ECE 341 - Section 52

Handshaking and LCD Control

Lab 6

Submitted By:
Tristan Denning

Contents

1 Introduction 2
2 Implementation 4
3 Testing and Verification 10
4 Conclusion 12
5 Questions 12

1 Introduction

The objective of this lab is to investigate the concepts behind handshaking and asynchronous parallel communications as well as their implementation in control systems. This project involves the communication of unsynchronized systems where the sender does not provide a clock signal to the receiver. For this project, asynchronous parallel communications will be implemented between the PIC32 and an LCD controller via the PMP module.

The PMP peripheral is designed to communicate with other parallel-data devices such as an LCD. The PMP may also be used to communicate with devices such as external memory devices, or another microcontroller where one controls the other in a master-slave configuration.

The specific LCD module used in this project includes an on-board controller chip. The LCD controller chip eliminates the need for software to control the LCD display. Without the controller, the embedded program would need to create each character by turning the associated pixels on or off. This would be very time-consuming for the programmer and may be prone to error.

One important concept of this project is handshaking. A handshake is a communication protocol between two systems operating at different clock frequencies. It is comprised of an exchange of signals between sender and receiver that dictate when data is transferred. This project utilizes communications classified as half duplex. Half duplex means both devices can send and receive data, but not at the same time. The following diagram illustrates the signals exchanged between the PIC32 acting as a master and the LCD module as the slave.

PIC32 - LCD Interface.

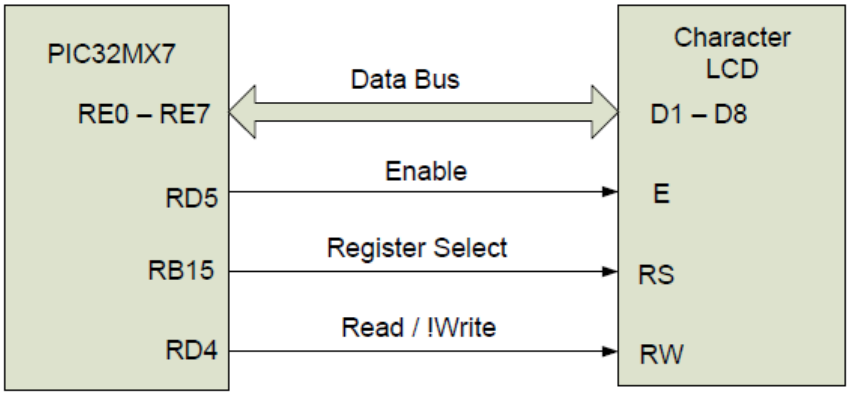


Figure 8 in "Handshaking and LCD Control with the Cerebot MX7cK™" Richard Wall.

Handshake protocols can be met by bit-banging or by use of the Parallel Master

Port peripheral on the PIC32. Bit-banging entails setting specific pins high or low at the appropriate time to meet handshake specifications. This method is tedious for the programmer, but may be necessary if no hardware is available as a substitute. The preferred method of meeting handshake protocols is to utilize existing hardware like the PMP. For this lab, the PMP communicates to the LCD module through 3 control lines and an 8-bit data bus shown in **Figure 8**. The RS signal selects which register data is being written to or read from. RS will be a 0 for the instruction register, or a 1 for the data register. The RW line will either be a 0 for a write operation or a 1 for a read operation. The Enable line is a read/write strobe which is high for read operation, then the falling edge writes data.

The LCD on the LCD controller module can show 16 characters on 2 lines. Each character is stored at an address in the controller's RAM. At each address, 8-bits store the character code. An address counter in the LCD controller keeps track of where the cursor is (what location in memory is present.) The LCD controller can store more than 16x2 characters in memory for scrolling. These characters do not appear on the screen unless proper instructions are given to scroll. Scrolling is beyond the scope of this project.

2 Implementation

The following data and control flow diagrams illustrate the structure of the embedded program:

Top-Level CFD

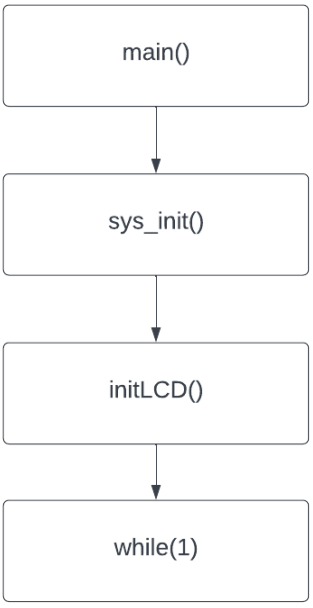


Figure 1

As shown in *Figure 1*, the first function call in main is system_init(). As in previous projects, this function runs the Cerebot board's setup function. The next step is to initialize the PMP and LCD module. This is done within the initLCD() function that resides in the LCDLibrary.c file created for this lab.

