#include <tistdtypes.h>

#include <coecsl.h>

#include "28335\_spi.h"

#include "user\_includes.h"

long SPIbyte1,SPIbyte2,SPIbyte3,SPIbyte4,SPIbyte5;

long SPIenc1\_reading = 0;

long SPIenc2\_reading = 0;

long SPIenc3\_reading = 0;

long SPIenc4\_reading = 0;

int SPIenc\_state = 0;

int SPIenc\_state\_errors = 0;

unsigned int dac1data = 0;

unsigned int dac2data = 0;

void init\_SPI(void){

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\* SPI \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

The below code sets up several general purpose registers to outputs. It then sets the GPIOs to high, which is sent to four LS7366s, to their SS inputs (5V). This initiates the transmission cycle between the microcontroller and the LS7366s.

/\*

EALLOW;

GpioCtrlRegs.GPAMUX1.bit.GPIO9 = 0;

GpioCtrlRegs.GPAMUX1.bit.GPIO10 = 0;

GpioCtrlRegs.GPAMUX1.bit.GPIO11 = 0;

GpioCtrlRegs.GPAMUX2.bit.GPIO22 = 0;

GpioCtrlRegs.GPADIR.bit.GPIO6 = 1;

GpioCtrlRegs.GPADIR.bit.GPIO9 = 1;

GpioCtrlRegs.GPADIR.bit.GPIO10 = 1;

GpioCtrlRegs.GPADIR.bit.GPIO11 = 1;

GpioCtrlRegs.GPADIR.bit.GPIO22 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO6 = 1;

GpioDataRegs.GPASET.bit.GPIO9 = 1;

GpioDataRegs.GPASET.bit.GPIO10 = 1;

GpioDataRegs.GPASET.bit.GPIO11 = 1;

GpioDataRegs.GPASET.bit.GPIO22 = 1;

EDIS;

InitSpiaGpio();

EALLOW;

// SS for DAC7564

GpioCtrlRegs.GPAMUX2.bit.GPIO19 = 0; // use GPIO19 for SS

GpioDataRegs.GPASET.bit.GPIO19 = 1; // enabled

GpioCtrlRegs.GPADIR.bit.GPIO19 = 1; // GPIO19 as output

EDIS;

/\*

This code gets the transmit and receive buffers ready, as well as communication between the microcontroller and the LS7366 boards (master is MC, slaves are LSs). It clears the buffers, sets them up as queues (FIFO), and clears and readies their interrupts.

\*/

SpiaRegs.SPICCR.bit.SPISWRESET = 0; // Put SPI in reset (to change settings)

SpiaRegs.SPICCR.bit.CLKPOLARITY = 0; // set for LS7366

SpiaRegs.SPICTL.bit.CLK\_PHASE = 1;

SpiaRegs.SPICCR.bit.SPICHAR = 7; // set to transmitt 8 bits

SpiaRegs.SPICTL.bit.MASTER\_SLAVE = 1;

SpiaRegs.SPICTL.bit.TALK = 1;

SpiaRegs.SPICTL.bit.SPIINTENA = 0;

SpiaRegs.SPISTS.all=0x0000;

SpiaRegs.SPIBRR = 39; // divide by 40 2.5 Mhz

SpiaRegs.SPIFFTX.bit.SPIRST = 1; // SPI resume transmit or receive

SpiaRegs.SPIFFTX.bit.SPIFFENA = 1; // FIFO enhancements

SpiaRegs.SPIFFTX.bit.TXFIFO = 0; // transmit FIFO reset, hold in reset

SpiaRegs.SPIFFTX.bit.TXFFINTCLR = 1; // clear TXFFINT flag

SpiaRegs.SPIFFRX.bit.RXFIFORESET = 0; //Reset FIFO pointer to 0, hold in reset

SpiaRegs.SPIFFRX.bit.RXFFOVFCLR = 1; // clear RXFFOVF flag (which indicates overflow or not)

SpiaRegs.SPIFFRX.bit.RXFFINTCLR = 1; // Clear RXFFINT flag (indicates interrupts from fifo)

SpiaRegs.SPIFFRX.bit.RXFFIL = 5; // trip interrupt after 5 things

SpiaRegs.SPIFFRX.bit.RXFFIENA = 0; // comparing will be disabled

SpiaRegs.SPIFFCT.all=0x00;

SpiaRegs.SPIPRI.bit.FREE = 1;

SpiaRegs.SPIPRI.bit.SOFT = 0;

SpiaRegs.SPICCR.bit.SPISWRESET = 1; // Pull the SPI out of reset

SpiaRegs.SPIFFTX.bit.TXFIFO=1; // turn reset off

SpiaRegs.SPIFFRX.bit.RXFIFORESET=1; // same

/\*

This section resets the encoder counts to zero, sets filter clock division factor to 2, and sets them up for x4 quadrature mode. It does this by enabling transmit by pulling the SS\ inputs low and writing the command to clear, then sending a command to write to MDR0 along with the desired command bits.

