ECE341

**Lab6 - PMP and LCD Communication**

Report

**Seth Cram**

**10/19/2021**

**Introduction:**

Goal:

To investigate concepts involving information passing between unsynchronized systems with handshaking using the parallel master port (PMP), liquid crystal display (LCD) controller, and LCD display.

Background Information:

This lab will be our first communication protocol, with parallel and asynchronous data transfer. Parallel meaning that we can pass multiple bits in a single transfer. Asynchronous meaning that there’s no clock involved so data can arrive at any time. Since we’ll pass information between two systems operating at different frequencies and asynchronously, we’ll need to utilize a handshake. Without a handshake, we run the risk of a setup or hold time violation and going metastable as a result.

We’re using an LCD module since its screen can convey much more instrumentation information than a simple LED. This is our first off-chip peripheral since it’s external to the PIC32. It’s connected to the PIC32 through wires. The external LCD module runs at around 270K Hz, with the PMP operating at 10M Hz. So, a handshake will be used to synchronize the different clock domains.

The PMP is for data transfers involving external peripherals. It operates as a slave to the PIC32 processor. For this lab, it also acts as the master of the LCD controller. The order of operations will be: our software running on the processor interacts with the PMP, then the PMP interacts with the LCD controller, and the LCD controller displays on the LCD screen. The signals that form the interface between the PIC32 and the LCD Controller are the bi-directional 8-bit data line, the 2-bit read/write and enable from PMP to LCD Controller, and the 1-bit address line, called the Register Select (RS), also from the PMP to the LCD Controller. The PMP also does the handshaking, every single data transfer, between the PIC32 and LCD module for us. The PMP can generate 16 address lines but we’ll only use a single one for this lab. Although not used in this lab, the PMP can also generate an interrupt.

The LCD screen is 16 characters on the top, locations 00-0F, and 16 on the next line, locations 40-4F. It has more display locations, but we won’t be using them for this lab. For this lab, we’ll configure it for ‘auto-incrementing’, so every time we write a character the address counter will advance to the next position. This means we’ll have to read the address counter to see if we’re at the end of a line yet, every character write.

Important components of the LCD controller include the Data Register (DR), Instruction Register (IR), address counter, display data RAM (DDR), and the busy flag. From the PMP, an RS value of zero corresponds to using IR, an RS of one corresponds to using the DR. The busy flag represents if the LCD controller is currently busy and can’t support a new operation. We can read the busy flag by polling the IR, but we also get the address counter, so we’ll have to use bit-masks to only retrieve the msb, which is the busy flag. In order to display on the LCD screen, we have to write to the address counter to move the cursor, then write data to the DDR. We’ll have to verify we’re displaying within the current range, since we won’t use scrolling to look at all the memory locations of the LCD screen.

This lab is also our first introduction to C strings. In C, a string is just an array of characters terminated by the null-termination character. They can be referred to as an array of characters or a pointer to characters.

*– Give a brief overview of the PMP peripheral. What sort of devices is the PMP peripheral intended for? Give a couple specific examples (other than an LCD).*

The PMP peripheral is intended for use with external devices capable of asynchronous, parallel data transfers. For example, external memory devices, other microcontrollers, and communications peripherals.

*– Why do we interface with the LCD via a controller chip? What purpose does the controller chip serve?*

We interface with the LCD via a controller chip, in order to utilize the LCD Controller and its data or instruction register. The controller chip, aka the LCD Controller, communicates information from/to the PMP, to/from the LCD Screen, acting as a middle man.