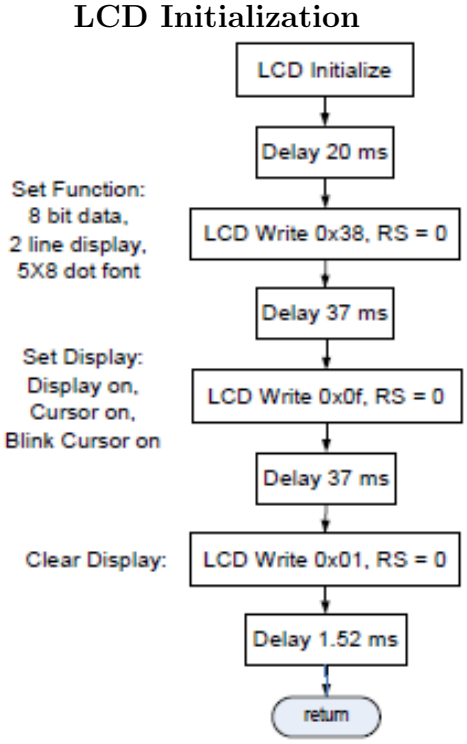


Figure 14. Control flow diagram for the LCD initialization[1]

From "Handshaking and LCD Control with the Cerebot MX7cK™" Richard Wall.

Within the initLCD() function, delays of 50 and 5 ms were used instead of 20, 37 and 1.52. This was a recommendation in the handout to ensure the LCD had time to process an instruction before proceeding to the subsequent instruction. The 50 and 5 ms delays were implemented with the same simple software delay developed in previous projects. The following code implements the LCD initialization:

Listing 1 - PMP and LCD initialization

```
void initLCD(void){
int cfg1 = PMP_ON|PMP_READ_WRITE_EN|PMP_READ_POL_HI|
   PMP_WRITE_POL_HI;
int cfg2 = PMP_DATA_BUS_8 | PMP_MODE_MASTER1 |PMP_WAIT_BEG_1 |
   PMP_WAIT_MID_2 | PMP_WAIT_END_1;
int cfg3 = PMP_PEN_0; // only PMA0 enabled
int cfg4 = PMP_INT_OFF; // no interrupts used

mPMPOpen(cfg1, cfg2, cfg3, cfg4);
    /*Register Select = 0 for instruction reg.*/
    PMPSetAddress(0);
    /* Set Function: 8 bit data, 2 line display, 5X8 dot fontSet*/
    PMPMasterWrite(0x38);
```

```
12      /*Wait 50 ms for instruction to execute*/
13      LCD_delay(50);
14      /*Set Display: Display on, Cursor on, Blink Cursor on*/
15      PMPMasterWrite(0x0f);
16      /*Wait 50 ms for instruction to execute*/
17      LCD_delay(50);
18      /*Clear the LCD Display */
19      PMPMasterWrite(0x01);
20      /*Wait 5 ms for instruction to execute*/
21      LCD_delay(5);
22  }
```

Each of the hexadecimal values sent to the PMPMasterWrite() function can be found in Digilent's **PmodCLP Reference Manual** in the following diagram:

LCD Controls

Table 3. LCD Instructions and Codes

Instruction	Instruction bit assignments										Description
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
Clear Display	0	0	0	0	0	0	0	0	0	1	Clear display by writing a 20H to all DDRAM locations; set DDRAM address register to 00H; and return cursor to home.
Return Home	0	0	0	0	0	0	0	0	1	X	Return cursor to home (upper left corner), and set DDRAM address to 0H. DDRAM contents not changed.
Entry mode set	0	0	0	0	0	0	0	1	I/D	SH	I/D = '1' for right-moving cursor and address increment; SH = '1' for display shift (direction set by I/D bit).
Display ON/OFF control	0	0	0	0	0	0	1	D	C	B	Set display (D), cursor (C), and blinking cursor (B) on or off.
Cursor or Display shift	0	0	0	0	0	1	S/C	R/L	X	X	S/C = '0' to shift cursor right or left, '1' to shift entire display right or left (R/L = '1' for right).
Function Set	0	0	0	0	1	DL	N	F	X	X	Set interface data length (DL = '1' for 8 bit), number of display lines (N = '1' for 2 lines), display font (F = '0' for 5x 8 dots)
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address counter AC5 – AC0
Set DDRAM address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address counter AC6 – AC0
Read busy flag/address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Read busy flag (BF) and address counter AC6 – AC0
Write data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into DDRAM or CGRAM, depending on which address was last set
Read data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from DDRAM or CGRAM, depending on which address was last set

