Design Features

The specific hardware requirements for this project were created in Verilog. These custom peripherals connected to the Nios II through a number of PIO ports. The peripherals were controlled by reading or writing specific memory addresses, similar to almost any other microcontroller. Writing the peripherals in Verilog allowed us to optimize them for the project.

Reports from other groups suggested that the motor encoder signal was noisy. To reduce these effects, we designed the encoder input to reject noise. This was done by enforcing a minimum time between pulses. The encoder input module outputs the period time directly. This allows the rpm to be read directly, without interrupts or calculating a difference.

The motor current is controlled with a single transistor. Initially we thought this was undesirable because there is a substantial difference in the acceleration and deceleration rates of the motor. This could be fixed in the case of a DC motor. Operating a heater would have a similar imbalance so the single transistor was decided to be a realistic option.

Step by step we show what the program does

test is the nios ii processor and PIO ports
the input capture module for reading motor encoder
reset counter when period time is reached
otherwise, increment counter
begin pwm generation
register comparation value to prevent noise
end pwm generation
We comment the code for a better explanation about what we did, so please look at the comments

```
/*
 * Nios II PID motor controller
 * Verilog code for input capture
 *
 * EE 554 midterm project
 * James Smith, Elias Badillo
 *
 */

module input_capture( clk, enc, deadtime , count , capture);
input clk;  // "clk" is the clock
input enc;  // encoder signal

input [31:0] deadtime; // expect enc to remain high for deadtime number cof clocks
output [31:0] count; // count is the counter value, made available to the nios ii
processor
output [31:0] capture;// capture is the time in clocks between rising edges of enc


reg [31:0] countreg; //
reg [31:0] capreg;        // stores counter value when enc has a rising edge

reg edgedetect;
// (enc | (countreg[31:0]<deadtime[31:0]))
// is true starting with the rising edge of enc
```

```
// stays true until deadtime is over or enc goes low
// This signal is a "filtered" version of enc
// it is forced high to avoid rising edges when deadtime is over

// edgedetect is the previous value of above

always @(posedge clk) // enc now high but was previously low
if( (enc | (countreg[31:0]<deadtime[31:0])) & !edgedetect )
begin // ran at rising edge of enc
    capreg=countreg; // record timer value
    countreg=0; // reset counter
       edgedetect=(enc | (countreg[31:0]<deadtime[31:0])); // save enc value for
use at next clock
end
else
begin // ran when no rising edge on enc
    countreg=countreg+1; // increment counter
     edgedetect=(enc | (countreg[31:0]<deadtime[31:0]));// save enc value for use
at next clock
end

// output signals
assign capture[31:0]=capreg[31:0];
assign count[31:0]=countreg[31:0];

endmodule
```

We designed the PID function with the equations (p.154 textbook):

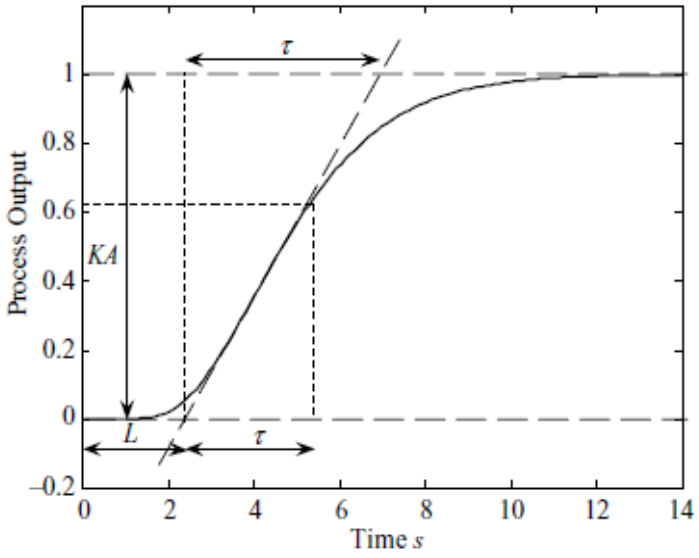
$$C(s) = K_p\left(1 + \frac{1}{T_i s} + T_d s\right)$$

