EFES Project

Room Temperature Controller

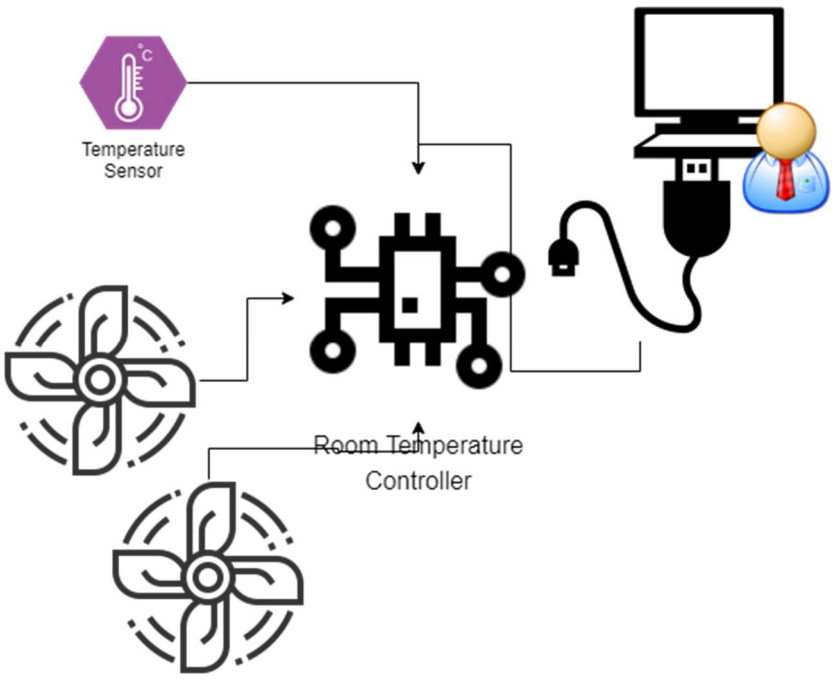
The project I am going to present is a Room Temperature Controller. It's a complex Embedded System composed of several parts that works together in order to maintain under control the temperature of a room (like a server room, university classroom, etc..).

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:

- Independents speed for two fan 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.

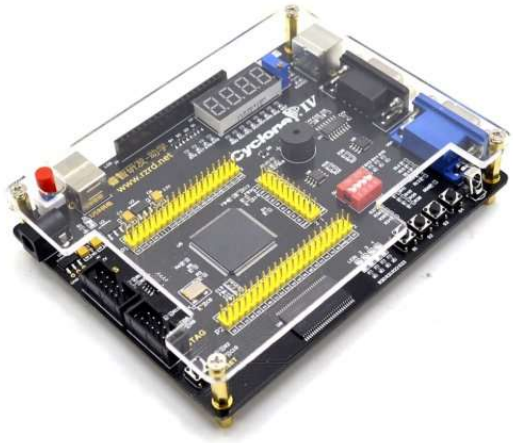
An operator can administrate the system through a USB port. A USB to UART converter is needed, thus the parameters used by the system to communicate through this interface are 19200 8N1.