Figure 2

The next important functions defined in LCDLibrary.c are the writeLCD() and readLCD() functions provided in Richard Wall's Project 6 handout. In this project, the read function is only ever provided a 0. The PMPSetAddress() takes the 0 and selects the instruction register to be read from. Then the current address on the instruction register is read, which indicates the current cursor position on the LCD.

The writeLCD() function takes a 1 as well as a character variable. First, the busyLCD() function is called to wait for the LCD to finish processing any current operations. Then the PMPSetAddress() takes the 1 and selects the data register to be written to. Finally the character, c, is written at the current location of the cursor with the PMPMasterWrite() function. See the listings below:

Listing 2 - LCD Read Function

```
char readLCD(int addr){
    PMPSetAddress(addr);
    mPMPMasterReadByte();
    return mPMPMasterReadByte(); //returns cursor location
```

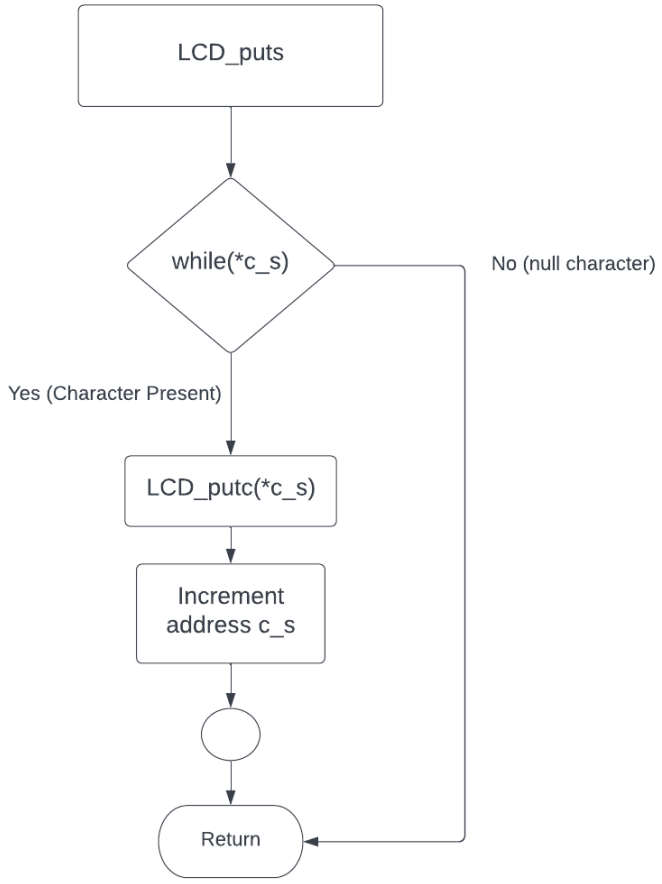
Listing 3 - LCD Write Function

```
void writeLCD(int addr, char c){
    while(busyLCD());   //Pause until PMP is not busy
    PMPSetAddress(addr);
    PMPMasterWrite(c);
```

The next two functions in LCDLibrary.c are the LCD_puts() and LCD_putc() functions. The LCD_puts() function takes a string of characters provided in main, and creates a pointer variable *c_s that points to the address of the first character in the string. When a string array is created in C, the compiler appends a null character to the end of the string. In C, anything other than 0 is logically interpreted as a one. This includes characters and spaces within a string. Within LCD_puts(), a while loop increments over each of the characters in the string, sending each one to the LCD_putc() function until the null character is reached. At each stage, the next element of the string array is obtained by incrementing the address of c_s. See CFD for LCD_putc and listing 4:

Listing 4 - LCD_puts()

CFD for LCD_puts




```
void LCD_puts(char *c_s){
    while(*c_s){
        LCD_putc(*c_s);
        c_s ++;
    }
```

The LCD_putc() function controls where on the LCD screen a character is written. The default case ensures that no characters are printed off-screen. A variable s is created to store the current cursor location on the LCD. When the cursor increments to an off-screen position, the LCD_putc function changes the cursor position to the next on-screen position before writing the next character. The cursor position is changed by writing the desired position's address bitwise-OR'd with the busy flag. The function also checks if the current character is the \n or \r character commands. When the current character is read to be \n, the cursor is moved to next line. If

the \r character command is received, the cursor is moved to the beginning of the current line. The following CFD provided in the lab handout was implemented into the LCD_putc function in listing 5:

