ServoSim V 1.00

4-Channel Servomotor Simulator

Dr.Varodom Toochinda

http://www.controlsystemslab.com

July 2009

In this document we discuss an implementation of ServoSim, a circuit that simulates operation of 4-channel servomotors equipped with incremental encoders. This unit simply receives 4 analog speed commands and generates quadrature encoder signals and their complements, similar to what we get from real encoders. The design uses low-cost components, mainly a dsPIC30F2011 DSC (Digital Signal Controller) from Microchip, an XC9572XL CPLD from Xilinx, and a few op-amps. The DSC and CPLD communicate via SPI protocol. We provide C source code for C30 compiler and Verilog code for CPLD.

Servomotor Fundamentals

There are many sources that explain servomotor and incremental encoder operations. Here we briefly describe only the essence. Servomotors come in many types, from some simplest brushed DC (Figure 1) preferred by hobbyists, to some sophisticated, high-performance AC brushless servos used in industrial CNCs and robots(Figure 2).

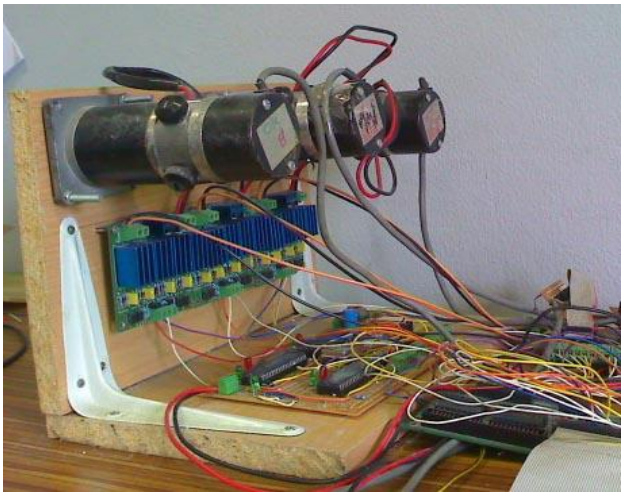


Figure 1: Brushed DC servomotors (Yasakawa Minertia Series)

Figure 2 AC Servomotors and drives (Panasonic)

In this simulator we assume the servo systems operated in velocity mode. When you send in analog signals in $\pm$ 10 V range, the motors rotate with speeds corresponding to the applied input voltages. The incremental encoders generate the quarature A,B, and index Z signals with frequencies related to the RPMs of motors. For those of you who are not familiar with quadrature encoders, Figure 3 could be helpful. The encoder disk has slots that pass the light beams from LEDs to receiving devices. This generates 3 main signals A, B, and Z. A and B are called quadrature because they have 90 degree phase differences. When the motor turns one direction, say, clockwise, the signal A leads B 90 degree. When it turns counter-clockwise, B leads A 90 degree. With this construction we could construct a circuit that detects the direction of rotation, in addition to the shaft position and speed.

In real industrial environment things are not perfect. There are evils like noise that contaminates the signal. So the signals A, B, Z are sent with their complements A/, B/, and Z/ that are 180 degree out of phase. When noise gets into these signals, the noise component in each signal tends to be similar, so when passed through differential amps the noise are rejected. This is called RS-422 standards.

Enough for the basics eh?. In case you're still confused, Mr.Google is around … ;)

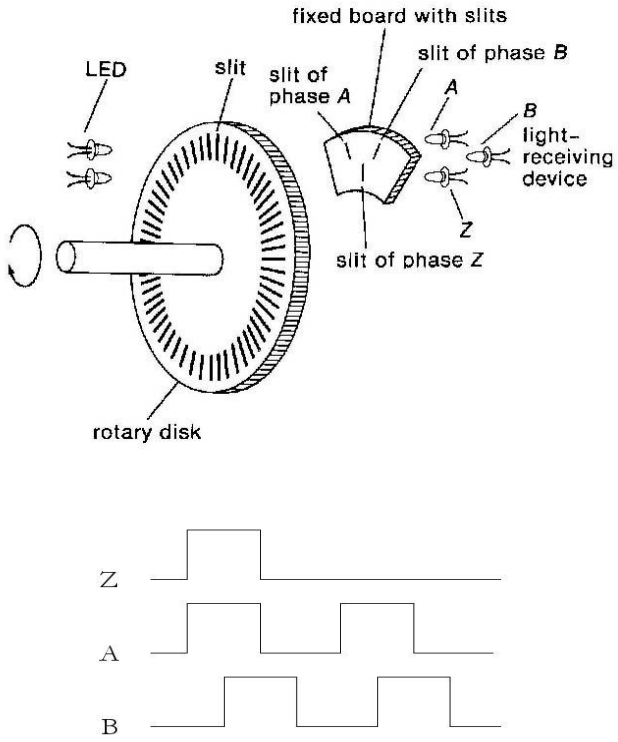


Figure 3: incremental encoder and quadrature signals

Why would one need a servo simulator?

Simple answers: It's portable, reliable, and cheap.

I have messed with control systems a lot in my life, especially motion control. I have CNC controller projects going on. During development I need to test the controller with "a plant," in this case 3-4 servomotors. These are bulky equipments. I often want to be mobile, sometimes working in lab, other time at home. It's problematic to carry 4 servomotors and drives along with me all the time.

When I assigned projects to students. They often bought old motors with worn brushes, crappy encoders, loose connectors, and these caused all sort of problems in their work. On the other hand, if I allowed testing a new design on expensive servos, chances were they could be damaged if something went wrong. Hence, during initial phase of development, you need a plant that is predictable, reliable, simple, and cheap. The simulator discussed here could be constructed on a proto board, the way I did it. It cost less than USD 20.

ServoSim Diagram and Functionality

Now we are ready to get into detail of the stupid simulator. The overall block diagram is shown in Figure 4. At a glance, an experienced designer would agree that this is no big deal, though one might wonder why wouldn't I make it simpler by using a microcontroller with enough I/O's and save the CPLD. Frankly, I have no good answer for that. My personal reason is I want to learn how to program the dsPIC SPI module, as well as to implement a slave SPI unit in the CPLD suing Verilog. To summarize the learning objectives of this project:

1. Construct signal conditioners using op-amps
2. Implement a simple slave SPI in the CPLD
3. Program the ADC and SPI module of dsPIC
4. Write real-time interrupt threads