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

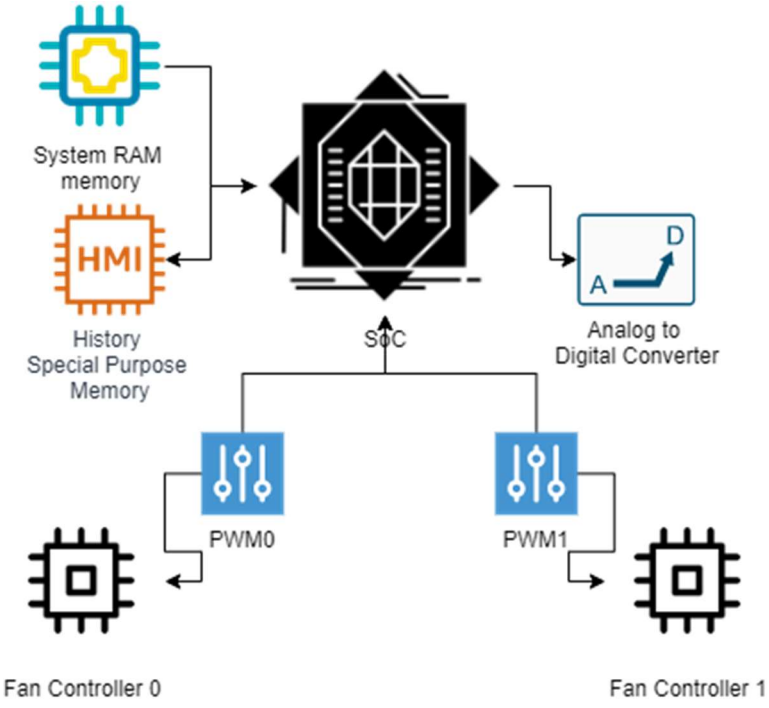
Now it's time to go deeper in analysing the system and how it works internally. The central controller it's the core of the system, and it's based on a System-On-Chip with different peripherals inside it and proprietary peripherals connected around it.



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.

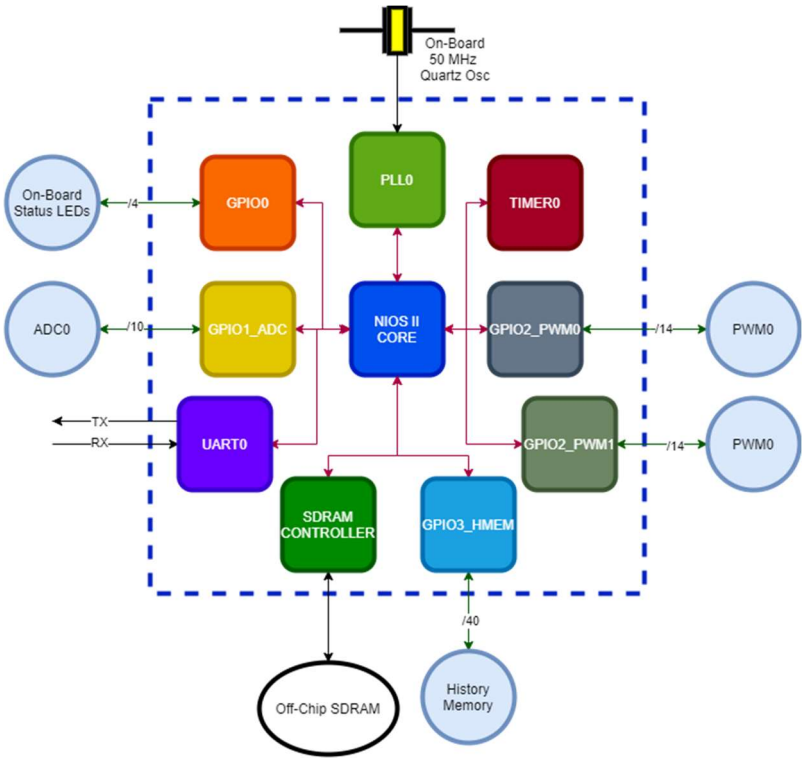
Those features are enough to implement what we need: A System-On-Chip, a special RAM memory for storing history, two PWM controllers and the logic of an Analog to Digital converter. We have enough pins to connect to the off-chip SDRAM and to external connections towards the breadboard with external electronic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

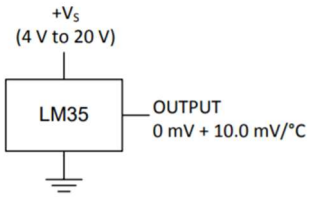
Through a PLL (PLL0), is possible to generate different clock signals starting from the external oscillator installed on the development board in use. It's a 50 MHz quartz oscillator and the PLL outputs 4 different clock signals for different uses. On of this is the frequency at which all the internal peripherals on the SoC and the core itself works 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.

