continuous-frequency controller without losing its fundamental properties? The zero-hold block method has been used to do so. It consists of sampling at a certain frequency the step response of the controller, determine its $\mathcal{Z}$-transformation and divide it by the $\mathcal{Z}$-transformation of a step to retrieve the $\mathcal{Z}$-transformation of the equivalent discretized controller:

$$\mathcal{Z}h_s = \frac{\mathcal{Z}step \star h_{fs}}{\mathcal{Z}step}$$

where h is the response of the controller to an impulse.

The initial controller is then discretized using the equivalent $\mathcal{Z}$-transformations and designed on Simulink. To check that the discretization has been properly realized, the same simulation have been carried out as the continuous-frequency controller (see figure 12).

Then, the remaining work is to derive the recursive sequence related to the $\mathcal{Z}$-transformation of the controller. Extracting the data from Simulink, C simulations are carried out to ensure that the outputs are similar and check the validity of the code.

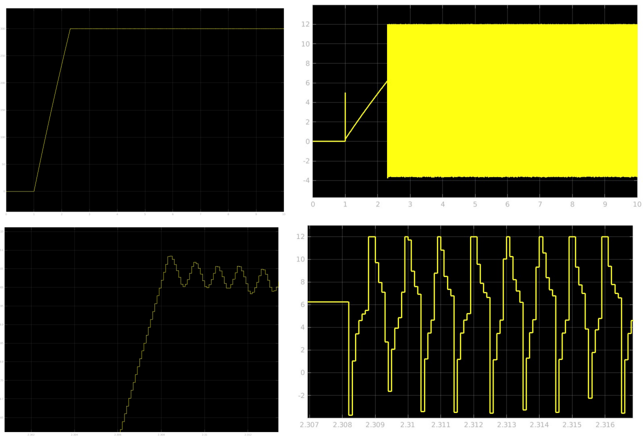


Figure 12: Response of the controller to a step of 300 rad/s at time t = 1s: Output speed of the motor (left) and output voltage command (right).

Nyquist's law explains that the sampling frequency of a signal should be at least twice the maximum frequency present in the signal. Since the switching frequency is set to 25 kHz, the fundamental of the signal will be 25 kHz. Applying a filter, this can be the maximum frequency present in the sampled signal. Consequently, the sampling frequency is set to 100 kHz. As a result, the time-step in the recursive sequence of the controller corresponds to the sampling period, that is 10 μs.

3.2.3 Implementation

The discretized controller is ready to be implemented in C. An object-oriented programming has been done regarding the controllers. It consists in a header *pi_controller.h* that defines the PI controller object and declares the function to use them, detailed in the corresponding source file:

- *pi_create* to initializes a controller;
- *pi_compute* to compute the output of a controller using the recursive relation and specially designed for proportional-integrator and proportional controller;
- *current_controller* and *speed_controller* to especially compute the output of the current and speed controller respectively;

Assuming the feedback from the hardware works as expected, the main algorithm to be run in a loop on the FPGA begins with an initialization of the drivers and controllers, pointers to send and receive data, and the creation and launch of several threads. These latter allow to update the values read from the ADC for the current feedback and the potentiometer that drives the motor command by the user; read and process the signals of the encoder to transform them in a speed feedback; send the duty cycle command derived from the voltage output of the controller. Finally, the program enters the infinite loop in which the computations of the controller outputs are carried out.

3.3 Hardware-software interface

Hardware and softare have been built independently, but need to be combined to work in symbiosis. To bridge the gap between both, a device called *pwm* has been created using Platform Designer. It embeds the module that is needed to read the pulse count and ensure the pwm function (hence its name). A driver has been written to read the encoder data from the peripheral, as well as send the commands to control the duty cycle and the rotation direction. The driver is also ready for the ADC values to be read, if the pwm peripheral would be used for it too. The duty cycle command is a number between 0 and 2000, because the a period of the PWM corresponds to 2000 counts of clock pulses (50 MHz clock). Thus, it is coded with 11 bits, which leads the chip to have a 16-bit pin to write data. The readdata pin is 16-bit too, even though the maximum value of the encoder count is around 7, as it is reset every 10 ms and the motor top speed is 5700 RPM.