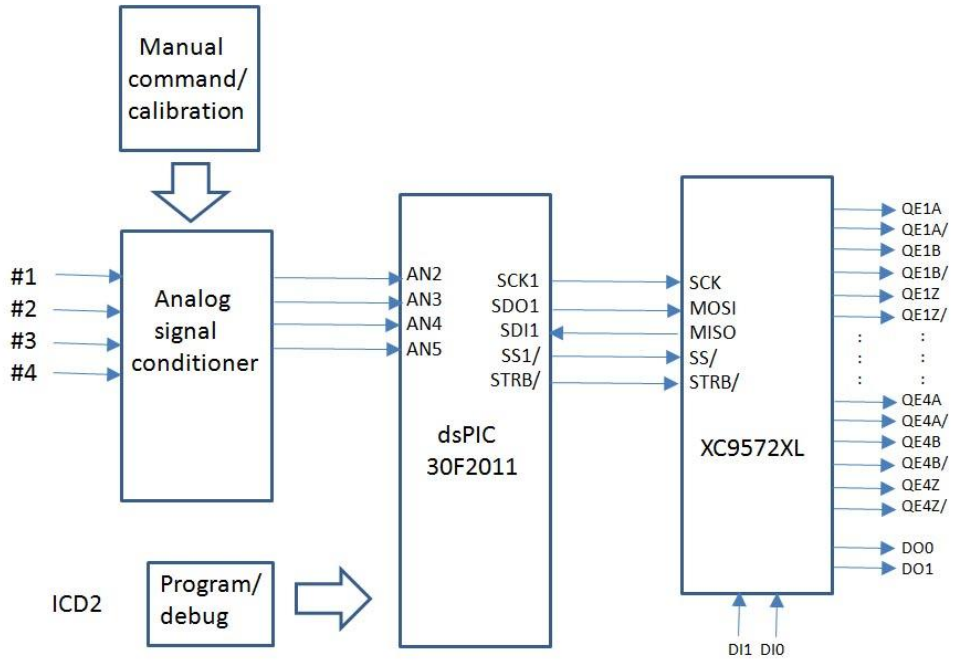


Figure 4: ServoSim Block Diagram

I don't know about you guys. But all of these learning objectives are vital and valuable for my future projects. I used parallel interface a lot in my previous work. It consumed many microcontroller pins that could be useful for other tasks. So I wanted to try serial

communications. I barely understood the various modes of SPI operations from looking at the timing diagram in the dsPIC documents alone. It became clear when I took the pain building the real hardware and fooled around with it. Yep it took a lot of time but worth the efforts.

Now we delve into each part individually.

Hello Analog World

Some people turn other way when talking about analog. Why bothers when everything seems to be labeled as digital nowadays? Nothing is far from the truth. Most things we encounter in daily life are continuous and have infinite values. To keep them under control, unfortunately, we have to digitize those analog quantities. Before that stage, we may need to condition the analog signals to the levels suitable for ADC modules, say, 0 – 5 volts (Talking about this, dsPIC has +Vref and –Vref pins for users' choice of analog level. But I couldn't find in the datasheet the maximum voltages these pins could handle. Anyone knows, email me please.) The voltage range used in industrial servo drives is normally $\pm$10 volts. The circuit in Figure 5 converts the $\pm$10 volts input range from a controller to 0 – 5 volts, before feeding the signal to AN2 – AN5 pins. (I made this in Powerpoint. Due to limited space, channel 3 is omitted. It's the similar to channel 1, 2 though.)

One caution: before using the circuit in Figure 5, the user must calibrate it properly. The calibration procedure is as follows. First, adjust V9 to give -2.5 volts at center tap. Then, with IN1 – IN4 at + 10 volts, adjust V1 – V4 to give 1.25 volts at center tap. This should make the voltages at outputs AN2 – AN5 equal 5 Volts. The calibration is now complete. Test by sweeping IN1 – INC4 between $\pm$ 10 volts and see the voltages at AN2 – AN5 changes between 0 – 5 volts.

If you fail to calibrate, the voltages at analog inputs of dsPIC could exceed the range at AVdd and AVss (I supply 5 volts and GND to them) and could damage the inputs. If you have children running around in the lab, maybe it's good idea to use fixed resistors and 2.5 volt reference ICs.

I add jumper J2 to bypass the circuit if driven by 0 – 5 volt DACs. J1 is used for manual control by pots V5 – V8.

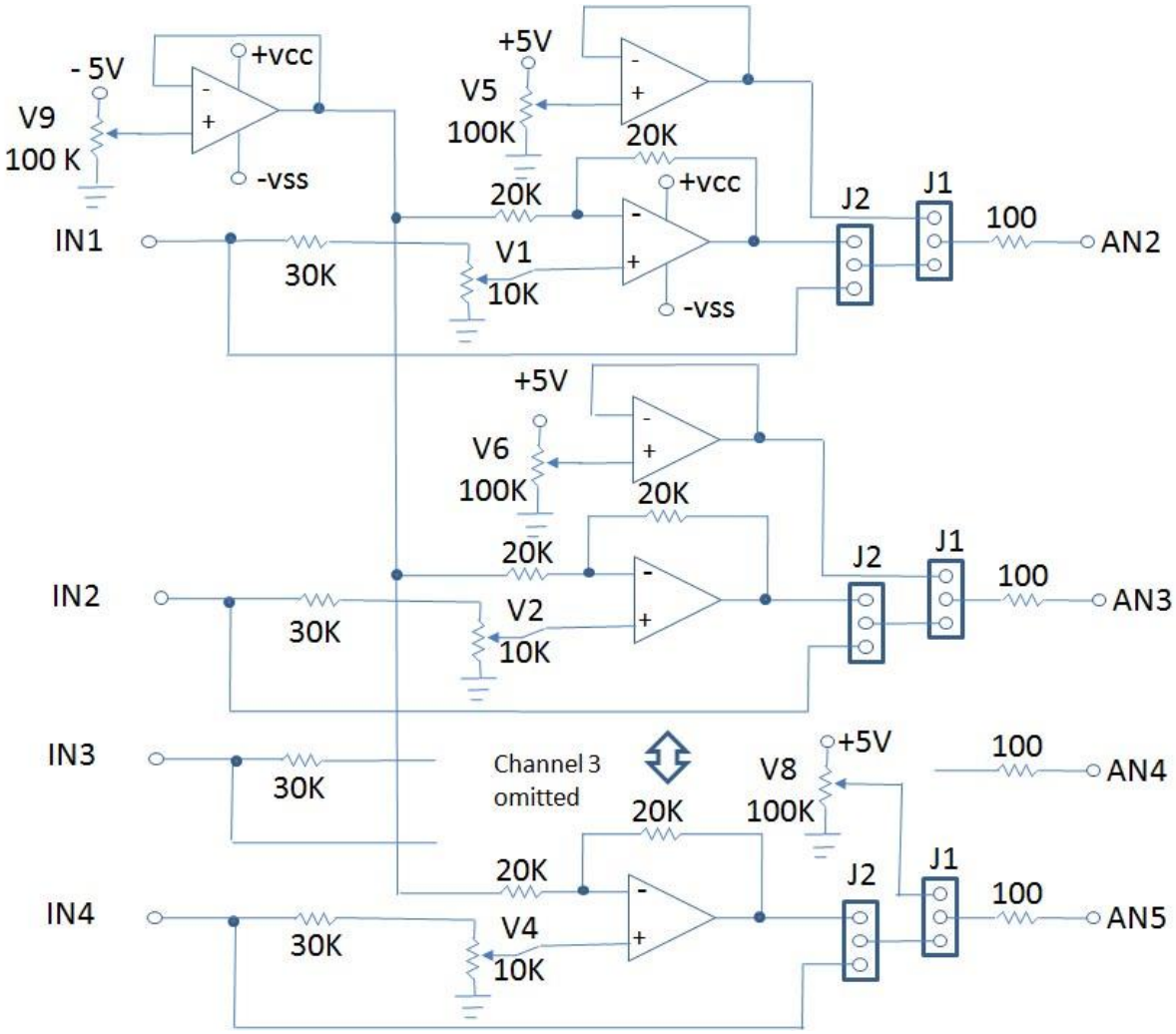


Figure 5: Analog Signal Conditioning

Suggestions:

