EFES Project
Room Temperature Controller

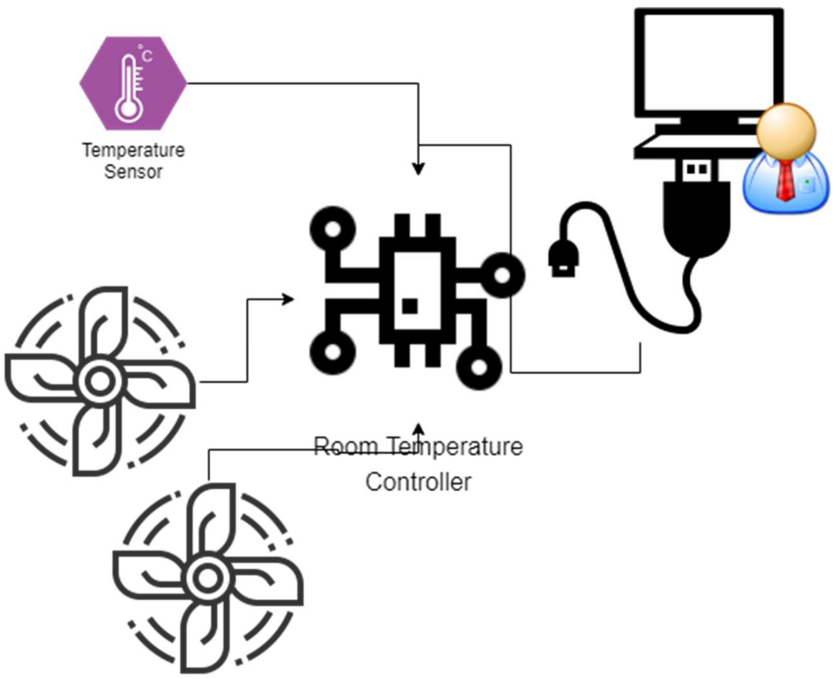
The project I am going to present is a Room Temperature Controller. It's a complex Embedded System composed of several parts that works together in order to maintain under control the temperature of a room (like a server room, university classroom, etc..).

The system is capable of manage autonomously the speed of two fan independently, according to three different temperature thresholds that can be configured via an operator through a user-friendly interface on a personal computer connected via USB to the system.

The main features are:

- Independent speeds for two fans with integrated DC motor controller and separate power supply to reduce noises.
- Three different temperature thresholds based on an average of all the temperature samples. For each threshold is possible to configure independently the speed.
- Ability to set on how many samples compute the average.
- Ability to set the sampling period of the temperature down to 100 ms.
- On-Board special purpose memory where the last 2048 samples and averages are saved.
- Analog temperature sensor and a 7-bit ADC for higher precision.
- Remote administration via USB interface and User-Friendly GUI.

A Top-Down approach: the high level overview of the system



As shown in the diagram on the left, there are two fan (actually two DC motors) controller that can be driven at different speeds.

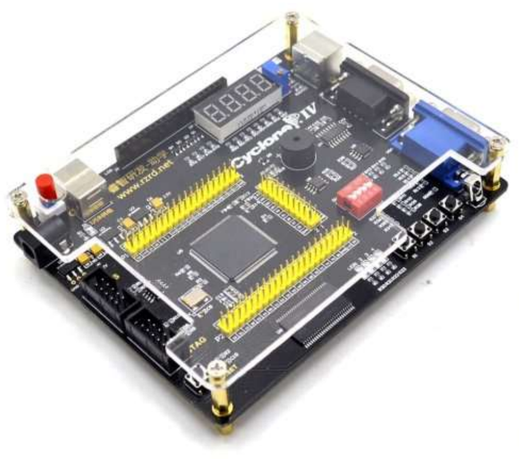
An operator can administrate the system through a USB port. A USB to UART converter is needed, thus the parameters used by the system to communicate through this interface are 19200 8N1.

The high level protocol used to communicate is a proprietary one, developed exclusively for this system, and it's called "G-UART or GabriUART", and its details will be discussed in the next pages.

The program delivered with the system already implements this protocol so the operator can simply put the serial parameters and administrate it easily. Alternatively, it's possible to write an own program that interfaces with it using the proprietary protocol.

A deep overview of the system

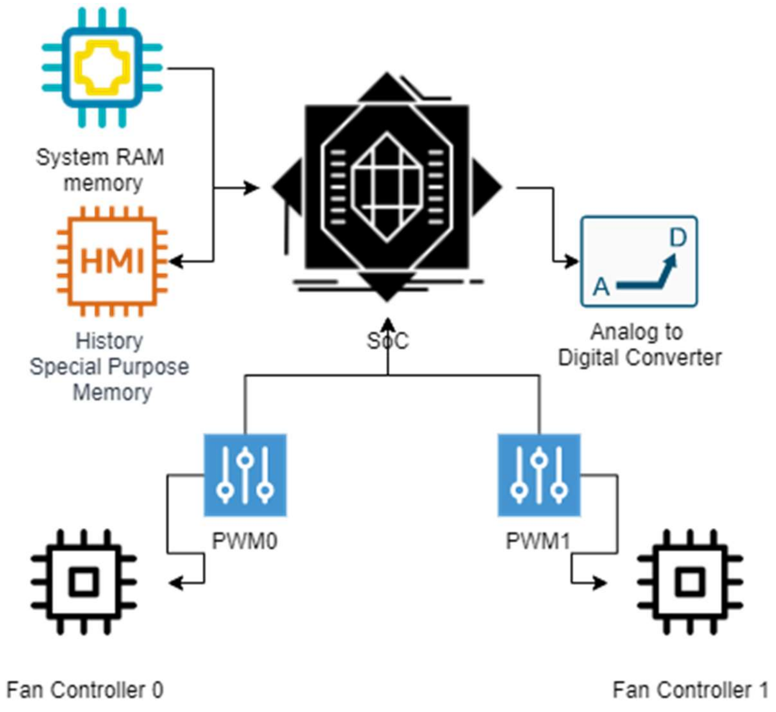
Now it's time to go deeper in analysing the system and how it works internally. The central controller is the core of the system, and it's based on a System-On-Chip with different peripherals inside it and proprietary peripherals connected around it.



Technically speaking, the entire System is implemented on a development board with a FPGA on it and external circuitry on breadboard. The FPGA is an Intel Cyclone IV EP4CE6. On the development board we found a SDRAM chip used by the SoC loaded inside the FPGA as Data and Code memory. On board there is a RS232 port driven by a MAX3232E IC, but I decided to not use it but a normal TTL UART connection and a TTL to USB converter.

There is no dedicated GPIO header on the board, and as will be discussed later looking directly at the schematic of the board, special purpose pins of the FPGA have been used connected to others peripherals.