$$T_i = K_p / K_i, \quad T_d = K_d / K_p$$

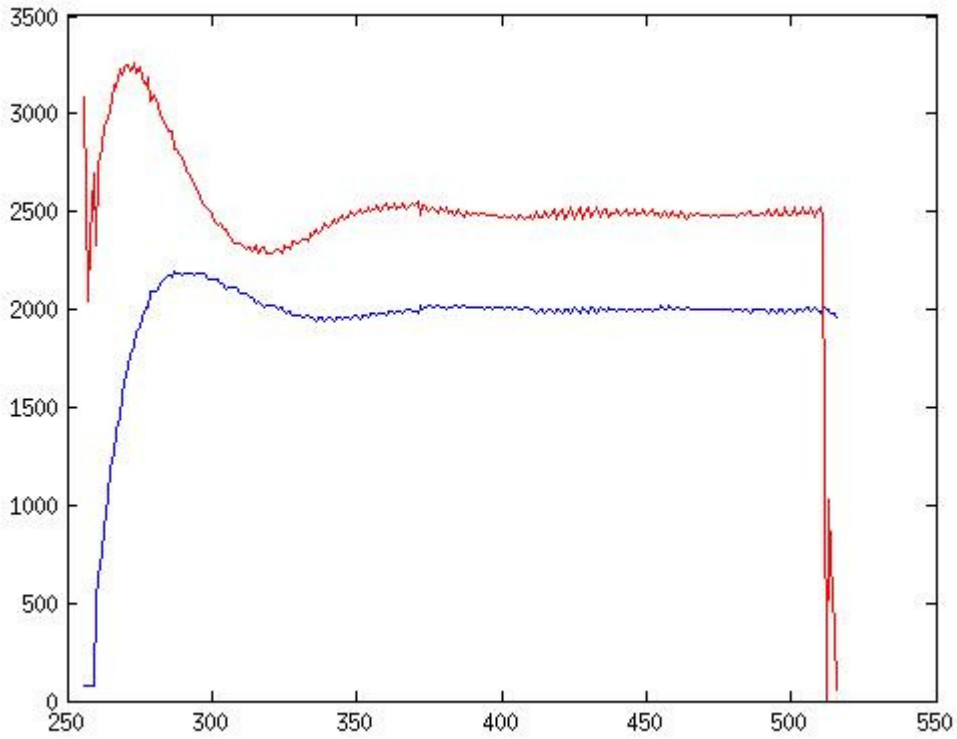
and

$$G(s) = \frac{K}{\tau s + 1} e^{-ls}$$

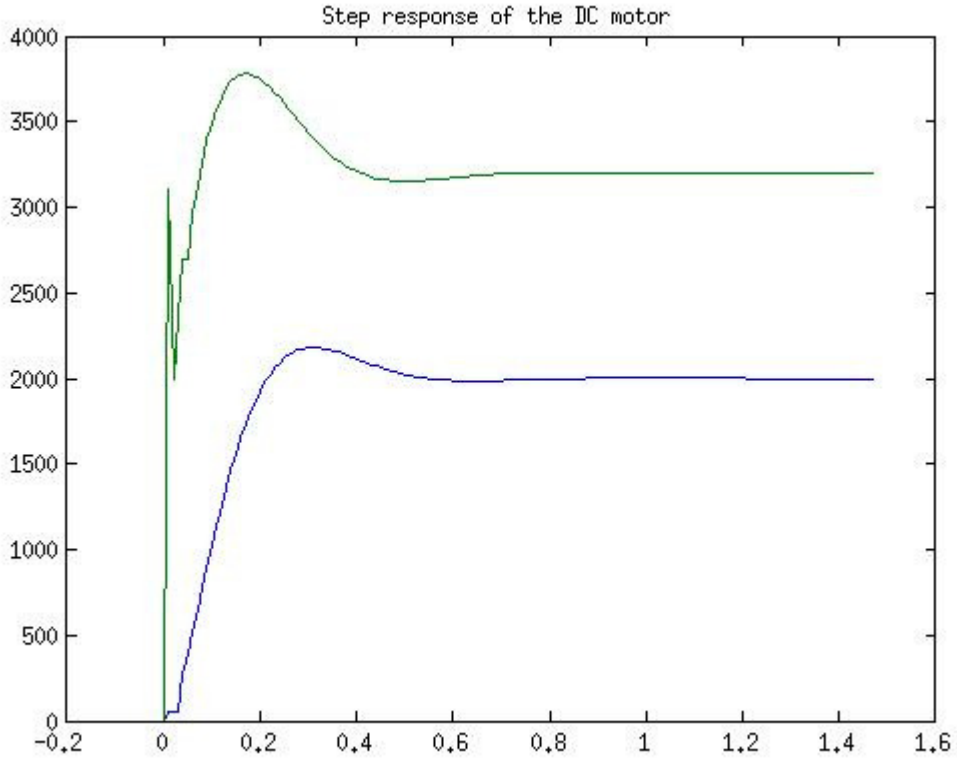
Taking some values from the system and seeing the next graphic