1. For easy calibration, an op-amp that gives ± 10 volts output could be added to the circuit.

2. If ServoSim is driven by noisy analog signals, you may consider adding low-pass filters to get rid of the noise.

SPI Slave Module in CPLD

The SPI slave module used in this project is shown in Figure 6. It consists of a shift register that receives serial data from the master via MOSI pin and shift data out of MISO pin at the same time. This operation is synchronized by SCK signal generated by the master. The SS/ signal marks the start and end of transmission of a 16-bit word. Since we need too many outputs than 16, two words have to be sent. Two latches, labeled High 16 and Low16, are used to capture the data received and sent to CPLD output pins. STRB/ is used to reset the word selection to initial state before transmitting.

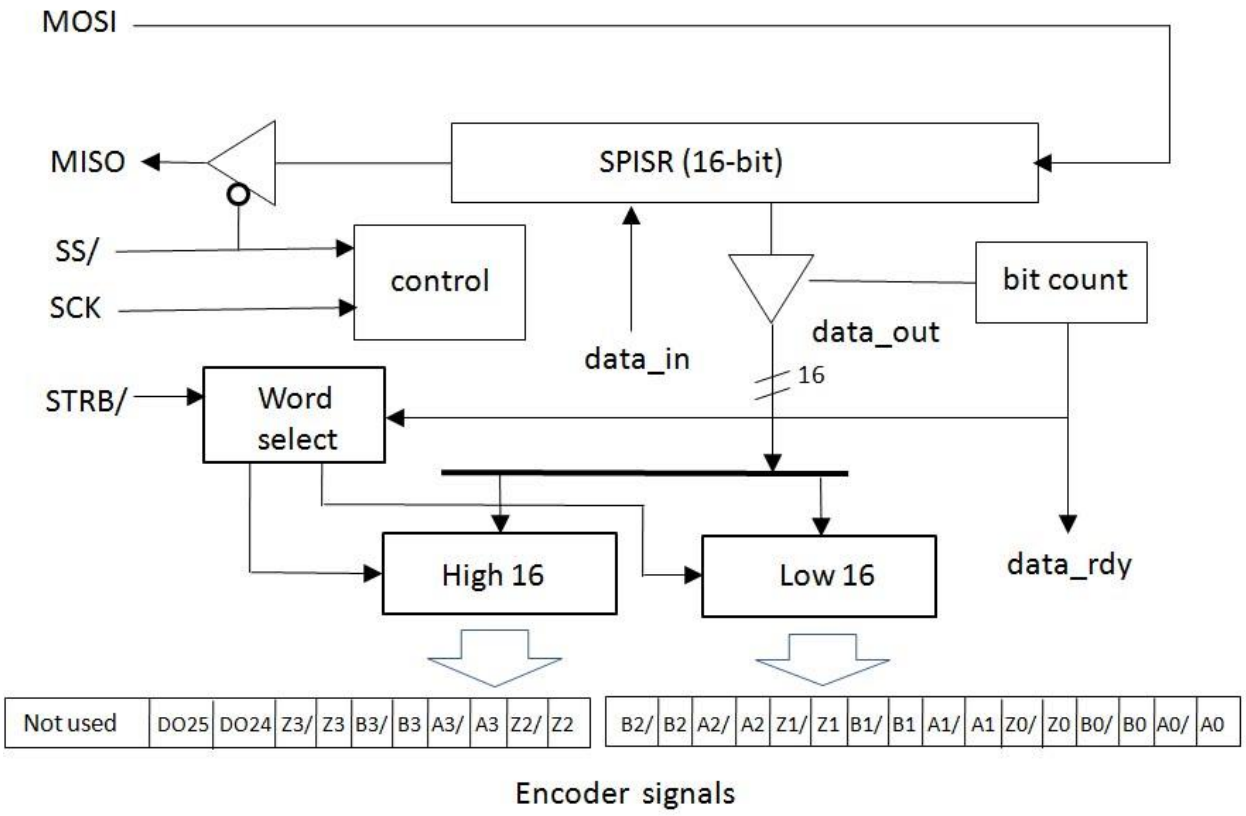


Figure 6: Block diagram of SPI slave module

In this application we do not really need to send data to the host. Anyway, I just want to test if the SPI module could shift data out properly, so that it can be used in other applications that require full-duplex communication. So I assign 2 LSBs of data_in to CPLD pinouts. This is used for selecting “encoder resolution” to 2048, 4096, 8192, or 16394 pulses per revolution.

As I said before, there are a few selectable modes of SPI operation. Do not be overwhelmed by this information. Just pick up a configuration and construct the slave module to give signals and timings conformed to those shown in the datasheet. For this project, I select CKP = 1, CKE = 0, and SMP = 1 (What the heck are these? Ask Microchip.) This gives the timing diagram shown in Figure 7.

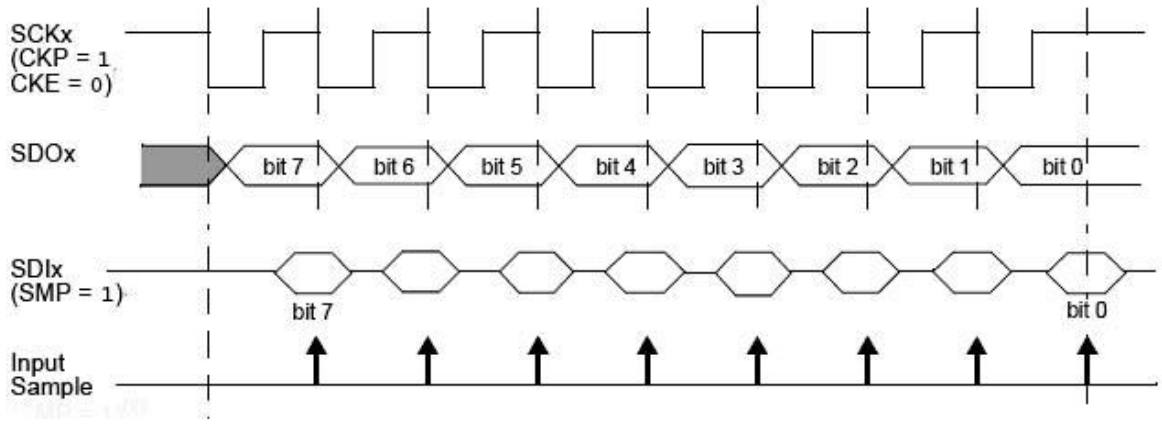


Figure 7: SPI timing diagram (CKP =1, CKE = 0, SMP = 1)

Now we are ready to start writing Verilog code. To give proper credit, I started by looking at www.fpga4fun.com and modifying the code provided on that website. (The code almost works as is, except that it is designed for 8-bit SPI, and the MISO signal does not conform to the timing digram in Figure 7. The data is shifted out too early.) Note that in this implementation the CPLD system clock is used to sample the SCK signal. One disadvantage is that the CPLD clock frequency has to be faster than SCK. It works ok when I use 10 MHz crystal for CPLD and program SPI to run at 2.5 MHz.