\*/

SpiaRegs.SPIFFRX.bit.RXFFIL = 1; // generate int after 1 or more things received

GpioDataRegs.GPACLEAR.bit.GPIO9 = 1; // transmit to all chip

GpioDataRegs.GPACLEAR.bit.GPIO10 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO11 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO22 = 1;

SpiaRegs.SPITXBUF = ((unsigned)0x20)<<8; // CLR COUNT all four chips

while (SpiaRegs.SPIFFRX.bit.RXFFST != 1) {} // wait for until it

GpioDataRegs.GPASET.bit.GPIO9 = 1; // no transmit to all chips

GpioDataRegs.GPASET.bit.GPIO10 = 1;

GpioDataRegs.GPASET.bit.GPIO11 = 1;

GpioDataRegs.GPASET.bit.GPIO22 = 1;

SPIbyte1 = SpiaRegs.SPIRXBUF;

SpiaRegs.SPIFFRX.bit.RXFFIL = 2;

GpioDataRegs.GPACLEAR.bit.GPIO9 = 1; // trasmit

GpioDataRegs.GPACLEAR.bit.GPIO10 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO11 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO22 = 1;

SpiaRegs.SPITXBUF = ((unsigned)0x88)<<8; // WR to MDR0 (select MDR0)

// Filter clock division factor = 2

// x4 quadrature count mode (four counts per quadrature cycle)

SpiaRegs.SPITXBUF = ((unsigned)0x83)<<8;

while (SpiaRegs.SPIFFRX.bit.RXFFST != 2) {} //wait to receive ACK

GpioDataRegs.GPASET.bit.GPIO9 = 1; // no trasmit

GpioDataRegs.GPASET.bit.GPIO10 = 1;

GpioDataRegs.GPASET.bit.GPIO11 = 1;

GpioDataRegs.GPASET.bit.GPIO22 = 1;

SPIbyte1 = SpiaRegs.SPIRXBUF; // get ACKs

SPIbyte2 = SpiaRegs.SPIRXBUF; // clear buffer

/\*

The last part of init is writing to the MDR1 register (similar to how we wrote to MDR0), clearing interrupt signals, and enabling future interrupts. Basically, it sets the chip up to receive future data.

\*/

SpiaRegs.SPIFFRX.bit.RXFFIL = 2;

GpioDataRegs.GPACLEAR.bit.GPIO9 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO10 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO11 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO22 = 1;

SpiaRegs.SPITXBUF = ((unsigned)0x90)<<8; // WR MDR1

// 4 byte counter mode

// enable counting

// NOP

SpiaRegs.SPITXBUF = 0x00<<8;

while (SpiaRegs.SPIFFRX.bit.RXFFST != 2) {} // ACK

GpioDataRegs.GPASET.bit.GPIO9 = 1; // no transmit

GpioDataRegs.GPASET.bit.GPIO10 = 1;

GpioDataRegs.GPASET.bit.GPIO11 = 1;

GpioDataRegs.GPASET.bit.GPIO22 = 1;

SPIbyte1 = SpiaRegs.SPIRXBUF; // clear buffer

SPIbyte2 = SpiaRegs.SPIRXBUF;

SpiaRegs.SPICTL.bit.SPIINTENA = 1; // enable inerrupt

SpiaRegs.SPIFFRX.bit.RXFFOVFCLR = 1; // clear any indication of overflow

SpiaRegs.SPIFFRX.bit.RXFFINTCLR = 1; // clear any indication of messages received

SpiaRegs.SPIFFRX.bit.RXFFIENA = 1; // enable interrupts when rx buffer gets data

/\*\*\*\*\*\*\*\*\* SPI \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// SPI

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6; // Acknowledge interrupt to PIE

PieCtrlRegs.PIEIER6.bit.INTx1 = 1; //Enable PIE 6.1 interrupt

}

/\*

This code is using a state machine to select and read from the individual counters, one at a time. It sends the command to read from the OTR register (that holds the data), then sends 4 more bytes of garbage and receives 4 bytes of data.

\*/

void SPI\_RXint(void) {

GpioDataRegs.GPASET.bit.GPIO9 = 1; //no transmit

GpioDataRegs.GPASET.bit.GPIO10 = 1;

GpioDataRegs.GPASET.bit.GPIO11 = 1;

GpioDataRegs.GPASET.bit.GPIO22 = 1;

GpioDataRegs.GPASET.bit.GPIO19 = 1; // something for the DAC

// state machine

switch (SPIenc\_state) {

case 1:

// add delay

SPIbyte1 = SpiaRegs.SPIRXBUF;

SpiaRegs.SPIFFRX.bit.RXFFIL = 5;

SPIenc\_state = 2; // next state

GpioDataRegs.GPACLEAR.bit.GPIO9 = 1;