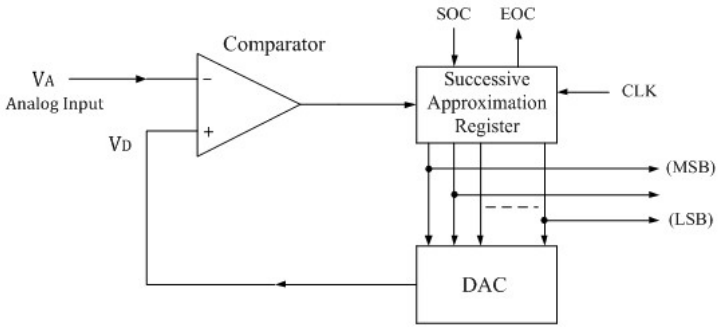
The home-made Analog to Digital converter is a SAR 7-bit ADC. The SAR logic is implemented, as said, in VHDL. Its job is to convert the analog output of the temperature sensor.



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.

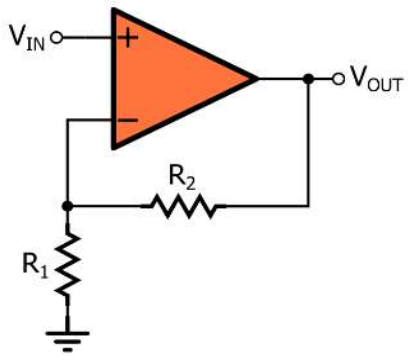
The input dynamic range of the implemented converter is from 0 V to 3.3 V, this means that we need to amplify the output of the sensor first, then convert it. The amplification is needed in order to have a more accuracy on the conversion, otherwise only a minimum subset of the entire dynamic range of 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!).

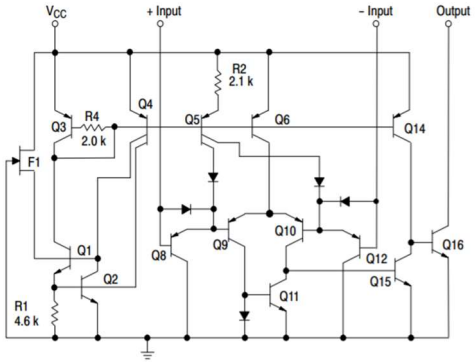
The amplifier stage of the ADC

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

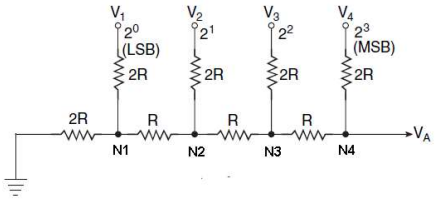


The comparator used is a LM393. This time it's a rail-to-rail comparator so we have to connect its power rail to 3.3V and not to 5V because, looking at the datasheet, the FPGA pins are rated to works at a maximum voltage of 3.6V! We need to use the output of the comparator as an input of our SAR logic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

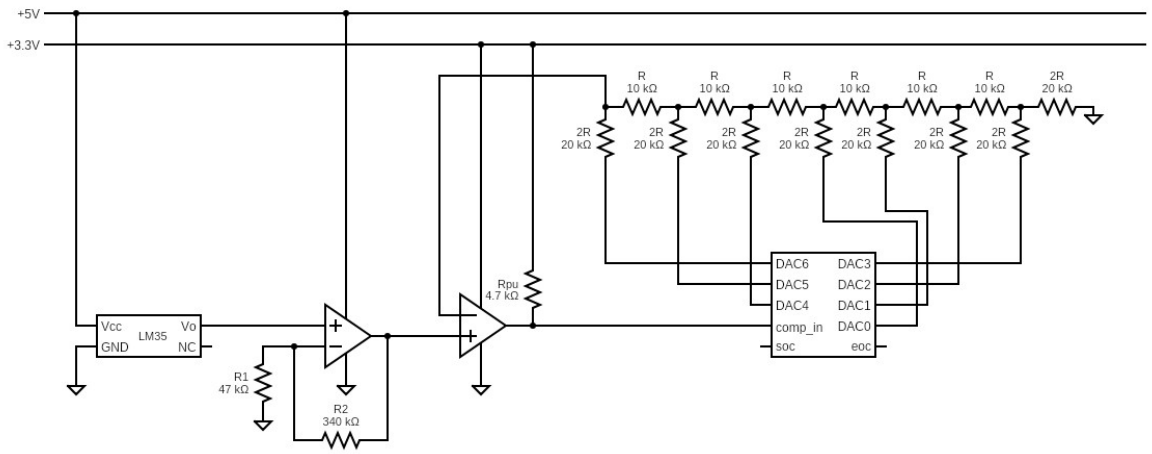
Finally, the last piece of the converter: the SAR logic. Implemented in VHDL on the FPGA, it's an entity with different control signals and I/O.