So here is the Verilog code for SPI slave module, in its entirely.

```
module spi_slave(
    input clk,
    input SSEL,
    input SCK,
```

```
    input MOSI,
    input [15:0] data_in,
    output MISO,
    output [15:0] data_out,
    output data_rdy
    );

// sync SCK to the FPGA clock using a 2-bits shift register
reg [1:0] SCKr;
wire SCK_risingedge = (SCKr==2'b01);  // now we can detect SCK rising edges
wire SCK_fallingedge = (SCKr==2'b10);  // and falling edges
reg [15:0] SPISR;                     // SPI shift register

// same thing for SSEL
reg [1:0] SSELr;
wire SSEL_active = ~SSELr[1];  // SSEL is active low
wire SSEL_start = (SSELr==2'b10);  // message starts at falling edge
wire SSEL_end = (SSELr==2'b01);  // message stops at rising edge

// and for MOSI
reg [1:0] MOSIr;
reg MISOr;
wire MOSI_data = MOSIr[1];
```

```
// we handle SPI in 16-bits format, so we need a 4 bits counter to count
// the bits as they come in
reg [3:0] bitcnt;
reg done;
reg [1:0] rdelay;                    // delay for data_rdy

always @ (posedge clk)
begin
        SCKr <= {SCKr[0], SCK};
        SSELr <= {SSELr[0], SSEL};
        MOSIr <= {MOSIr[0], MOSI};
        if (SSEL_start)
        begin
                SPISR <= data_in;       // parallel load to shift-register
        end
        if(~SSEL_active)
        begin
                MISOr <= 1;
                bitcnt <= 0;  // 4'b0000;
                done = 0;
                rdelay <= 0;
        end
        else
                if(SCK_risingedge)
```

```
            begin
            // implement a shift-left register (since we receive the data MSB first)
            MISOr <= SPISR[15];
            SPISR <= {SPISR[14:0], MOSI_data};
            bitcnt <= bitcnt + 1;
            if (bitcnt==15) done=1;
        end
        if (done && ~rdelay[1]) rdelay <= rdelay+1;
end

assign MISO = (SSEL_active) ? MISOr: 1'bz;          // send MSB first
assign data_out = (done) ? SPISR: 16'bz;            // parallel output
assign data_rdy = SSEL_active && rdelay[1];
endmodule
```

If SS/ is high, the MISO and data_out are in high-impedance state. The module starts operation when SS/ goes to low. It receives serial data from MOSI and shifts the content of SPISR out of MISO, starting from MSB. This operation is synchronized with the positive edge of SCK. Note that data does not appear at data_out until all 16 bits are received and data_rdy is asserted high. I add some delay to data_rdy output because I use its positive edge to latch data_out to CPLD output pins. So data_out has to be stable at that moment.

The top module servosim.v and the rest are trivial. I just list them here for completeness.

```
// SERVOSIM.V  Servomotor simulator
module servosim(
```

```
    input clk,
    input SSN,
    input MOSI,
    output MISO,
    input SCK,
    input STRBN,
    input [1:0] DI,
    output [25:0] DO
    );

wire [15:0] spi_dout;
wire [13:0] spi_din ;
wire data_rdy;
wire [31:0] outport;
wire [15:0] inport;
wire sel_l, sel_h;                  // select high or low word

assign DO = outport[25:0];
spi_slave spi(clk, SSN, SCK, MOSI, {spi_din,DI}, MISO, spi_dout, data_rdy);
do16simple do16L(spi_dout, clk, data_rdy, sel_l, outport[15:0]);          // 16-bit lower
do16simple do16H(spi_dout, clk, data_rdy, sel_h, outport[31:16]);         // 16-bit upper
wcount2                 wc2(data_rdy, STRBN, {sel_h,sel_l});  // word selection
endmodule
```

```
// DO16SIMPLE.V   a simple 16-bit latch
module do16simple(
   input [15:0] inp,
   input clk,
   input enable,
   input seln,
   output reg [15:0] outp
   );
reg done;
always @ (negedge seln) done <= 0;
always @ (posedge clk)
begin
if (~seln && enable && ~done)
         begin
         outp <= inp;
         done <= 1;
         end
end
endmodule

// WCOUNT2. V   selection of 2 16-bit words
module wcount2(
   input wcnt,
   input resetn,
```

```
    output [1:0] selout
    );
reg count;
always @ (negedge wcnt, negedge resetn)
begin
        if (resetn==0) count <= 0;
        else count <= ~count;              // just toggle
end
assign selout[0] = count;
assign selout[1] = ~count;
endmodule
```

These Verilog files are added to project SERVOSIM.ISE. They are tested with Xilinx ISE 10.1 /ModelSim XE III 6.3c. The last thing is to assign CPLD pins to our signals. Table 1 summarizes the pin assignment.

XC9572XL pins	Signal name	XC9572XL pins	Signal name
1	DI0	24	DO18 (4A)
2	MISO	25	DO17 (3Z/)
3	MOSI	26	DO16 (3Z)
4	SSN	27	DO15 (3B/)
5	clk	28	DO14 (3B)
6	DO23/ (4Z/)	29	DO13 (3A/)
7	DO22 (4Z)	33	DO12 (3A)
8	DO5 (1Z/)	34	DO11 (2Z/)