*The background information came from the lecture notes and “Lab5” handout.*

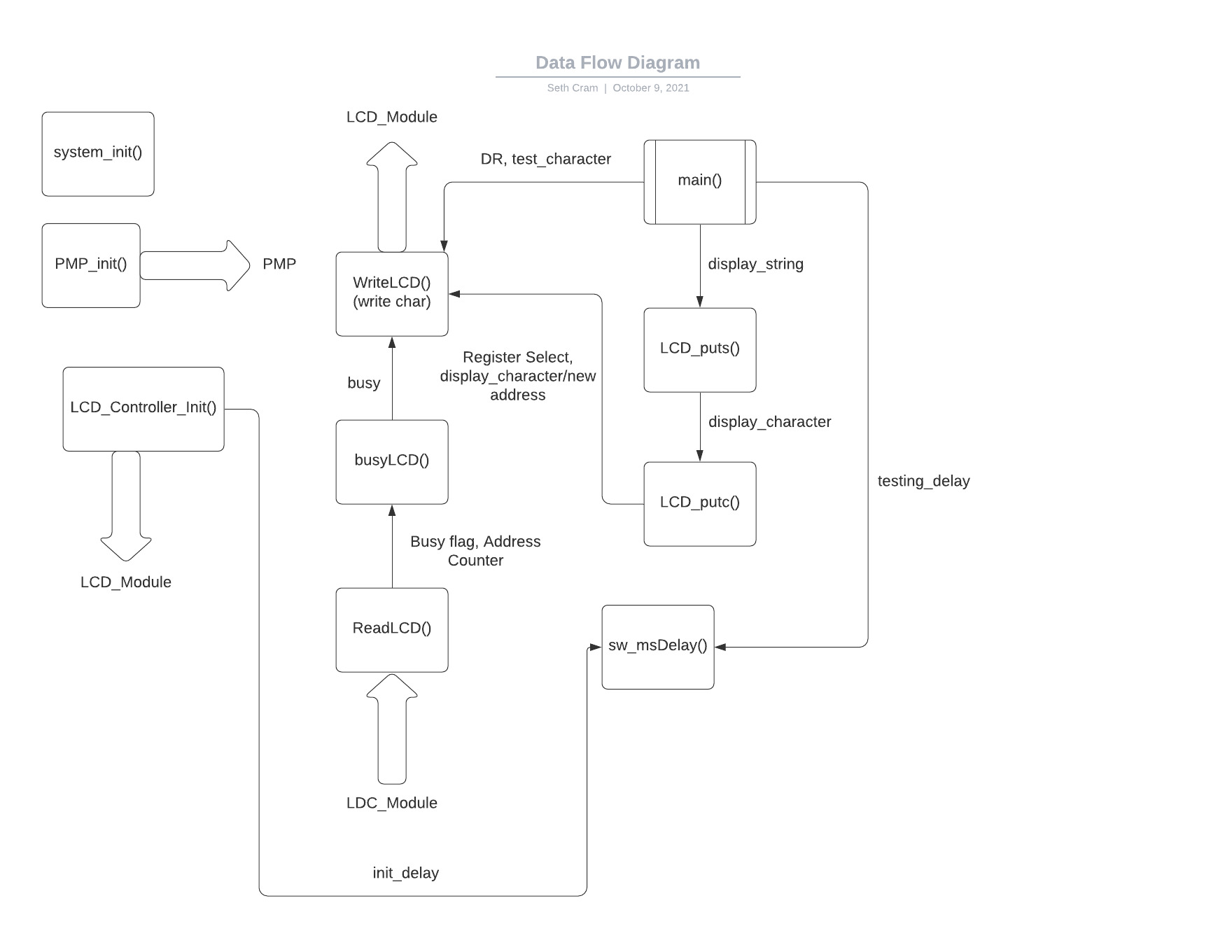
Plan:

First, I’ll put the PMP initialization code from the lab 5 handout, listing 2, at the bottom of my system\_init(). Next, we’ll implement ‘writeLCD()’ from listing 3 and ‘readLCD()’ from listing 4, both from the lab handout. But, we won’t call these functions yet. To use ‘writeLCD()’, we need to write the ‘busyLCD()’ function. This function will ‘readLCD()’ for the IR, then use a bitmask to get the busy flag, which is the msb, and return it.

Finally, I’ll write the LCD initialization function and call it right below initializing the PMP. Deriving this from figure 14 in the lab handout, we’ll delay for 50ms, write 0x38 to the IR, delay for 50ms, write 0x0f to the IR, delay for 50ms, write 0x01 to the IR, and finally delay for 5ms. We can’t poll the busy flag during our initialization due to design constraints put in place, so instead we delay for enough time that the LCD controller should be ready for another write operation. Next, I’ll get Listing 5 from the lab handout and implement ‘LCD\_puts()’ for testing purposes. But, in order to use the function, we’ll first need to write ‘LCD\_putc()’ to write a character to the LCD screen. First, we loop until the busy flag isn’t set, read from the LCD IR, bitmask away the busy flag msb, then make sure the cursor is within the screen. If it isn’t, write to the LCD to set it to the start of a line, then check for the control characters using a switch statement. If there are none, write the passed in character to the LCD screen.

**Implementation Discussion:**

Before implementation, I designed a data flow diagram to get a visual of what functions I’d need to design or modify.