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.

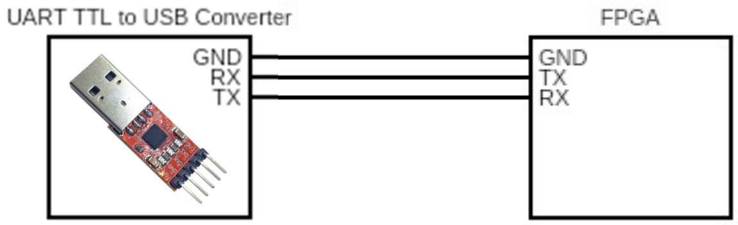
Internally, it's implemented as a High-Level State Machine (Moore Machine). There are mainly two states, one tries to guess putting a 1 at the current position while the second one puts at the current position the value that comes from the output of the comparator, then it goes forward repeating everything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



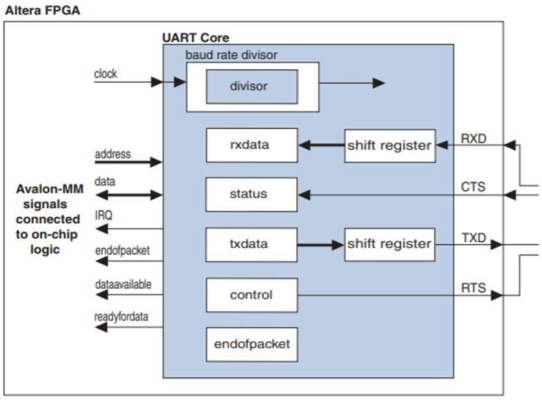
The serial connection



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).

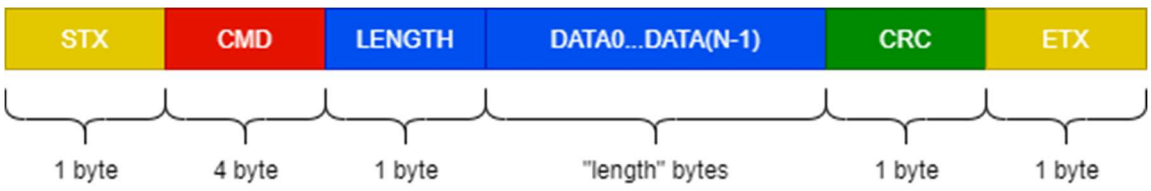
There is the choice to leave the baud rate configurable via software and I enabled it, in this way is possible to configure it via the drivers I wrote. This means that the baud rate is "softwired" at 19200.

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.

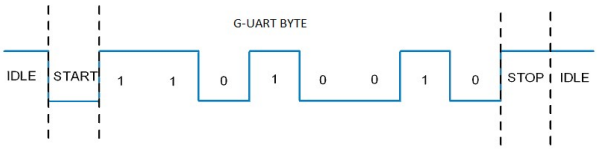


In details, we have:

- 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.
- 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 PWM peripheral

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 firmware that manages everything

Here describe the peripheral drivers, interrupts (maybe), and the MEMORY MAPPING ADDRESSES of the peripherals. Show how peripheral registers are mapped and their use in the drivers.

The history memory and sdram configuration

The DC Motor / FAN Controller