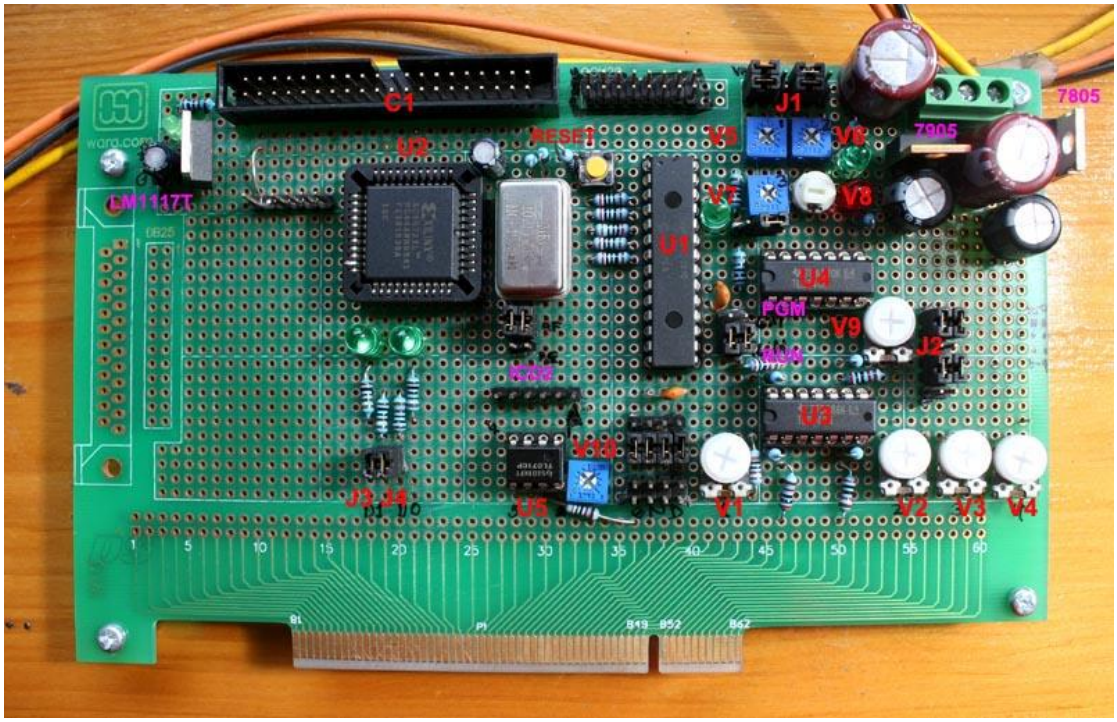
9	DO4 (1Z)	35	DO10 (2Z)
11	DO3 (1B/)	36	DO9 (2B/)
12	DO2 (1B)	37	DO8 (2B)
13	DO1 (1A/)	38	DO7 (2A/)
14	DO0 (1A)	39	DO6 (2A)
18	DO25	40	STRBN
19	DO21 (4B/)	42	DI1
20	DO20 (4B)	43	DO24
22	DO19 (4A/)	44	SCK

Table 1: XC9572XL pin assignment

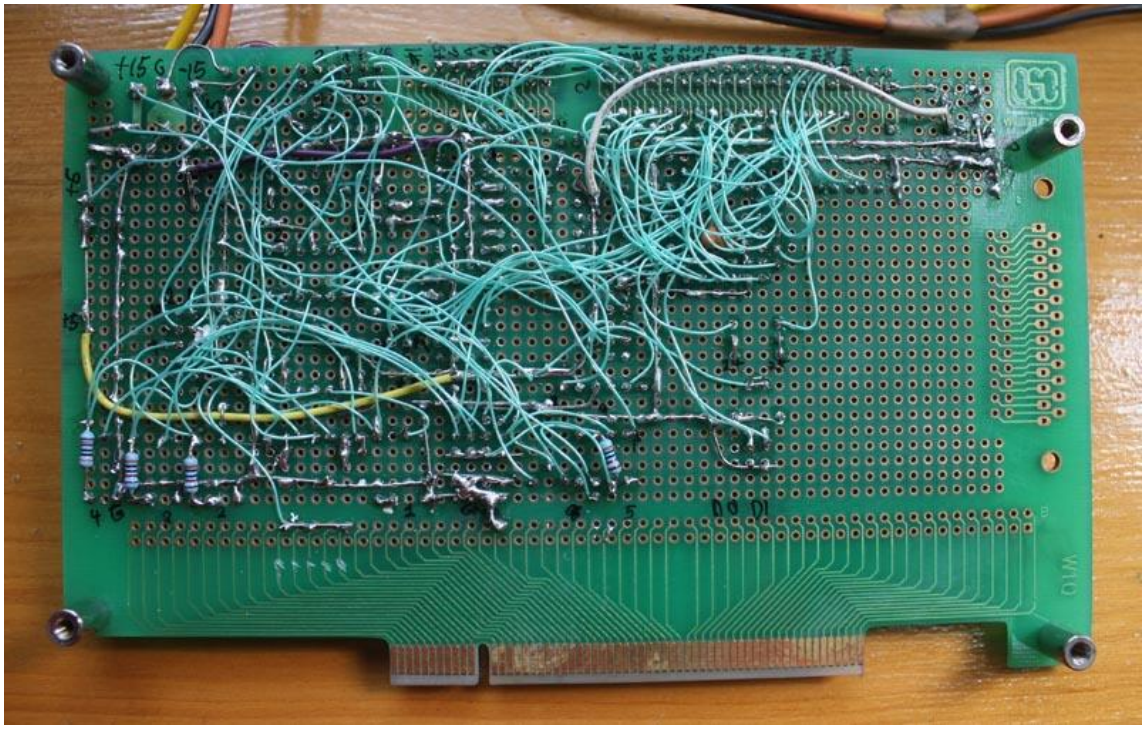
Construct the prototype

After the CPLD pins are defined, we can build the hardware. If you are good at PCB design, start right away. This circuit is not too complicated and the clock frequency not so high, so I could, with some patience, wire the components on a general purpose board. Figure 8 shows the top and bottom view of my prototype. You might think how something looking so messy could actually work. It does! I was surprised myself LOL.

Whatever method you use, take into consideration the basics of component placement and signal routing. I use the same crystal for both the CPLD and dsPIC clock signals. They are placed close together. Also keep the SPI signal paths short. Add small capacitors across the supply rail near the dsPIC and CPLD.



Top view



Bottom view

Figure 8: ServoSim prototype

Program the dsPIC

The last step, and perhaps the most entertaining, is to write a C program for this embedded application. You could figure out right from the start that, however simple and small, our ServoSim is a real-time, multi-threaded system. That is where the fun starts.

To simplify the discussion, we consider the case when there is only one servo motor. This can be extended easily to 4-axis motor simulation by looping for each motor. Figure 9 shows the simplified flowcharts of the two threads involved. The idea is to use infinite loop in the main thread to read the A/D value and compute a loop variable corresponding to the value. The motor movement is done in the timer interrupt. Every time the timer ISR starts, it will decrement the loop variable and check if it equals zero. If that's the case, the motor is moved and encoder signals sent via the SPI. So the frequency of the update depends on the value of loop variable, which is dictated by the analog command applied to ServoSim.