The neighbours of the System On Chip



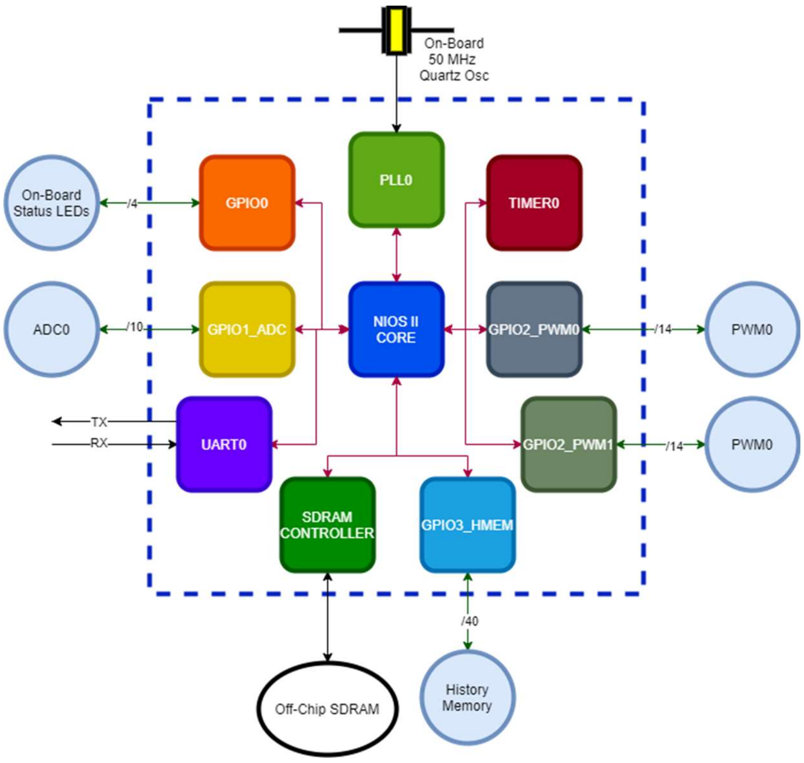
The FPGA chip mainly offers 6272 Logic Elements, 2 PLL, 270 Kbits of embedded memory, and 179 I/O.

Those features are enough to implement what we need: A System-On-Chip, a special RAM memory for storing history, two PWM controllers and the logic of an Analog to Digital converter. We have enough pins to connect to the off-chip SDRAM and to external connections towards the breadboard with external electronic components.

Both ADC and PWM are custom-made peripherals. I decided to implement it in hardware and not in software because they implement operations that are time-consuming and the core needs to do other operations while the ADC is converting or the PWM controller is generating its output signal at a high frequency.

It's both an advantage and disadvantage. It's a disadvantage because I had to spend times in implementing them in hardware, so I had to design, simulate and implement them using VHDL (this means a very low level work), but by other hand it has a lot of advantage because doing the same operation in software would be very complicate and I think pretty impossible to multiplex the time through these operations and all the other operations that the core should do like administrate all the system.

Inside the System On Chip



The core of the entire system is the SOC showed here. At its core we found a NIOS II core.

NIOS II is a 32-bit embedded-processor architecture designed specifically for the Altera family of field-programmable gate array (FPGA) integrated circuits.

Its architecture is a RISC soft-core architecture which is implemented entirely in the programmable logic and memory blocks of Altera FPGAs.

The soft-core nature of the Nios II processor lets me design specific and custom NIOS II core, tailored for my specific application requirements.

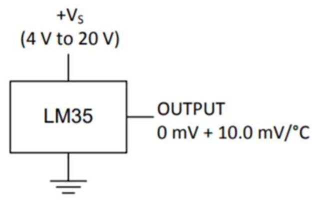
Through a PLL (PLL0), is possible to generate different clock signals starting from the external oscillator installed on the development board in use. It's a 50 MHz quartz oscillator and the PLL outputs 4 different clock signals for different uses. One of these is the frequency at which all the internal peripherals in the SoC and the core itself work and it's a 50 MHz clock. Other three PLL0's outputs will be discussed later in this document.

I want to underline the high number of GPIO peripherals used. This is because GPIO is useful to control peripherals outside the SoC like the ADC or PWM controllers. Especially for this last peripheral, unfortunately through all the list of possible official NIOS peripherals, there is no PWM controller! So I had to implement it on my own and this is why we have two GPIO dedicated to this.

The TIMER0 peripheral is used to have a bit of timing inside the core application. It's used mainly for managing the correct sampling behaviour according to the timing configured by the operator.

The UART0 and SDRAM Controller will be discussed further in their respective paragraphs.

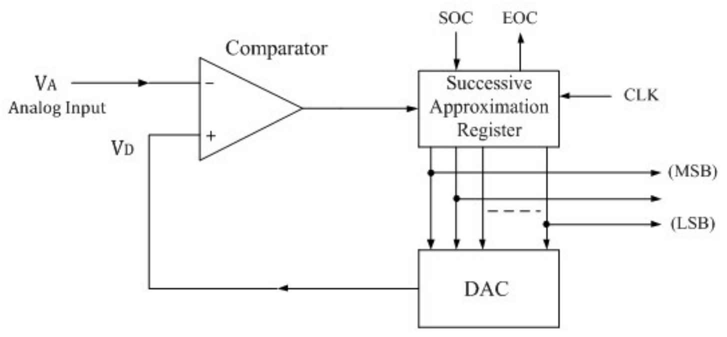
The home-made Analog to Digital converter is a SAR 7-bit ADC. The SAR logic is implemented, as said, in VHDL. Its job is to convert the analog output of the temperature sensor.



The temperature sensor is a LM35. It's powered at 5V (provided by the FPGA) and its output is an analog voltage that starts from 0 mV and increase by 10 mV at each °C.

During the design stage of the ADC, only a certain range of temperature was taken under consideration. In particular, the system is designed to work correctly in a room temperature that spaces from 0°C to 40°C. Electrically spoken, this means that the input dynamic range of the signal to convert is from 0 mV to 400 mV.

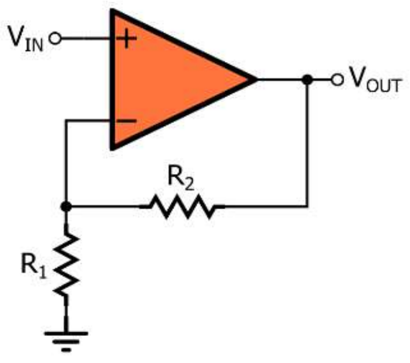
The input dynamic range of the implemented converter is from 0 V to 3.3 V, this means that we need to amplify the output of the sensor first, then convert it. The amplification is needed in order to have a more accuracy on the conversion, otherwise only a minimum subset of the entire dynamic range of the converter is used, and this means a less accuracy. We know that our signal will always be from 0V to 400 mV, this means that all values from 401 mV toward 3.3V will never be reached!



I decided to use a SAR converter because of its velocity and relative ease of its implementation. We don't need a high speed conversion so a flash converter will be useless (and impossible to implement in my design with 7 bits: this means $n_{comp} = 2^7 - 1 = 127$ comparator!).

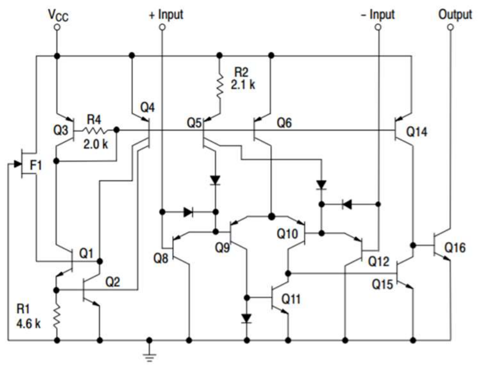
The amplifier stage of the ADC

The amplifier used is a LM-358P Operational Amplifier. Its rails are connected to 5V (provided by the FPGA and to 0V). Why 5V and not 3.3V if I don't need a so high voltage? Mainly because it's not a rail-to-rail operation amplifier (rail-to-rail means that the output can range from within a few millivolts of the positive supply voltage to within a few millivolts of the negative supply voltage). In my case this means that would be impossible to amplify the input signal from 400 mV to 3.3V having a Vcc = 3.3V!