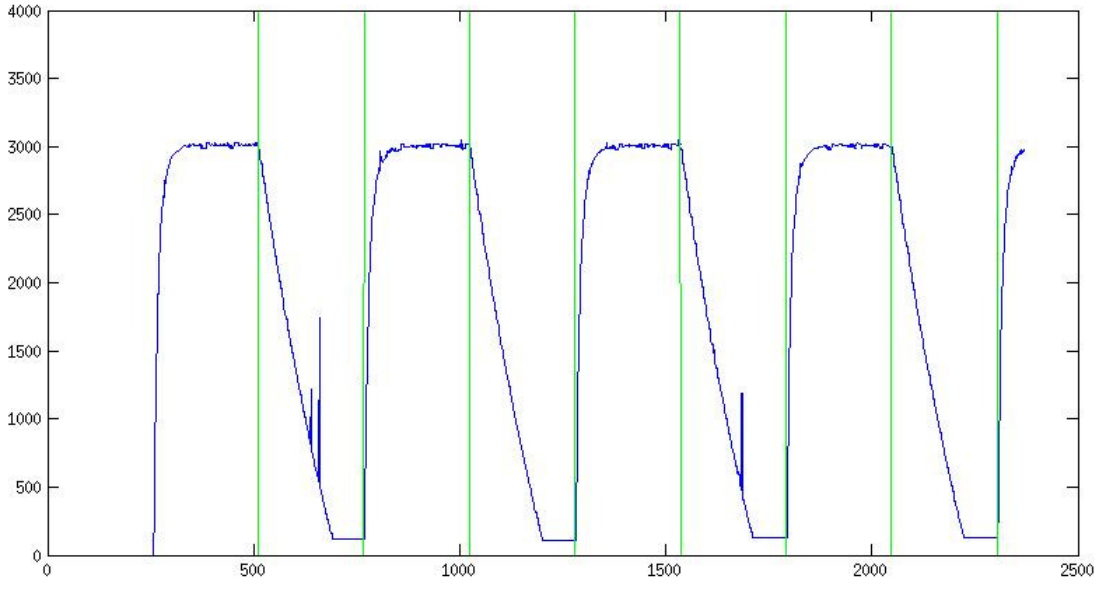

```
tau = 11.5000
kp = 0.6900
cb = 1.5525   -2.0700    0.6900
ca =1    -1     0
mdelay = 3
mtau = 12.5000
```



Step response without PID control



Step response using PID control



System response