CFD for LCD_putc

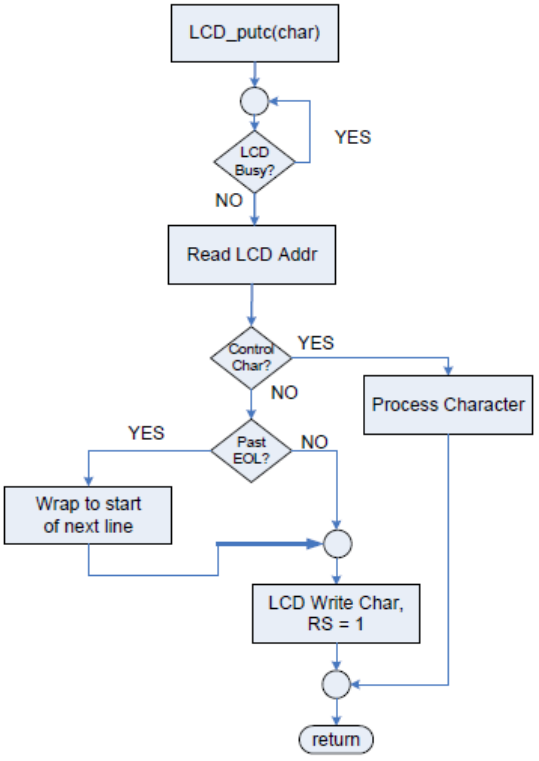


Figure 15. LCD display character control flow diagram – LCD_putc()

Figure 15 in "Handshaking and LCD Control with the Cerebot MX7cK™" Richard Wall.

Listing 5 - LCD_putc()

```
void LCD_putc(char c){
    int s;
    while(busyLCD());
    //s = readLCD(0);
    switch(c)
    {
        case '\n':      // next line
            if(s >= 0x00 && s <= 0x0F)
                writeLCD(0, (0x40 | BIT_7));
            else
                writeLCD(0, (0x00 | BIT_7));
            break;
        case '\r':      // reset to start of line
```

```
14              if(s >= 0x40 && s <= 0x4F)
15                  writeLCD(0, (0x00 | BIT_7));
16              else
17                  writeLCD(0, (0x40 | BIT_7));
18              break;
19          default:        // process character
20              s = readLCD(0) & 0x7F;
21
22              if(s > 0x0F && s < 0x40)
23                  writeLCD(0, (BIT_7 | 0x40));
24              if( s > 0x4F )
25                  writeLCD(0, (BIT_7 | 0x00));
26              writeLCD(1,c);
27              break;
28      }
29  }
```

3 Testing and Verification

After the code was compiled successfully, it was tested by writing two strings to the LCD that would switch every 5 seconds:

Listing 6 - Testing Code

```
1       char string1[] = "Does Dr J prefer PIC32 or FPGA??";
2       char string2[] = "Answer: \116\145\151\164\150\145\162\041";
3
4       while(1){
5         // writeLCD(1, 'T');
6
7          clrLCD();
8          LCD_puts(string1);
9          LCD_delay(5000); // 5 Second wait
10
11         clrLCD();
12         LCD_puts(string2);
13         LCD_delay(5000); // 5 Second wait
14
15      }
```

The LCD screen showed first the question: “Does Dr J prefer PIC32 or FPGA??” Then the LCD printed the answer: ”Neither!” indicating that the LCD control program was working correctly.

The next step was to verify handshake timing requirements by continuously writing a character to the LCD using the writeLCD() function. For this step, lines 7-13 of

Listing 6 were commented out, and line 5 was uncommented. The LCD screen filled up with T's. With oscilloscope connections to measure the Read/Write, Enable and Data lines, the following waveform and timing measurements were obtained:

Handshake Signal Timing Measurements

Parameter	Symbol	Min	Units	Test Pin	Measured [ns]	Reference
Enable Cycle Time	t_c	500	ns	Enable	900	Image 1
Enable High Pulse Width	t_w	220	ns	Enable	310	Image 2
RS, RW set up time	t_{su}	40	s	RS/RW	300	Image 3
RS, RW Hold time	T_H	10	ns	RS/RW	100	Image 4