This Operational Amplifier is used in a "non-inverting" configuration, obviously with a feedback connection. We know that the maximum input voltage is 400 mV that should correspond to a 3.3 V output, so the gain we want is $A_v = \frac{3.3\ V}{400\ mV} = 8.25$. The configuration of the Op-Amp in use imposes a gain that is $A_v = 1 + \frac{R2}{R1}$, and comparing them we have $1 + \frac{R2}{R1} = 8.25 \rightarrow \frac{R2}{R1} = 7.25$. It's not so easy choosing two resistors that respect this ratio among the fixed value that we find on the market. I used $R_1 = 47\ k\Omega$ and $R_2 = R_a + R_b = 330\ k\Omega + 10k\Omega$ for a ratio of about 7.23. The used resistors are made with a tolerance of 5%.

The comparator of the SAR converter

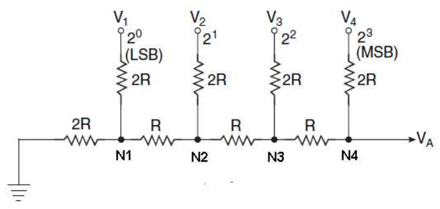


The comparator used is a LM393. This time it's a rail-to-rail comparator so we have to connect its power rail to 3.3V and not to 5V because, looking at the datasheet, the FPGA pins are rated to works at a maximum voltage of 3.6V! We need to use the output of the comparator as an input of our SAR logic implemented inside our FPGA so we have to pay attention on this particular aspect.

Another important aspect to consider is that the comparator I decided to use is an open drain comparator. In fact, looking at its internal structure we discover that the output is driven only towards ground and the output toward Vcc is left floating. So, an external pull-up resistor is needed, and I choose a value of $R_{pu} = 4.7k\Omega$ because we need only a small amount of current to drive our FPGA pin so there is no need to use a lower value.

The DAC inside the ADC

For a 7-bit SAR ADC we need a 7-bit DAC. How to realize it? I thought about it for a few days and at the end I came up with a circuit that is a R-2R ladder network because it's easy to implement and very efficient for my application.



In the figure on the left we have only 4 bits but it's easy to extend to 7 bits (the entire electronic ADC schematic will be found in the next pages).

In my circuit, I used a value of $R = 10\ k\Omega$. The maximum output voltage will be (with $V = 0b111\_1111$) $V_A = V_{ref} \cdot \frac{V}{2^7} = 3.274\ V$ where $V_{ref} = 3.3\ V$ is the voltage applied by our digital output pins when logic 1. The value of a LSB, that corresponds to $V = 0b000\_0001$ is $V_A = V_{ref} \cdot \frac{1}{2^7} = 25.8\ mV$. This is the precision of our DAC and it seems to low compared to the precision of our temperature sensor (10 mV/1°C) but this is the LSB that corresponds to the DAC dynamic range. The signal range is lower (it's amplified), so it means that a LSB of the DAC is equal to ${}^{LSB}/_{A_v} = 3.1\ mV$ and it's even higher than the precision of the temperature sensor!

N.B. Usually, together with this kind of DAC we found an operational amplifier. In this case it's not needed because the output V_A is not used on a heavy load but it's used as an input of the comparator so the current draw is very negligible that doesn't justify the use of an extra operational amplifier. Looking at the datasheet of the LM393, the maximum input bias current is of 400 nA when not at $T_a = 25\ °C$, while the typical value is 20 nA.

The SAR logic

```
component SAR is
   generic(
      nbits       : integer := 2;
      reg_siz     : integer := 2
   );

   port(
      clk         : in std_logic;
      rst         : in std_logic;

      --Start of conversion
      soc         : in std_logic;

      --End of conversion
      eoc         : out std_logic;

      --Comparator input
      comp_in     : in std_logic;

      --Data output: last conversion
      data        : out std_logic_vector((nbits-1) downto 0);

      --Data output: to DAC
      DAC_data    : out std_logic_vector((nbits-1) downto 0)

   );
end component;
```

Finally, the last piece of the converter: the SAR logic. Implemented in VHDL on the FPGA, it's an entity with different control signals and I/O.

The control signals are the Start of conversion (soc) useful to start a conversion, and the end of conversion (eoc) that signals that a new data is available on "data" bus. "comp_in" is the output of the comparator, while the difference between "data" and "DAC_data" is that the first one is a latched output and the value stays stable until "eoc" becomes one when it's updated with a new value. "DAC_data" change continuously and it's the value that feeds the input of the DAC showed before.

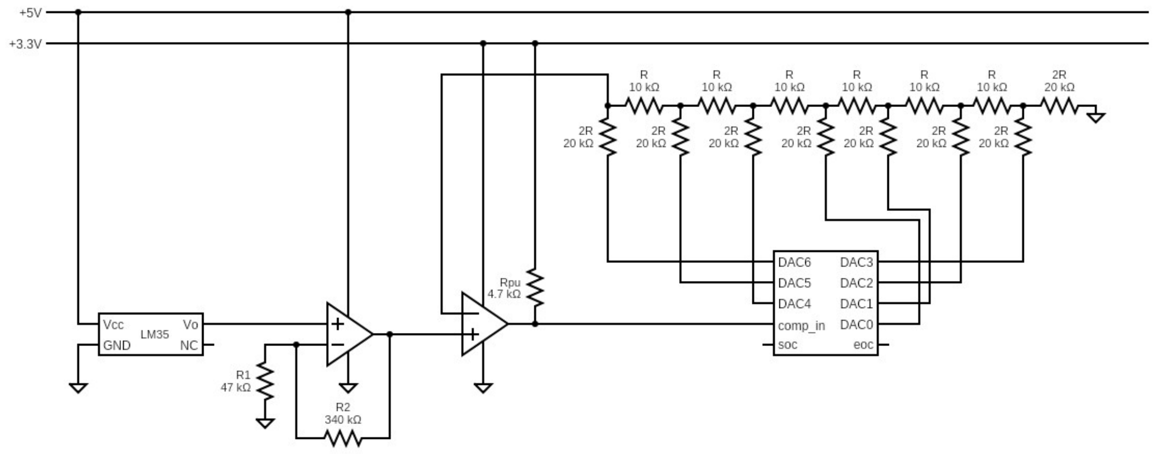
Internally, it's implemented as a High-Level State Machine (Moore Machine). There are mainly two states, one tries to guess putting a 1 at the current position while the second one puts at the current position the value that comes from the output of the comparator, then it goes forward repeating everything at the next position, until it reaches the lsb. This logic works at a lower frequency of the internal SoC one.

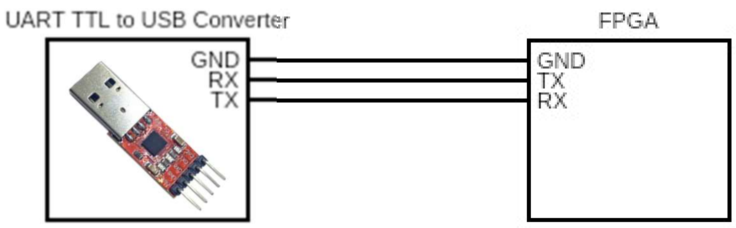
It's one of the four clock output generated by PLL0 and it's a clock signal of 5 KHz. Maybe the converter can work at higher frequency but I didn't want to exaggerate. For my application it's more the sufficient: it means a $T_{conv} = T_{clk} \cdot 16 = 3.2\ ms$.