We have attached a block diagram in 13 briefly showing our hardware/software interface.

4 Conclusion

In this report, we have presented a project that has aimed to design and implement an industrial DC motor controller by taking in input commands that can be set by the user during runtime. Through our project, the user is able to feed input voltage values to the FPGA, have the values input into the cascade control framework(that has been used to design the current and speed controller, and has also been simulated through Simulink. The C controller has been tested to ensure the performances), and finally output an appropriate PWM signal in order to control the speed of the motor. A user interface has been created to control in part the DC motor. The remaining work, for the project to be perfectly work, would be to build the hardware-software interface to process the data from the ADC and be able to control the motor with the potentiometer of a DC source and retrieve the current feedback for the controller, knowing that the driver for the ADC is ready to use. While not perfect, we believe

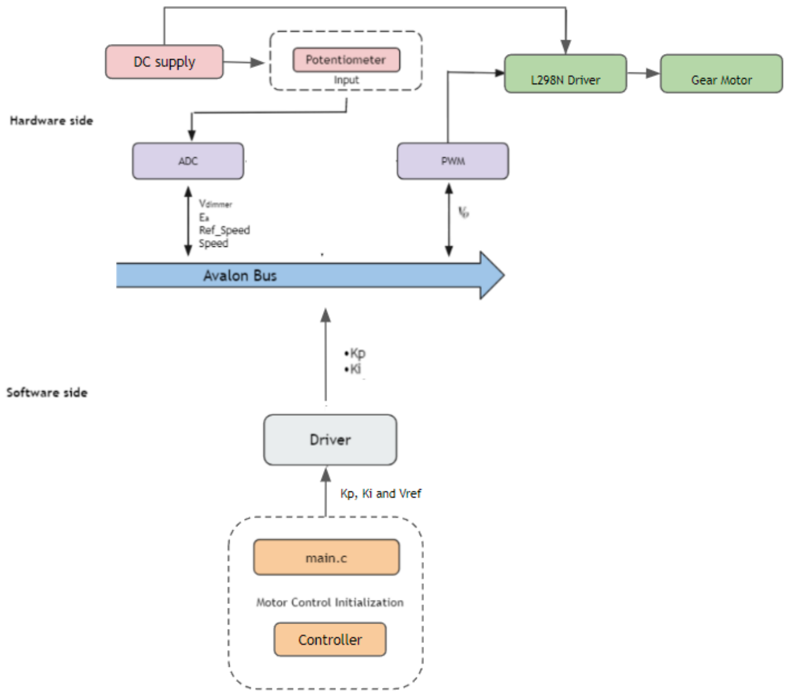


Figure 13: Hardware/Software Interface

that this FPGA based motor controller project has a lot of potential as it can be generalized to many different motor ratings. With a little bit of polishing, this work can be extended to a plethora of applications including robotic arm control, electric vehicles, and so on.

5 Project attribution and Insights

For this project, the analysis of the system and the control laws, as well as the use of the encoder, the development of the hardware-software interface and the merge to create the main code to be run have been implemented by Alexandre, while the hardware code for the PWM and the ADC readings from the LTC2308 have been developed by Ayman.

While working on this project, we have learned a great deal of things. These include the precision that we need to work with while trying to read values and recreate timing diagrams for something as simple as reading values from an ADC, all the way to the difficulties involved in actually implementing simulations from controllers to actual practice on the FPGA. If there's just one advice that we could give to students working on future projects, it would have to be to prioritize the datasheet timing requirements and board schematics, and understand every detail pertaining to your respective projects from it.

6 List of files

1. ADC.sv
2. hex7seg.sv
3. pwm.sv
4. motor controller.m
5. model motor controller.slx
6. discretized motor controller.slx
7. fpga main.c
8. pi controller.c
9. pi controller.h
10. pwm.c
11. pwm.h