Image 1

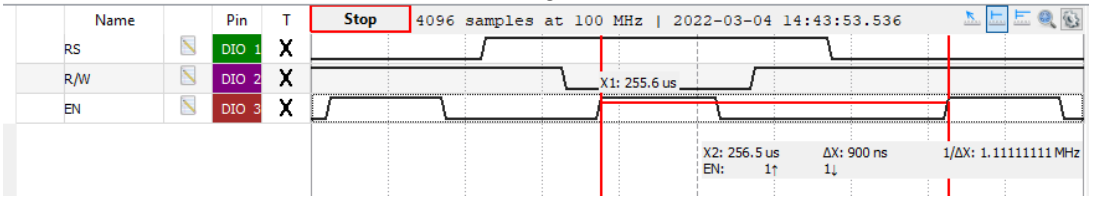


Image 2

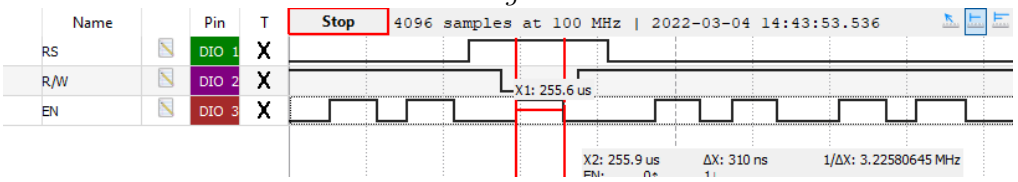


Image 3

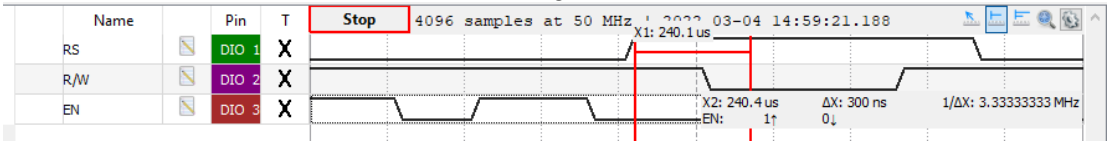
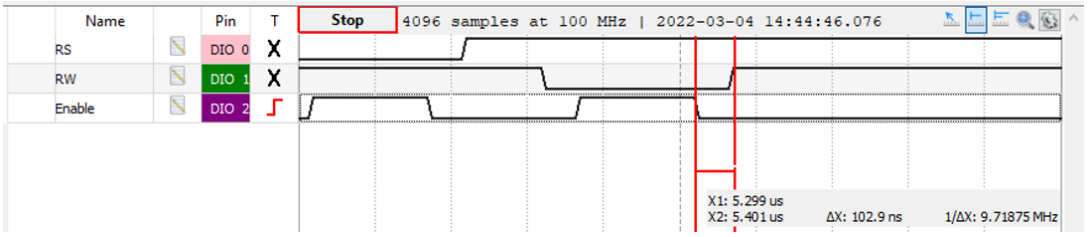


Image 4



Thus, each of the handshake minimum timing constraints were met.

4 Conclusion

The objective for this lab was accomplished and the LCD was successfully controlled by PIC32. This project utilized the concepts of handshaking and asynchronous parallel communications via the PMP. Each of the minimum timing constraints as defined in the PmodCLP reference manual were met with this implementation.

5 Questions

1. What is the maximum rate at which the LCD can receive characters? Using this rate, how much time would it take to completely erase and rewrite the visible display area?

   (a) The minimum time that the read or write operation can be repeated is set by the enable cycle time, t_c. According to Dr. Wall in the lab handout, the maximum rate the PmodCLP can be accessed is 2MHz (corresponding to the minimum t_c of 500 ns. So the maximum rate that characters can be written to the LCD is given by:

$$R_{Max}\frac{[Characters]}{[second]} = \frac{1}{t_c} = \frac{1}{500 * 10^{-9}}$$
$$= 2 * 10^6 \frac{[Characters]}{[second]}$$

   According to the Samsung data-sheet for the KS0066U LCD, clearing the display takes 1.53 ms. So the time to erase and rewrite the visible display area is calculated by:

$$T_{E+W} = 1.53 * 10^{-3}[s] + 16[characters] * 2[rows] * 500 * 10^{-9}[s]$$
$$= 1.546\ ms$$

2. Is using the PMP peripheral more efficient than bit-banging the LCD? Why or why not?

   (a) The PMP peripheral is more efficient than bit-banging because it can send 8-bits of data at a time without waiting on instructions from the program. It is also more efficient for the programmer, who saves time by letting hardware like the PMP take care of handshake protocols rather than writing each step and delay.