The implementation of the DAC

Here there is the electric schematic of the ADC. It's implemented in the same way on the breadboard. The two power rails come from the FPGA and the SAR logic (the box on the bottom-right side) is, as said, inside our board.

The "soc" and "eoc" wires are not showed because they are directly connected to the dedicated GPIO lines towards our SoC, so they aren't external wires.

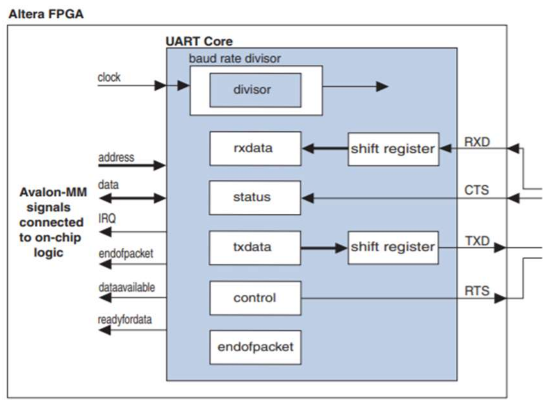




Thanks to the NIOS II's UART peripheral, is possible to give to my system the ability to communicate with an external PC using the UART protocol. UART means Universal Asynchronous Receiver-Transmitter.

The protocol is pretty simple: it's composed by two wires (RX and TX) and a third wire that is the ground. The clock is not transmitted and it's generated internal by each receiver.

On the system, a bi-directional communication is supported (a computer can send data to the controller and vice versa). In order to support those operations, a "home-made" drivers were written in order to simplify the utilization of the peripheral. More details can be found under the "software and firmware" chapter of this document.



The NIOS II's UART peripheral can be partially configured via software. During its generation, is possible to "hardwire" some configurations. In this case, I configured the number of bits (8 bits), number of stop bits (1 bit), the kind of control bit (like parity or odd bit, I choose to not use any control bit).

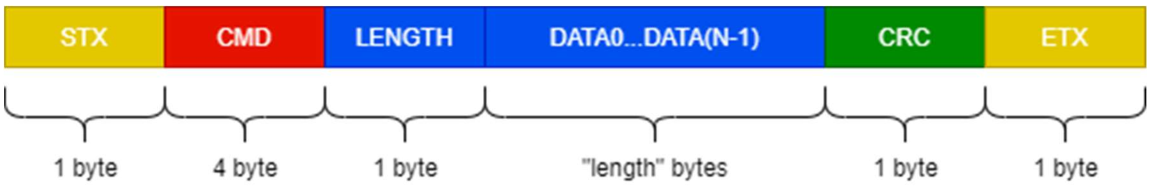
There is the choice to leave the baud rate configurable via software and I enabled it, in this way is possible to configure it via the drivers I wrote. This means that the baud rate is "softwired" at 19200.

To configure the baud rate at this value, there is to configure the "divisor" register inside the peripheral, and this is done via software. The formula used to compute the divisor given the operating frequency of the peripheral (50 MHz, the frequency used inside the SoC) is the following one:

$$divisor = \frac{clock\ frequency}{baud\ rate} - 1$$

The "G-UART" protocol

G-UART is a high level communication protocol based on UART written specifically for this application. Its job is to give a standardized way to communicate from the PC to the System and vice versa. The protocol is mainly based on character commands, and this means that it uses Control ASCII characters too. Frame types are DATA, ACK or NACK.

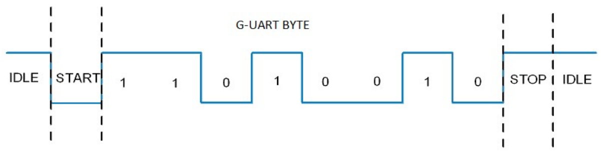


In details, we have:

- STX: Start of Text and it's equal to the ASCII code 0x02.
- CMD: it can be a command on 4 bytes or if it's a ACK or NACK packet, it's 1-byte wide and contains respectively 0x06 or 0x0f.
- LENGTH: 1 byte wide, means that a DATA packet size can space from 0 byte to 255 bytes.
- DATA0...DATA(n-1): it's optional if LENGTH=0, otherwise contains N data. If you want to send, for example, a 32-bit integer, you have to send it in "little-endian", means that data[0] is the less significant byte
- CRC: it's computed as the sum of the CRC of the CMD + CRCs of all DATA.
- ETX: End of Text and it's equal to the ASCII code 0x03

ACK and NACK frames are special: LENGTH, DATA and CRC fields are mandatory and they can contain any data. For example, it's possible to specify a length of 10, sending 0 data with a dummy CRC.

ACK or NACK are sent from the receiver when it receives a frame. It's one of the two based on the CRC computed on receiver side and compared to the one inside the sent frame. If they are equal a ACK is sent, otherwise the transmitter will receive a NACK.



Obviously, each byte has to be encapsulated in a UART frame. This means a START bit, then the byte content on 8 bits, then a STOP bit. This is according to the UART peripheral settings.

The available commands

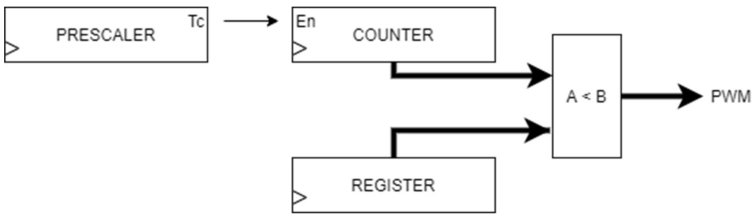
The controller's firmware implements a few commands (that it receives or send to a client). These commands are:

- CURR: it's a 1-byte data frame containing the current temperature read.
- CAVG: it's a 4-byte unsigned integer containing the sum of last samples. Is up to the client compute the average by knowing on how many samples the average is computed
- PING: sent at the beginning of a communication, it's a sort of handshake between the client and the controller
- PONG: is an answer sent from the controller after the PING command is successfully received
- CSET: a group of new configurations sent by the client. It's a 20-byte data frame containing all the new configurations of the controller
- CGET: a data frame sent by the client to request the current configuration. It can be 0-byte wide
- CSEN: a 20-byte data frame sent by the controller after the CGET frame to the client to instruct it about the current configurations
- HGET: a data frame sent by the client to request the current history. It can be 0-byte wide
- HSEN: After a HGET is received, the System will start to send data frame of 128 bytes each containing all the history saved in the memory. They are called "chunks" and actually there are 16 chunks sent for a total of 2048 bytes.
- LIVE: sent by the client, informs that the client is still connected. The client has to send it periodically to the system, otherwise it will stop sending information like CURR and CAVG. This is because the controller implements an anti-flood protection system on the UART channel.

As said, unfortunately NIOS II doesn’t have its own PWM peripheral, so I had to design it from scratch.

I need to have the ability to generate some PWM signal in my system because it’s the smartest way to control the speed of a DC motor. A PWM signal is a digital signal with the idea behind it to have the ability to generate a signal with different duty cycles.

