ECE 3720

Microcomputer Interfacing Laboratory

Section 005

Aaron Bruner

Date Performed: 03 / 25 / 2021

Lab 9

**ABSTRACT**:

A lab designed to demonstrate how to program the PIC32 microcontroller SPI module and use a shift register to display 8 bits on LEDs. The microcontroller will be sending serial data to the shift register chip and then send that data to LEDs. However, the data will only change when a button is pressed. Like the previous labs, this will also enhance skills in reading documentations for new and existing parts.

**INTRODUCTION:**

The goal of lab 9 is to understand how to program our PIC32MX to display 8-bit data on LEDs using a shift register and SPI. A button will be what triggers the interrupt and results in the change of the LED output. In simpler terms, it will increase the global variable i which will select the different values in the spiChars array. We will be using the logic part of our WaveForms software to view the output instead of actual LEDs. To perform our lab, we required a variety of materials including:

* A breadboard
* Wires
* Push Button
* SN74HC595
* PIC32MX150F128D
* Analog Discovery 2 (AD2)
* Digilent WaveForms
* MPLAB Snap Debugger
* MPLAB X IDE (Programming Software)

The AD2 will power the PIC32 chip along with partially powering the MPLab Snap Debugger so that it can write to the PIC32. The AD2 will also be supplying 3.3V power to the SN74HC595 chip. The majority of the power for the MPLAB Snap Debugger will come from the Micro USB cord.

**EXPERIMENTAL PROCEDURES:**

We must assume that the individual reproducing this lab has already setup their breadboard in a manner that their PIC32 chip can be programmed using the MPLAB X IDE software.

To begin we analyze figure one and work out how the circuit is setup. The circuit shown is almost identical to what was used in the final attempt. However, we must notice that the shift register chip was not used and instead we used WaveForms built-in DIO to read the SPI data. This does not modify the result of our lab but instead makes it easier to read the data over zoom. If we were building this circuit in the lab, we would substitute the AD2 for the SN74HC595 chip and the result from the shift would be displayed on the LEDs we would setup. However, we opted to use the AD2 and so the way we setup our circuit is as shown. The serial clock output from our PIC32 went into DIO5. Then, the serial signal line from our PIC32 went into DIO6 and finally the serial data output went into DIO7. Lastly, the input for our button went into B7 on our circuit diagram which was used. Instead of a physical button it was substituted for a button supplied through the DIO of our AD2.

In our actual circuit we set serial signal to B15 and serial data output to B5, the serial clock was on B14, and instead of using DIO5-7 we used DIO0-2. As mentioned, this does not modify the result it just makes things easier to explain for this portion of the report.

Once we had the circuit setup we moved on to setup of our code. Some code was provided so those parts will not be mentioned. The first thing in our code is our global variable i which will be what value we are picking in the spiChars array. Every time the button is pressed we increase i by one and once it gets to 18 we start back at 0. The next part of code is our interrupt for the button. Every time the button is pressed we check what value i is and either set it to 0 or move on. Then, we take whatever value is at spiChars[i] and put it on the buffer to be sent out. We reset the interrupt flag and return to the program. The main function primarily sets up our SPI connection and then waits for the user to press the button. To start we set all B pins to output using ANSELB =0. Then we select pins B15, B14, and B5 to be used for our SPI connection. B15 will be used for SS1, B14 will be used SCK1, and B5 will be used for SDO1. By using PPS we set those pins accordingly and then set the above pins to be outputs. Pin B7 is the input for our button so we use TRISB and set that to input.

Now, we begin to setup our interrupt starting by setting the edge polarity to a catch on a falling edge. Then we set the priority level to 1, the interrupt flag to 0 (clear), and finally enable the interrupt module. Next is to setup the SPI we start by disabling the SPI interrupt with IEC1bits.SPI1EIE, IEC1bits.SPI1RXIE, and IEC1bits.SPI1TXIE all being set to 0. Then we disable SPI using SPI1CONbits.ON = 0. Next, we clear the receive buffer with SPI1BUF = 0. The next line disabled the enhanced buffer mode because we do not want that. Next, we are setting the clock frequency for the SPI module using SPI1BRG. This is a baud rate slow enough to observe the transmission of our data with the logic analyzer. SPI1STATbits.SPIROV clears the overflow detection module in SPI. SPI1CONbits.MSTEN and SPI1CONbits.MSSEN will set the SPI peripheral to be the master in our master slave architecture. Lastly, we set SPI1CONbits.CKP, CKE, ON; CKP means the clock is idle when we receive a low signal and active when there is a high signal. The CKE determines when the output changes; we want it to change when the clock transitions from active to idle so we set it to 1. Lastly, we set ON to 1 to enable the SPI module. The final while loop means we run this forever and let the interrupt control the code.