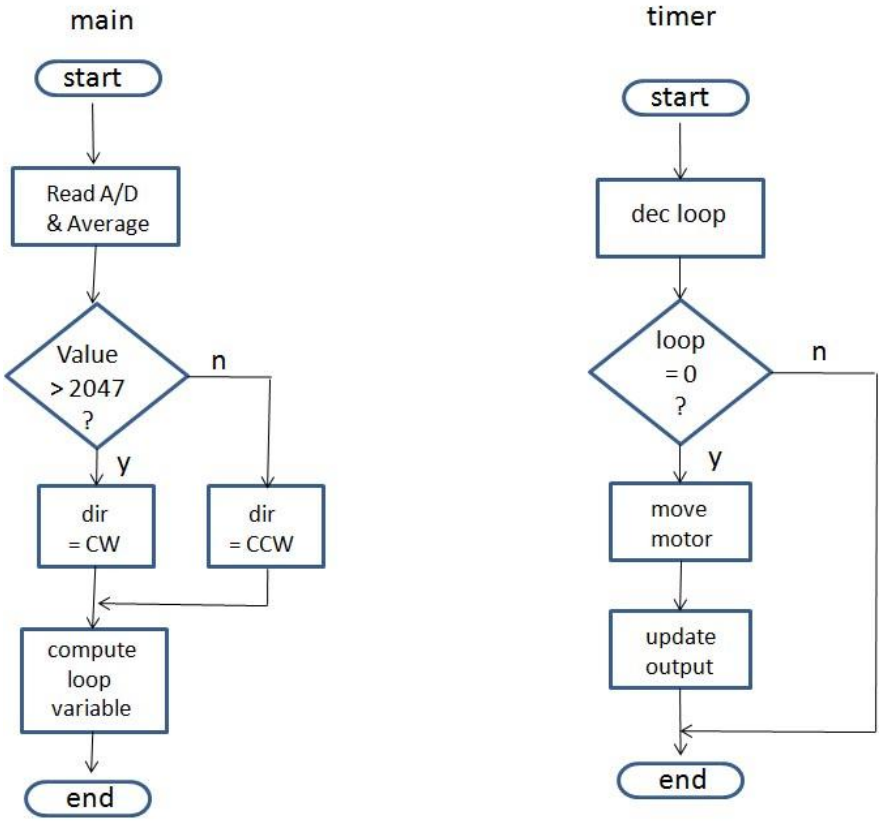


Figure 9: flowcharts of ServoSim

One important thing to keep in mind when writing a multi-threaded program is how to protect the critical section. In our case it is the loop variable. You don't want the timer thread to read a loop value at the same time the main thread is writing to it. The easiest way is to make the writing process "atomic"; i.e., disable the interrupt while writing. I use the macro code suggested in the C30 manual to protect the critical section.

There are details in initializing the dsPIC resources and peripherals, such as I/O pins, timer, ADC and SPI modules. You can consult the datasheets or reference book for details. Here I just briefly address them.

ADC: The 4 ADC pins, AN2 – AN5, are scanned sequentially, 4 times each, and the results kept in ADBUF0 – ADBUF15. The timing of sampling and conversion is controlled automatically by timer3. After it is done, 4 samples from each analog pin are read and averaged. This helps filter out noise. (Note that in the main loop, the value read is further averaged with previous 3 values. That is done to mimic the motor inertia. A motor is a mechanical apparatus. It responds to a command much slower than its simulated electronic counterpart. So this second-stage averaging just adds delay to the dynamics.)

SPI: The SPI1CON is set to value 0x67E. This selects 16-bit mode, CKP = 1, CKE = 0, SMP = 1, and SCK frequency equals one fourth of dsPIC system clock. For this simple application, polling method works fine. That is, after initialization, you operate the SPI module by writing a 16-bit value to SPIBUF, wait for a flag to complete, and then read the received value from SPIBUF. One point worth mentioning: the SPI did not work properly when I put a value into SPIBUF. It took me quite a long time to find out. I searched the net until found it somewhere (not from Microchip!) that I first had to read out some garbage stuck in the SPIBUF, to empty that register, before writing a value to it.

Timer: I use timer2 to perform the motor movement. The period is set to 70 microseconds. You could use this value to compute the max/min RPM of the motors the simulator could produce. Now I have one timer left, Timer 1. To add life to the module, I just set it up to blink an LED each second. At first, my purpose was only just that. I liked to see LED blinking. But it turned out to have some usefulness. After I experimented with the module by making timer2 period smaller, at some point the program just froze when I commanded the 4 motors to move at fastest speeds. The reason was, Timer2 just couldn't finish the job before the next interrupt occurred.

I knew things stopped working because my LED also stopped blinking. Oh how I love the LED.