Varying the duty cycle equals to directly varying the DC component of the signal itself but in our case it’s useful because having a high frequency signal (in the order of KHz), with a duty cycle of 20% for example we can tell to a motor “stay on for the 20% of the time and off for the 80% of the time during the period of the signal”. Doing this at high frequency means that the motor effectively changes its speed because of its natural inertia.



The peripheral is written in VHDL too and it consists of a top level entity that connects together 4 sub-entities has shown on the left.

The value of the register can be changed externally and it indicates the duty cycle. The counter counts up continuously when the Prescaler enables it. The PWM signal is HIGH when the counter has a lower value that the one stored in the register, LOW otherwise.

Both the Counter and Register works on 10 bits, this means that the duty cycle can be expressed as a value that spaces from 0 to 1023.

The Prescaler is on 32 bit and it’s there to possibly change the clock frequency of the Counter. However, its value is hardwired so is not possible to change it via software. It’s set to 0x1, this means that the counter will work at a frequency that is 0.5 times the frequency given as input to the PWM peripheral.

```vhdl
entity pwm is
    generic(
        nbits_pwm, nbits_divisor: integer
    );
    port(
        clk, rst: in std_logic;

        divisor: in std_logic_vector((nbits_divisor-1) downto 0);
        ld_divisor: in std_logic;
        done_divisor: out std_logic;

        duty: in std_logic_vector((nbits_pwm-1) downto 0);
        ld_duty: in std_logic;
        done_duty: out std_logic;

        pwm_out: out std_logic
    );
end pwm;
```

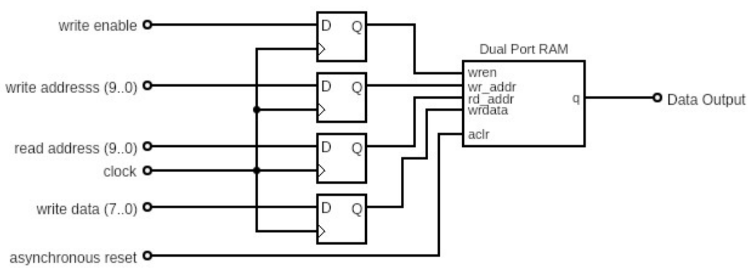
As shown here, we have mainly 4 important signals. Duty is the value that we want to load into the Register, ld_duty is the signal that communicates we want to update the value inside the register and done_duty signals that the update is completed.

Pwm_out is the signal we want and it’s a Pulse With Modulation signal that we will use as input of our DC Motor controller. The clock frequency of the peripheral is one of the four generated by the internal PLL of the SoC and it is at 2 MHz. This means that the counter works at 1 MHz, and because we need a full count from 0 to 1023 of the counter to complete a period of the pwm signal, means that the frequency of the PWM signal is $f_{pwm} = \frac{1\ MHz}{2^{10}} = 976.56\ Hz.$

The History Memory and the SDRAM Memory

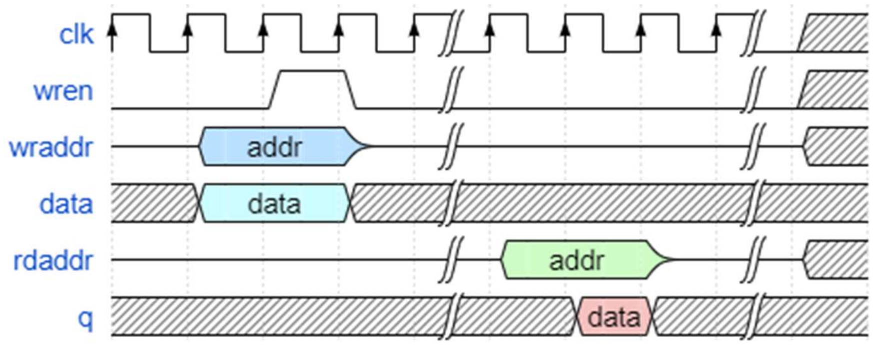
The History Memory is a special purpose memory meant to store few data about the samples. It is inside the FPGA and it is made using a portion of the available embedded memory. On the Cyclone IV, they are known as M9K because they are memories with words that are 9-bits wide. And the "9K" means that they have 9216 programmable bits.

It's strange to see a word with 9 bit, we are usually used to see 8 bit or multiple of bytes. The ninth bit is there meant to be used as a parity bit of the word so it can be useful in certain applications.



Our "History Memory" is actually composed of two M9K: it's a dual-port RAM with 2048 words and each word is 8-bit wide. Dual-port means that we have a "write port" and a "read port".

The memory is meant to store our samples, but each sample is composed of 7 bits, so what is the purpose of the eighth? It's used as a control bit, if "1" means that the current word is an average temperature on previously samples, while "0" means that is a pure temperature sample. It's easy to know on how many samples that average is composed because between an average and the next one we only have temperature samples that belong to the next average.



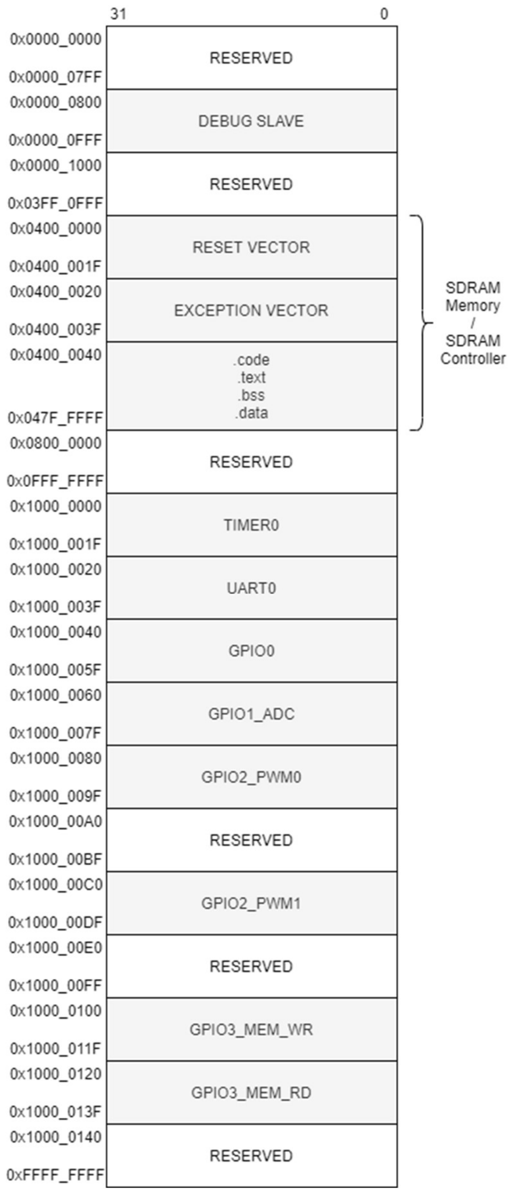
The clock frequency of the memory is the same as the PWM peripheral's one, so 2 MHz.

As see in the block diagram, all the inputs are latched so in order to have a correct behaviour during the write operation, I decided to follow this timing diagram where wraddr and data are first latched then the wren is set to HIGH for a clock cycle after 1 clock period.