**RESULTS and DISCUSSION:**

Once the microcontroller was programmed it was apparent that things were working as expected when we pressed the push button we could see, through the logic part of WaveForms, the values being displayed. Below is an image of the output when you press the button rapidly:

A picture containing text, screenshot, white

Description automatically generated

One problem during this experiment was that the data was not transferring to the AD2 and so every time we pressed the button, we read 0 as our data since nothing was being transferred. After modifying many different variables and moving the pins around it finally began to work after putting the data line on B5. Once it was put on B5 we started to get output, and everything worked as expected. It may be worth mentioning that we went from using a physical button to using the digital IO built into the WaveForms software instead. This is not a very useful solution and there are not many ways to improve it since the root cause was not ultimately found.

This week we demonstrated how to utilize the SPI module, shift registers and WaveForms logic settings. Being able to read the datasheet and determine what pin everything needs to be connected to and how to properly setup your circuit is crucial in the real world. Transferring data through SPI and moving it over shift registers will be common in microcontroller programming careers.

**CONCLUSION:**

To conclude, lab 9 taught us many useful techniques on how to transfer data through SPI, manage that data with a shift register and combining those with interrupts on the PIC32 microcontroller. We also learned how to setup the logic in the WaveForms software to analyze SPI data. Also, as with many other labs, this one teaches us how to debug our circuit to check if everything is working. This lab also taught us more complex ways to analyze problems and how to solve them.

Diagram, schematic

Description automatically generated**FIGURES AND TABLES:**

**Figure 1: Wiring for lab 9 (Pin connections described in experimental procedures)**

**CODE:**

#include <xc.h>

#include <sys/attribs.h>

int i = 0;

char spiChars[18] = {0, 1, 2, 4, 8, 16, 32, 64, 128, 255, 254,

253, 251, 247, 239, 223, 191, 127};

void \_\_ISR(3)pushbutton(void){ // Interrupt for button press

if(i == 18) {

i = 0;

}

SPI1BUF = spiChars[i]; // Write data to the buffer

i++;

IFS0bits.INT0IF = 0;

}

int main(void) {

INTCONbits.MVEC = 1;

\_\_builtin\_enable\_interrupts();

CFGCONbits.JTAGEN = 0;

ANSELB = 0; // Digital IO

RPB15R = 0b0011; // PPS : B15 is SS1

RPB5R = 0b0011; // PPS : B5 is SDO1

TRISBbits.TRISB15 = 0; // Output for SS1

TRISBbits.TRISB14 = 0; // Output for SCK1

TRISBbits.TRISB5 = 0; // Output for SDO1

TRISBbits.TRISB7 = 1; // Button input

INTCONbits.INT0EP = 0; // Edge polarity (falling)

IPC0bits.INT0IP = 1; // Priority Level

IFS0bits.INT0IF = 0; // Interrupt Flag

IEC0bits.INT0IE = 1; // Enable

IEC1bits.SPI1EIE = 0; // Disable SPI interrupt

IEC1bits.SPI1RXIE = 0;

IEC1bits.SPI1TXIE = 0;

SPI1CONbits.ON = 0; // Disable SPI

SPI1BUF = 0; // Clear recieve buffer

SPI1CONbits.ENHBUF = 0; // Disable enhanced buffer mode

SPI1BRG = 1000; // SPI clock frequency

SPI1STATbits.SPIROV = 0; // Clear overflow detection

SPI1CONbits.MSTEN = 1; // SPI peripheral is the master

SPI1CONbits.MSSEN = 1; // Select to be in master mode

SPI1CONbits.CKP = 0; // Clock is idle when low, active when high

SPI1CONbits.CKE = 1; // The output data changes when clock transitions from active to // idle

SPI1CONbits.ON = 1; // Enable SPI

SPI1BUF = 0;

while(1){

}

}