// Read from OTR, first encoder value

SpiaRegs.SPITXBUF = ((unsigned)0x68)<<8;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

break;

case 2:

SPIbyte1 = SpiaRegs.SPIRXBUF; // ACK

SPIbyte2 = SpiaRegs.SPIRXBUF & 0xFF; // the data

SPIbyte3 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte4 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte5 = SpiaRegs.SPIRXBUF & 0xFF;

SPIenc1\_reading = (SPIbyte2<<24) | (SPIbyte3<<16) | (SPIbyte4<<8) | SPIbyte5;

SpiaRegs.SPIFFRX.bit.RXFFIL = 5;

SPIenc\_state = 3;

GpioDataRegs.GPACLEAR.bit.GPIO10 = 1;

SpiaRegs.SPITXBUF = ((unsigned)0x68)<<8;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

break;

case 3:

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIbyte2 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte3 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte4 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte5 = SpiaRegs.SPIRXBUF & 0xFF;

SPIenc2\_reading = (SPIbyte2<<24) | (SPIbyte3<<16) | (SPIbyte4<<8) | SPIbyte5;

SpiaRegs.SPIFFRX.bit.RXFFIL = 5;

SPIenc\_state = 4;

GpioDataRegs.GPACLEAR.bit.GPIO11 = 1;

SpiaRegs.SPITXBUF = ((unsigned)0x68)<<8;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

break;

case 4:

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIbyte2 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte3 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte4 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte5 = SpiaRegs.SPIRXBUF & 0xFF;

SPIenc3\_reading = (SPIbyte2<<24) | (SPIbyte3<<16) | (SPIbyte4<<8) | SPIbyte5;

SpiaRegs.SPIFFRX.bit.RXFFIL = 5;

SPIenc\_state = 5;

GpioDataRegs.GPACLEAR.bit.GPIO22 = 1;

SpiaRegs.SPITXBUF = ((unsigned)0x68)<<8;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

SpiaRegs.SPITXBUF = 0;

break;

case 5:

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIbyte2 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte3 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte4 = SpiaRegs.SPIRXBUF & 0xFF;

SPIbyte5 = SpiaRegs.SPIRXBUF & 0xFF;

SPIenc4\_reading = (SPIbyte2<<24) | (SPIbyte3<<16) | (SPIbyte4<<8) | SPIbyte5;

/\*

At this point, the microcontroller has read the encoder values and is ready to give control to the control (pun unintended) function. The control function can use the newly read values.

\*/

SWI\_post(&SWI\_control);

break;

case 6: // NOOOOOOOOOOOOOOOOOOOOOOOO

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIenc\_state = 7;

// Output to DAC Ch2

SpiaRegs.SPIFFRX.bit.RXFFIL = 3;

SPIenc\_state = 7;

GpioDataRegs.GPACLEAR.bit.GPIO19 = 1;

SpiaRegs.SPIFFRX.bit.RXFFIL = 3;

SpiaRegs.SPITXBUF = ((unsigned)0x12)<<8;

SpiaRegs.SPITXBUF = (int)(dac2data << 4);

SpiaRegs.SPITXBUF = ((int)(dac2data))<<12;

break;

case 7:

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIbyte1 = SpiaRegs.SPIRXBUF;

SpiaRegs.SPICCR.bit.CLKPOLARITY = 0; // set for LS7366

SpiaRegs.SPICTL.bit.CLK\_PHASE = 1;

SPIenc\_state = 0;

// Debug to see how long 4 SPI enc read takes

// GpioDataRegs.GPACLEAR.bit.GPIO6 = 1;

break;

/\* this is for errors \*/

default:

SPIbyte1 = SpiaRegs.SPIRXBUF;

SPIbyte2 = SpiaRegs.SPIRXBUF; // these reads are not needed except to add some delay

SPIbyte3 = SpiaRegs.SPIRXBUF;

SPIbyte4 = SpiaRegs.SPIRXBUF;

SPIbyte5 = SpiaRegs.SPIRXBUF;

SPIenc\_state\_errors++;

break;

}

SpiaRegs.SPIFFRX.bit.RXFFOVFCLR=1; // Clear Overflow flag

SpiaRegs.SPIFFRX.bit.RXFFINTCLR=1; // Clear Interrupt flag

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6; // Acknowledge interrupt to PIE

}

/\*

Enables SPI communication by setting SS\ to low. Enables interrupt state machine to work.

\*/

void start\_SPI(void) {

SpiaRegs.SPIFFRX.bit.RXFFIL = 1;

GpioDataRegs.GPACLEAR.bit.GPIO9 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO10 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO11 = 1;

GpioDataRegs.GPACLEAR.bit.GPIO22 = 1;

// Allow CNTR to dump data to OTR

SpiaRegs.SPITXBUF = ((unsigned)0xE8)<<8; // Latch All ENCs

SPIenc\_state = 1;

}