A few words about the off-chip SDRAM. It's a 64 mbit synchronous DRAM, with 4 banks of 16 mbit each. And to be honest, I spend a day more or less to make it works with the NIOS II core. After a few hours, I came up with the idea that the problem maybe is the timing! And the first thing I tried (and fortunately it worked immediately) was to delay the clock signal of the SDRAM, in this way the SDRAM Controller can set its control data first, then after a few nano seconds arrives the rising edge of the SDRAM clock (that is different as said from the SDRAM Controller because of the delay) the latches everything. The delay is needed because of this latch, so I think the trouble was caused by the setup time!

So the SDRAM clock signal is the fourth and last one generated by the PLL, and it is at 50 MHz as the SDRAM Controller but with a phase shift of -65°.

The firmware of the System is very important, so it needs a dedicated chapter too. It runs thanks to our NIOS II core, that came up with a set of libraries and predefined drivers to manages SoC's peripherals like the UART, the Timer and the GPIOs (called PIO in NIOS' terminology) like in my case.



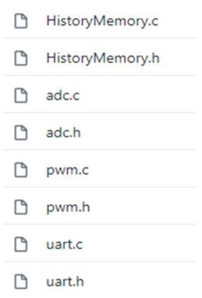
All these peripherals are accessed through direct addressing, and this means that the address of the peripherals and of the memories are all together in the same space address.

We are working on a 32-bit architecture, and this means that we can address up to 4.3 billion different memory locations, so it's better to give a bit of organization to all of this addresses.

This space address is configurable during the process of generation of the SoC, and I decided to split the memory into two halves: the first one is dedicated to the memories and debug peripherals while the second one is entirely dedicated to the peripherals.

Each peripherals contains, inside its space address, different registers useful to program the peripheral itself. The drivers I wrote make use of them. I decided to write my own driver for the UART and GPIOs peripherals. The ones related to the GPIO peripherals are specially dedicated to the external peripheral they are associated to, so for example I wrote a driver called "adc" that exposes some APIs to get data from the ADC and internally it uses the GPIO to control the ADC peripheral.

The drivers



HistoryMemory.c
HistoryMemory.h
adc.c
adc.h
pwm.c
pwm.h
uart.c
uart.h

In order to make my life easier in programming those drivers, I decided to make use of C's structs and C's bit field structs. I like a lot the bit field structs because they permit us to access single bits in an easier way without the need of writing manually all the bitwise mask.

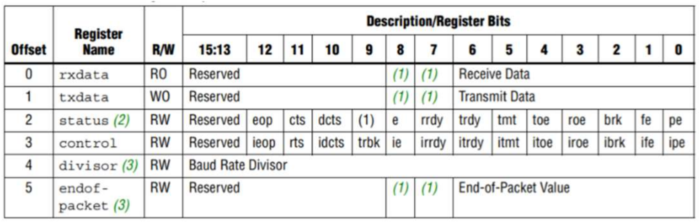
But if it's true that for complex peripherals like the UART and the ADC I used this feature of the C language, by other hand for the GPIO0 (used to manage the four leds on the board) I decided to not write any driver but to program its GPIO peripheral manually with classic bitwise operations.

The UART driver

The UART driver is the longest driver in term of code size. It exposes a data type that is *uart_t* initialized by `uart_t uart_begin(int *base_address, alt_u16 baudrate_divisor)`.

Then we have different functions useful to read and write to the UART line, like *uprintf, uputc, ugets, ugetc.* Last but not least, we have *uart_register_irq()*, this is very useful. In fact, in this firmware the RX side of the UART line, on the FPGA side, is managed by a mechanism based on interrupts. The NIOS II core has inside it an internal interrupt controller (so no external one is needed for my application).

Using interrupts is very useful, in this way the overall system is very reactive to external stimuli: it can do its work on sampling the temperature and managing everything and only when there is a data available it interrupts its work and executes the UART Interrupt Service Routine.



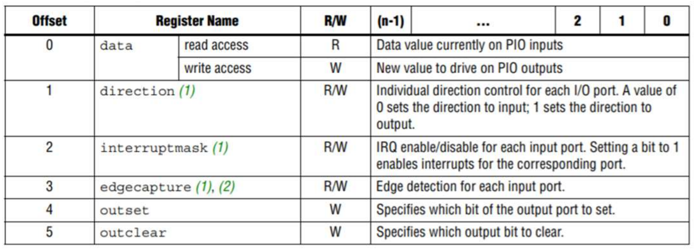
Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata	RO	Reserved					(1)	(1)	Receive Data						
1	txdata	WO	Reserved					(1)	(1)	Transmit Data						
2	status (2)	RW	Reserved	eop	cts	dcts	(1)	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	control	RW	Reserved	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	divisor (3)	RW	Baud Rate Divisor													
5	endof-packet (3)	RW	Reserved					(1)	(1)	End-of-Packet Value						

The *uart_t* is a type based on a struct as described before. It virtualizes all the bit fields of the peripheral's registers as showed in the table on the left.

Registers are 32-bit wide either if only 16 bits are shown in the table.

The ADC/PWM/History Memory drivers

These drivers are very similar to each other because they control peripheral that are external to the System on Chip and then drove with GPIO. The only difference is in the name of the exposed APIs and data types, obviously, and in the number of GPIO pins used and how each GPIO pin is connected to its relative peripheral Entity.



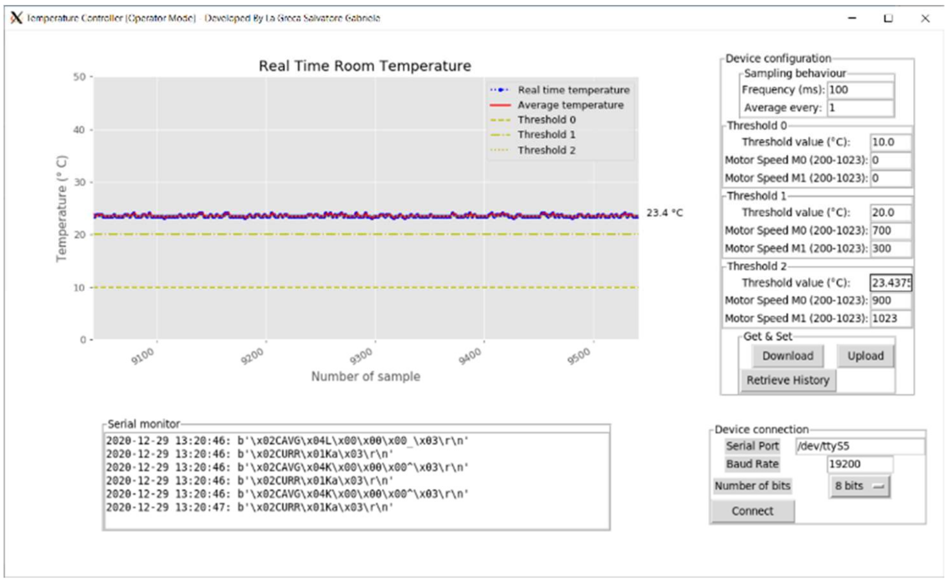
Offset	Register Name		R/W	(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	direction (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture (1), (2)		R/W	Edge detection for each input port.				
4	outset		W	Specifies which bit of the output port to set.				
5	outclear		W	Specifies which output bit to clear.				

For the ADC, we use only 10 pins, and they are mapped as follow: bit[0] is soc, bit[1] is eoc, bit[3..9] are latched data output.