I think there is not much left to discuss about the program. Here I provide it in full.

```
/*  ----------------------  SERVOSIM.C by Dr.Varodom Toochinda  -------------------------------

        Created May 09, 09

        This program simulates a 4-axis servo. It is commanded by analog values on pin RB2, RB3,
        RB4, RB5

        The output is sent to CPLD XC9572XL via SPI1. Input DI<1:0> selects encoder resolutions

        Output words:


        (HW)  x   x   x   x |  x   x DO25 DO24 | Z4/ Z4 B4/ B4 | A4/ A4 Z3/ Z3

        (LW)  B3/ B3 A3/ A3 | Z2/ Z2  B2/  B2  | A2/ A2 Z1/ Z1 | B1/ B1 A1/ A1

 ------------------------------------------------------------------------------  */


#include <math.h>

#include <stdlib.h>

#include <p30f2010.h> // Header file for dsPIC30F2010

#include <ports.h>                //  Module function for Interrupt configuration port

#include <adc10.h>


//-----------------------------------------------------------------------

// Configuration bits
```

```
  _FOSC(CSW_FSCM_OFF & XT_PLL4);   //XT with 4xPLL oscillator, Failsafe clock off
  _FWDT(WDT_OFF);                  //Watchdog timer disabled
  _FBORPOR(PBOR_OFF & MCLR_EN);    //Brown-out reset disabled, MCLR reset enabled
  _FGS(CODE_PROT_OFF);             //Code protect disabled
//---------------------------------------------------------------------------
#define FCY  10000000                       // xtal = 10Mhz; PLLx4
#define NCHAN        4                      // number of channels
#define CW    0                             // clockwise
#define CCW   1                             // counter-clockwise
#define LED   _RE1
#define SPI_MASTER16 0x067E                       // Master 16-bit CKP=1 CKE=0
#define SPI_ENABLE 0x8000                         // Enable SPI port, clear status
#define        SS1N          _RE0
#define        STRBN         _RE3
#define        TSS1N         _TRISE0
#define        TSTRBN        _TRISE3

// macro from C30 user guide
#define INTERRUPT_PROTECT(x) { \
char saved_ipl; \
\
SET_AND_SAVE_CPU_IPL(saved_ipl,7); \
x; \
RESTORE_CPU_IPL(saved_ipl); } (void) 0;
```

```
// ---------- global variables ----------

//////// Timer ////////////////////

double T1Period = 1;            // period of timer 1

int T1Prescale = 256;           // timer1 prescale

double T2Period = 0.00007;          // period of timer 2

int T2Prescale = 1;             // timer2 prescale


////// ADC ////////

unsigned int ADval[NCHAN]={2048,2048,2048,2048};                    // new AD values

unsigned int ADval1[NCHAN]={2048,2048,2048,2048}, ADval2[NCHAN]={2048,2048,2048,2048},
ADval3[NCHAN]={2048,2048,2048,2048};            // keep old values

unsigned int ADval_avg[NCHAN];                      // keep average values


/////// motor simulator ///////

unsigned int sscale=0;          // scale the encoder resolutions 1,2,4,8 according to DI<1:0>

unsigned int ppr=2000;          // pulse per revolution, also depends on DI<1:0>

int di0, di1;               // keep status of di


// encoder output signals A, B, Z and their complement's

unsigned int A[NCHAN]={0,0,0,0};

unsigned int B[NCHAN]={0,0,0,0};

unsigned int Z[NCHAN]={0,0,0,0};

unsigned int AN[NCHAN]={1,1,1,1};

unsigned int BN[NCHAN]={1,1,1,1};
```

```
unsigned int ZN[NCHAN]={1,1,1,1};


int pcount[NCHAN]={0,0,0,0};                                    // position counters
int rev[NCHAN]={0,0,0,0};                                  // keep number of revolutions
unsigned int dir[NCHAN]={CW,CW,CW,CW}, dir1[NCHAN]={CW,CW,CW,CW};
// direction of rotations
unsigned int mperiod[NCHAN]={0,0,0,0};                                  // motor period
unsigned int cloop[NCHAN], nloop[NCHAN];                                // loops for each channel
int changeFlag = 0;                // flag output change
union {
        int hw[2];        // half word = 16 bits
        long fw;          // full word = 32 bits
} spiout;          // it is not really necessary to define this as union. But the technique is handy to
                   // relate 16 and 32-bit words.


// ------------------- function declarations -------------------
// ----------- ADC functions --------------
void InitADC(void);
void LoadADC(unsigned int offset);
void AverageADC(void);


// -------------- SPI functions ----------------
void initSPI1(void);
int writeSPI1(int);
```

```
// ------------------- timer functions ---------------

void initTimer1(void);

void initTimer2(void);

void initTimer3(void);   // use for ADC


// ------------------ general functions ---------------

void strobe(void);                          // send strobe signal

void init(void);

void move_enc(int);

void update(void);


// ------------------- function definitions ----------------------------------------

// --------------------- SPI routines -------------------------------

void     initSPI1(void)

{

         TSS1N = 0;                         // make SS1N pin output

         SS1N = 1;                          // SS signals are active low

         SPI1CON = SPI_MASTER16;            // select mode

         SPI1STAT = SPI_ENABLE;             // enable the modules

}


int      writeSPI1(int data) // send one word of data and receive one back at the same time

{
```

```
        int dummy;
        SS1N = 0;                       // assert SS1
        dummy = SPI1BUF;            // dummy read of the SPI1BUF register to clear the SPIRBF
                                        // flag.  Without this line the SPI might not work!
        SPI1BUF = data;                 // write data to Tx buffer
        while( !SPI1STATbits.SPIRBF);   // wait for completion
        SS1N = 1;                       // deselect the slave
        return  SPI1BUF;                // return the received value
}


// ------------Timer routines ----------------------
void _ISR _T1Interrupt(void)
{
  IFS0bits.T1IF = 0;          //Clear Timer1 interrupt flag
        LED = 0;          //Turn LED on
  while(IFS0bits.T1IF == 0);              //Wait for next Timer1 interrupt flag
  IFS0bits.T1IF = 0;          //Clear Timer1 interrupt flag
        LED = 1;
}
void     initTimer1(void)          // timer1 initialization
{
        T1CON = 0;                 //Turn off Timer1 by clearing control register
        TMR1 = 0;                 //Clear timer1
        PR1 = (FCY/T1Prescale)*T1Period;
```

```
        switch (T1Prescale) {
                case 1:
                        T1CON = 0x8000;
                        break;
                case 16:
                        T1CON = 0x8010;
                        break;
                case 64:
                        T1CON  = 0x8020;
                        break;
                case 256:
                        T1CON = 0x8030;
                        break;
        }
        // Set Timer1 interrupt priority and enable interrupt
        _T1IP = 5;              //Set Timer1 interrupt priority level to 5
        _T1IF = 0;              // Clear Timer1 interrupt flag
        _T1IE = 1;              // Enable Timer1 interrupt
}
void    initTimer2(void)        // timer2 initialization
{
        T2CON = 0;               //Turn off Timer1 by clearing control register
        TMR2 = 0;               //Clear timer1
        PR2 = (FCY/T2Prescale)*T2Period;
```

```
        switch (T2Prescale)  {
                case 1:
                        T2CON = 0x8000;
                        break;
                case 16:
                        T2CON = 0x8010;
                        break;
                case 64:
                        T2CON  = 0x8020;
                        break;
                case 256:
                        T2CON = 0x8030;
                        break;
        }
        // Set Timer2 interrupt priority and enable interrupt
        _T2IP = 6;              //Set Timer1 interrupt priority level to 6
        _T2IF = 0;              // Clear Timer1 interrupt flag
        _T2IE = 1;              // Enable Timer1 interrupt
}

void initTimer3(void)           // used for ADC sample & conversion
{
        T3CON = 0x0010;                         //Tcy/8 internal clock