For the two PWMs, we use 14 pins, where bit[2] is ld_duty, bit[3] is done_duty and bit[13..4] represents the data that goes into the duty cycle register of the peripheral.

Last but not least, for the History Memory peripheral we have two separate GPIOs, one for the read port and one for the write port, and they are symmetric for an easier utilization. The mapping is shown in the right table.

PIN	GPIO WR	GPIO RD
0	wren	Reserved
8..1	wrdata	rddata
19..9	wraddr	rdaddr



Device configuration
Sampling behaviour
Frequency (ms): 100
Average every: 1
Threshold 0
Threshold value (°C): 10.0
Motor Speed M0 (200-1023): 0
Motor Speed M1 (200-1023): 0
Threshold 1
Threshold value (°C): 20.0
Motor Speed M0 (200-1023): 700
Motor Speed M1 (200-1023): 300
Threshold 2
Threshold value (°C): 23.4375
Motor Speed M0 (200-1023): 900
Motor Speed M1 (200-1023): 1023
Get & Set

Device connection
Serial Port: /dev/ttyS5
Baud Rate: 19200
Number of bits: 8 bits

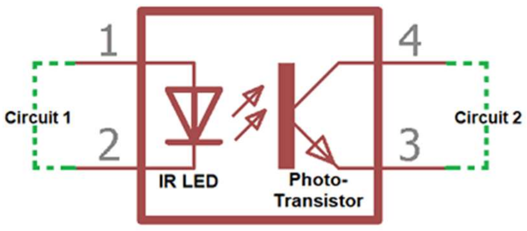
Serial monitor
```
2020-12-29 13:20:46: b'\x02CAVG\x04L\x00\x00\x00_\x03\r\n'
2020-12-29 13:20:46: b'\x02CURR\x01Ka\x03\r\n'
2020-12-29 13:20:46: b'\x02CAVG\x04K\x00\x00\x00^\x03\r\n'
2020-12-29 13:20:46: b'\x02CURR\x01Ka\x03\r\n'
2020-12-29 13:20:46: b'\x02CAVG\x04K\x00\x00\x00^\x03\r\n'
2020-12-29 13:20:47: b'\x02CURR\x01Ka\x03\r\n'
```

Finally we can give a look at the GUI that interfaces our PC with the System.

We can appreciate how many different value can be set by an operator and the plot that shows all the samples, the average and the three different thresholds.

Finally, we can give a look at the last piece of the entire system: the DC Motor Controller. Its job is to correctly handle the higher currents (compared to a normal signal circuit like the signals exchanged by our FPGA with the externa peripherals) needed to run correctly the two DC Motors.

The two DC motor used are two motor with a maximum velocity of 130 RPM. They work with a voltage that ranges from 3V to 6V and the maximum current is 140 mA more or less. In my circuit, they actually work with a regulated 5V outputted by an Arduino UNO (I didn't have other separate voltage sources except my FPGA and my PC and for sure they can't handle this very well, I used a wall adapter for smartphone recharging as power supply).



During the design stage of the circuit, the first I thought is that is better to separate electrically the two part of the entire system to remove possibly noise (and damages) provoked by the inductive load of the motors: the signal part (where there is the FPGA, the ADC, the sensor and so on) and the power part (where there are the two motors). To achieve this, I decided to use an optocoupler (a PC817P). And it resolved me another problem indirectly.

Another problem is: what kind of switch and configuration should I use to drive the two motors? Well, I decided to use two N-Channel MOSFETs with a heatsink in low side configuration. The only problem is that the only transistors I had are two IRF520 MOSFETs, that are not exactly thought to be driven with a TTL logic on the Gate but with a higher voltage (like 10 V to fully turn on them).

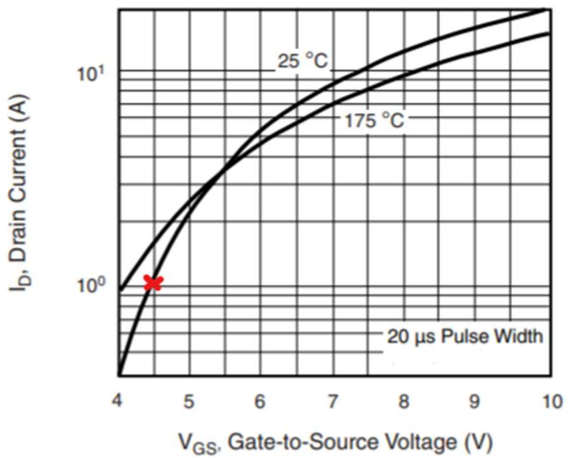


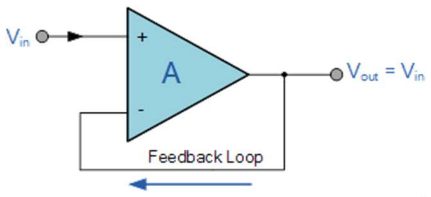
Fig. 3 - Typical Transfer Characteristics

But, fortunately, I decided to examine the datasheet in details, and I discovered that first the minimum V_{GS} to turn on them spaces from 2.0 V to 4.0 V even if the I_{DS} is not at its maximum.

But this is not bad. We know that our motors don't require more than 200 mA, and looking at the transfer characteristics we discover that with a $V_{GS} \cong 4.5$ we obtain a $I_{DS} \cong 1A$ that is even higher than what we need. It looks good!

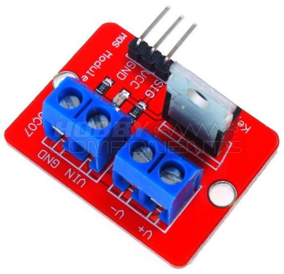
Now there's another problem: we want to drive the gate with the PWM signal from our FPGA, but 1- the maximum output voltage is 3.3 V and 2- the maximum current is 8 mA.

The output current is a problem because with the PWM signal we have to drive the IR led inside the opto-coupler and it's too low. A way to solve this is with a voltage follower.

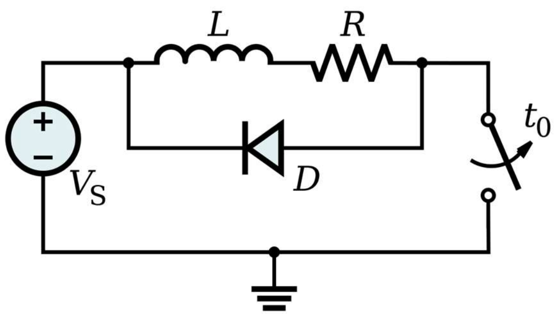


I realized it using another operational amplifier (the same model as the one used for the ADC). The input bias current is very low so I can drive it with my PWM signal without problem. Its Vcc is connected to the 3.3V of the FPGA, so the output will be much lower than this value because, as said, it's not a rail-to-rail operational amplifier, but it's not a problem. To turn the IR led inside the optocoupler I need around 10 mA, and the output voltage of the buffer is around 1.9 V – 2.0 V, so I putted a $100\ \Omega$ resistor between the output of the Op Amp and the input of the opto-coupler.

The output is a NPN photo-transistor and I used it in a way that negates the input signal (the PWM one). So I would like to 1- negate it again and 2- having a higher voltage then 3.3 V (we said that we need around 4.5 V). We can connect the collector through a pull up resistor to 5 V but the supported current is lower then what we need to turn on both the MOSFET then other additional circuitry connected to the MOSFET (actually the MOSFET I have available is installed on a breakout board with two resistors and a LED connected to the gate so I have to consider them in the circuit too).



So, to solve this last problem in order to obtain a higher voltage and the original PWM signal (without negations), I decided to put another signal NPN BJT transistor.

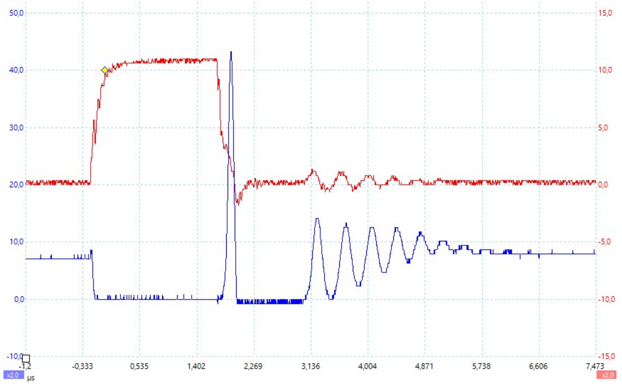


Last but not least, a Schkotty diode is added between the Drain and the Source of the MOS transistor, with the anode connected to the Source: this is called a "free-wheeling" diode.

This diode is connected across an inductor and its purpose is to eliminate flyback, which is the sudden voltage spike seen across an inductive load when its supply current is suddenly reduced or interrupted (like in this case where there is a fast PWM signal that continuously open and closes the "switch" where the switch is implemented by the MOSFET).

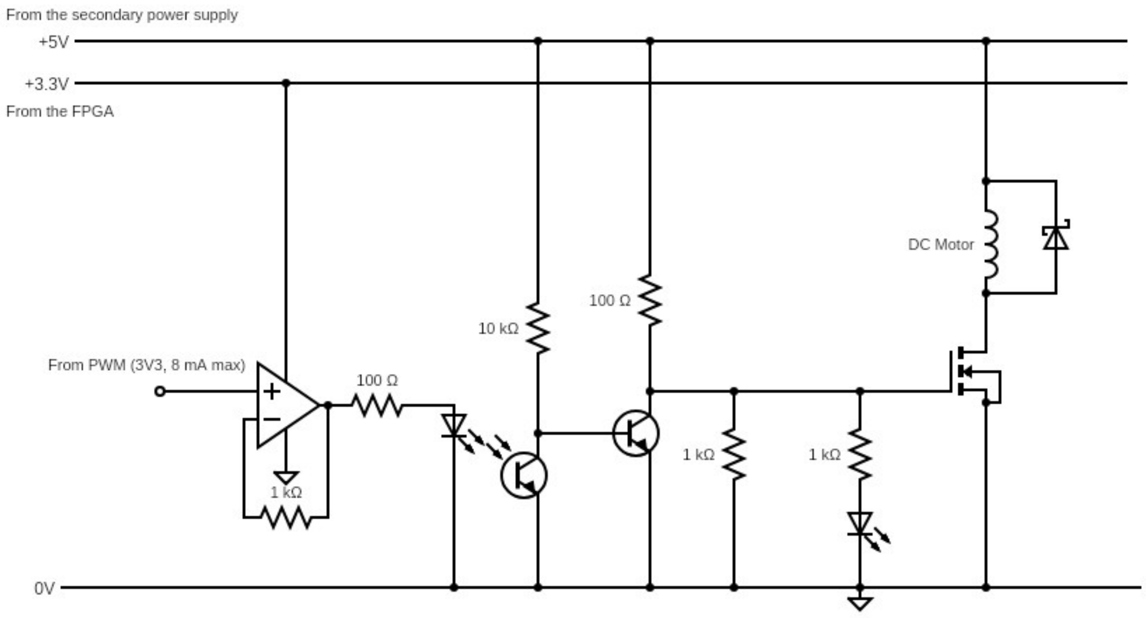
The cause of these voltage spikes can be seen in two different ways. The mathematical one: we know that the relation that describes the inductor is $V(t) = L \cdot \frac{d\,I(t)}{dt}$, so when the switch opens up the current ideally goes down (I am supposing a positive flow of the current) and its derivative becomes infinite large and negative and this means a voltage across the inductor that is high and negative and this can destroy the transistor because this current has no way to flow!

The easiest way to see this is that, because of the way on how a DC motor actually works, when it runs freely it generates current (if it is left floating). So, when the switch turns off, for the inertia it continues to run for a small amount of time and this generates the current (the same current seen by the mathematical point of view) that destroys the transitor. To solve this, a diode is connected to give the current a path to flow without problems (a Schkotty one, because we want fast switching).

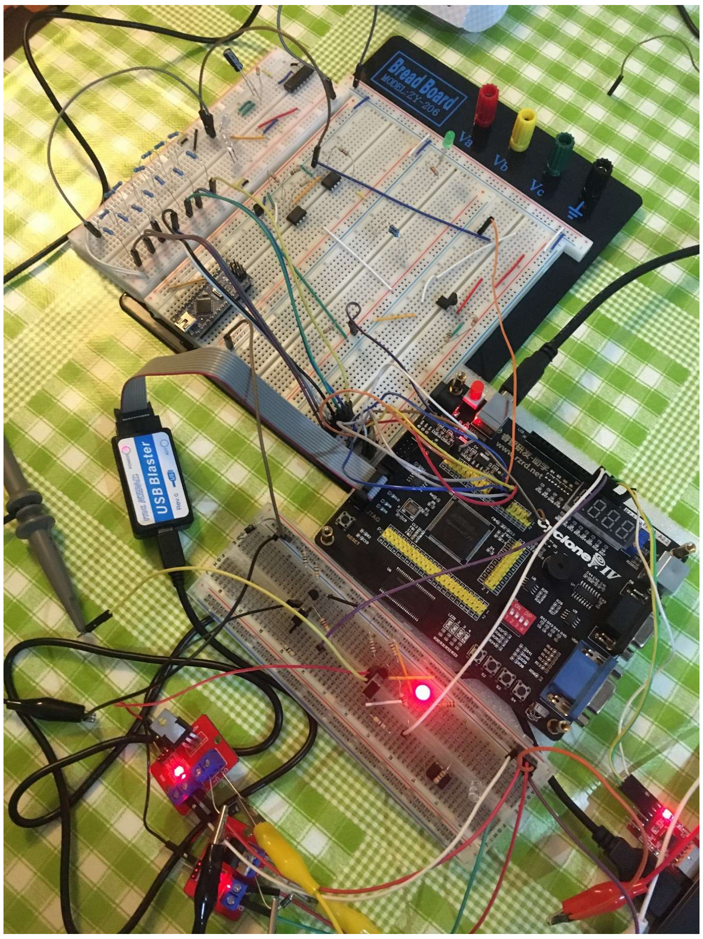


The final circuit

And finally, this is how the entire circuit looks like:



A proof that all the overall system is truly implemented



I want to conclude this report with a photo, a proof that the entire system is effective implemented!

On the bottom-right side is possible to see the UART TTL to USB Converter connected to my computer.

On the bottom left side is possible to see the two breakout board with the MOSFET and the diode connected directly to the board for simplicity. The two alligator clips (black and yellow) goes to the DC motor.

The small breadboard in the middle is the one containing the two DC Motor controller circuits.

The small breadboard on the top contains the ladder network of the DAC inside the ADC.

The big breadboard in the middle is there only for having a large space where place some components.