        TMR3 = 0;
```

```
        PR3 = 0x2400;                       // time out for 10 mS
}

void InitADC(void)
{
 _TRISB1 = 1; _TRISB3 = 1; _TRISB4 = 1; _TRISB5 = 1;  // set as input pins
 ADPCFG = 0x0003;                   // xxxxxxxxxx00_0011select RB2, RB3, RB4, RB5 as analog
 ADCON1 = 0x0044;                   // Auto convert using TMR3 sample for 10 mS ...
                                    // then convert
 ADCON2 = 0x043C;           // 0000_0100_0011_1100 scan inputs and interrupt after 16 samples
 ADCSSL = 0x003C;           // 0000_0000_0011_1010 Scan inputs: AN5, AN4, AN3 and AN2
 ADCON3 = 0x000F;                   // TMR3 = 10ms , Tad = 8Tcy = 1uS
ADCON1bits.ADON = 1;                    // turn ADC ON
}

/*********************************************************************
        AverageADC() reads 4 10-bit values and gives results as 12-bit
        That's why I comment out the >>2 lines!
*********************************************************************/
void AverageADC(void)
{
        ADval[0] = ADCBUF0 + ADCBUF4 + ADCBUF8 + ADCBUFC;
        //ADval[0] = ADval[0] >> 2;
        ADval[1] = ADCBUF1 + ADCBUF5 + ADCBUF9 + ADCBUFD;
```

```
        //ADval[1] = ADval[1] >> 2;
        ADval[2] = ADCBUF2 + ADCBUF6 + ADCBUFA + ADCBUFE;
        //ADval[2] = ADval[2] >> 2;
        ADval[3] = ADCBUF3 + ADCBUF7 + ADCBUFB + ADCBUFF;
        //ADval[3] = ADval[3] >> 2;
}

void strobe(void)               // send strobe signal
{
        STRBN = 0;
        STRBN = 1;
}

void init(void)  // do all initializations here in one place
{
        int datain,i;
        for(i=0;i<NCHAN;i++)  {                 // initialize loop variables to 100 iterations
                cloop[i] = 100;
                nloop[i] = 100;
        }
        initSPI1();
        _TRISE1 = 0;            // LED
        TSS1N = 0;
        TSTRBN = 0;
```

```
TRISJP01 = 1;
STRBN = 1;              // strobe signal active low
spiout.fw = 0;
strobe();
datain = writeSPI1(spiout.hw[0]);          // get resolution selection from DI<1:0>
datain = datain & 0x0003;
switch (datain)
{
        case 0:                         // fastest
                sscale = 0;
                ppr = 16383;
                di0 = 0;
                di1 = 0;
                break;
        case 1:
                sscale = 1;
                ppr = 8191;
                di0 = 1;
                di1 = 0;
                break;
        case 2:
                sscale = 2;
                ppr = 4095;
                di0 = 0;
```

```
                di1 = 1;
                break;
        case 3:                         // slowest
                sscale = 3;
                ppr = 2047;
                di0 = 1;
                di1 = 1;
                break;
    }
    datain = writeSPI1(spiout.hw[1]);           // just clear all outputs
    changeFlag = 0;
    initTimer1();
    initTimer2();
    initTimer3();
    InitADC();
    T3CONbits.TON = 1; //turn on TMR3
}

// --------------- Timer 2 ---------------------
// --- The encoder signals are generated here
void _ISR _T2Interrupt(void)
{
    int i;
    IFS0bits.T2IF = 0;          //Clear Timer2 interrupt flag
```

```
        for (i=0; i<NCHAN; i++)  {
                cloop[i]--;                 // decrement loop variable
                if (cloop[i] == 0)   {
                        move_enc(i);            // move the motor
                        cloop[i] = nloop[i];    // load new loop value
                        changeFlag = 1;         // set the change flag
                }
        }
        if (changeFlag)  {
                update();                   // write new output to CPLD
                changeFlag = 0;
        }
}


// Move motors relative to state and direction
void move_enc(int i)
{
        if(dir[i] == CW)  {       // rotate clockwise
                if(A[i]== 0 && B[i] == 0) {A[i] = 1; AN[i] = 0;}
                else if(A[i] == 1 && B[i] == 0) {B[i] = 1; BN[i] = 0;}
                else if(A[i] == 1 && B[i] == 1) {A[i] = 0; AN[i] = 1;}
                else if(A[i] == 0 && B[i] == 1) {B[i] = 0; BN[i] = 1;}
                pcount[i]++;
                if (pcount[i] > ppr) {pcount[i] = 0; rev[i]++;}
```

```
        }
        else if(dir[i] == CCW)  {          // rotate counter-clockwise
                if(A[i]== 0 && B[i] == 0) {B[i] = 1; BN[i] = 0;}
                else if(A[i] == 0 && B[i] == 1) {A[i] = 1; AN[i] = 0;}
                else if(A[i] == 1 && B[i] == 1) {B[i] = 0; BN[i] = 1;}
                else if(A[i] == 1 && B[i] == 0) {A[i] = 0; AN[i] = 1;}
                pcount[i]--;
                if (pcount[i]<0) {pcount[i] = ppr; rev[i]++;}                    }
        if (pcount[i] == 0) {Z[i] = 1; ZN[i] = 0;}  // set index pulse
        else {Z[i] = 0; ZN[i] = 1;}
}

// Update the spiout variable
//      spiout.hw[1]:  x   x   x   x  | x   x   DO25  DO24  | Z[3]/ Z[3] B[3]/ B[3] | A[3]/ A[3]
// Z[2]/ Z[2]
//      spiout.hw[0]:  B[2]/ B[2] A[2]/ A[2] | Z[1]/ Z[1]  B[1]/  B[1]  | A[1]/ A[1] Z[0]/ Z[0] | B[0]/
// B[0] A[0]/ A[0]
// I just echo the state of <DI1, DI0> to <DO25, DO24>

void update(void)
{
        int dummy;
        spiout.hw[0] = (BN[2]<<15) + (B[2]<<14) + (AN[2]<<13) + (A[2]<<12) + (ZN[1]<<11) + (Z[1]<<10)
                                + (BN[1]<<9) + (B[1]<<8) + (AN[1]<<7) + (A[1]<<6)
                                + (ZN[0]<<5) + (Z[0]<<4) + (BN[0]<<3) + (B[0]<<2) + (AN[0]<<1) + A[0];
```

```
        spiout.hw[1] = (di1<<9) + (di0<<8) + (ZN[3]<<7) + (Z[3]<<6) + (BN[3]<<5) + (B[3]<<4)
                                + (AN[3]<<3) + (A[3]<<2) + (ZN[2]<<1) + Z[2];
        strobe();
        dummy = writeSPI1(spiout.hw[0]);
        dummy = writeSPI1(spiout.hw[1]);
}

//--------------------------------Main Program-------------------------------------------------------------
int main()
{
        int i;
        init();
        while(1)  {
                while (!IFS0bits.ADIF);          // conversion done?
                IFS0bits.ADIF = 0;                       // clear flag
                /** once the conversion is completed, get the data **/
                AverageADC();                            //get average ADC value from buffer
                for (i=0; i<NCHAN; i++)
                {
                        ADval_avg[i] = (ADval[i]+ADval1[i]+ADval2[i]+ADval3[i]) >> 2;
                        if (ADval_avg[i] > 2047) {INTERRUPT_PROTECT(dir[i] = CW);}
                        else {INTERRUPT_PROTECT(dir[i] = CCW);}
                        mperiod[i] = 2050 - abs(ADval_avg[i] - 2047);
                        INTERRUPT_PROTECT(nloop[i] = mperiod[i]<<sscale);
```

```
                    ADval3[i] = ADval2[i];

                    ADval2[i] = ADval1[i];

                    ADval1[i] = ADval[i];

                }

        }

        return 0;

}
```

Summary

In this document, we discuss implementation of ServoSim, a 4-axis servomotor simulator. The prototype works fine. I can use it to accompany my CNC controller development. I only need the unit to generate quadrature encoder signals with frequencies and phases corresponding to the command analog values. If you feel the urge to improve it, here are some suggestions:

- You could make this simulator corresponded to a mathematical model of a real DC motor. $P(s) = \frac{K}{(S+\tau)}$. Use Z-tranform and modify the program to suit the resulting transfer function.
- It would be more flexible if you add communication with a PC. The serial-port pin is still available on dsPIC.
- You can add more input commands, such as more choices of encoder resolutions, adjusting the motor inertia. Etc.
- How about implementing servomotor simulator that runs in "position-mode"? What is done in real servo drives is to have closed-loop PID control in it.

References

[1] dsPIC30F Family Reference Manual, Microchip Technology Inc., 2006.

[2] dsPIC30F2010 datasheet, Microchip Technology Inc., 2004.

[3] L.D. Jasio, Programming 16-bit Microcontrollers in C, Elsevier, 2007.

[4] MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs User ‘s Guide. Microchip Technology Inc., 2008.