As seen above, our data flow diagram is extremely interconnected, where a function likely relies on another “helper” function to perform its purpose. Almost all of these functions are new, so I cleared out almost every function used from the previous lab. The only one that was left to reuse was my delay function.

**Listing 1. LCDlib.h:**

First, I’ll start with the changes to my header file. From previous feedback provided by Dr. J, I planned on changing most of my macro constants into constant data types only defined within their needed function, but I ran out of time in the lab and had to settle on what I knew worked. I implemented new macros: for the instruction register, data register, start and end address of each line on the LCD screen, DDRAM control bit to change cursor address, and a variety of new function prototypes.

*//consts for data and instr register:*

*#define IR 0*

*#define DR 1*

*//LCD screen consts:*

*#define LINE\_1\_START 0x00*

*#define LINE\_1\_END 0x0F*

*#define LINE\_2\_START 0x40*

*#define LINE\_2\_END 0x4F*

*//const for changing the LCD address/cursor location:*

*#define DDRAM\_CNTRL\_BIT 0x80*

*/\* Function Prototypes \*/*

*void initPMP();*

*void initLCD();*

*void clearLCD();*

*int busyLCD();*

*void writeLCD(int addr, char c);*

*char readLCD(int addr);*

*void LCD\_puts(char \*char\_string);*

*void LCD\_putc( char currChar );*

*void LCD\_delay(unsigned int mS);*

**Listing 2. Initializations:**

Moving onto our initialization functions before ‘while(1)’, the first change I made was only including the Cerebot\_setup() function in system\_init(). The other system initializations already present weren’t needed, such as setting up the LEDs, buttons, and interrupts.

After initializing our system, I called ‘initLCD()’ to initialize the LCD. The first operation this function performed was called ‘initPMP()’, so we’ll look into initializing the PMP first. In initPMP(), we use a variety of bitmasks that were given to us in the lab handout to initialize the PMP for our handshaking and data transferring to the LCD Controller.

*int cfg1 = PMP\_ON|PMP\_READ\_WRITE\_EN | PMP\_READ\_POL\_HI | PMP\_WRITE\_POL\_HI;*

*//changed because of Dr. J recommendation:*

*int cfg2 = PMP\_DATA\_BUS\_8 | PMP\_MODE\_MASTER1 |*

*PMP\_WAIT\_BEG\_4 | PMP\_WAIT\_MID\_15 | PMP\_WAIT\_END\_4;*

*int cfg3 = PMP\_PEN\_0; // only PMA0 enabled*

*int cfg4 = PMP\_INT\_OFF; // no interrupts used*

*mPMPOpen(cfg1, cfg2, cfg3, cfg4);*

Now that we have the PMP initialized, we can move onto initializing the LCD Controller. Within this function, we can’t use the busy flag since the controller isn’t properly configured, so we delay for 50ms between write operations and initializing the PMP.

*LCD\_delay( 50 );*

*PMPSetAddress( IR );*

*//Set Function = 8 bit data, 2 line display, 5X8 dot font:*

*PMPMasterWrite( 0x38 );*

*LCD\_delay( 50 );*

*//Set Display = Display on, Cursor on, Blink Cursor on:*

*PMPMasterWrite( 0x0f );*

*LCD\_delay( 50 );*

*//Clear Display:*

*clearLCD();*

*LCD\_delay( 5 );*

The comments above describe what happens at each write operation and why they’re needed. We delay for 5ms at the end to satisfy hold timing constraints. Next, we’ll delve into the ‘clearLCD()’ function and explain why it’s a separate function.

**Listing 3. clearLCD():**

This function was implemented after my initial completion of the lab for demonstration purposes. I figured out that clearing the LCD screen in-between writing strings increased readability. This function provides useful controllability of the LCD screen.

*PMPSetAddress( IR ); //set to IR*

*//Clear Display:*

*PMPMasterWrite( 0x01 );*

**Listing 4. LCD\_puts():**

In addition to clearing the display every string write, writing a string essentially just outputs every character in the string until it reaches the null termination character, \0.

*//clear LCD before outputting whole string:*

*clearLCD();*

*while(\*char\_string) // Look for end of string NULL character*

*{*

*LCD\_putc(\*char\_string); // Write character to LCD*

*char\_string++; // Increment string pointer*

*}*

**Listing 5. LCD\_putc():**

To explain how the previous listing works, we’ll have to delve into how we output a character to the LCD screen. First, I loop until the LCD Controller is no longer busy, then I get the address of the screen by reading from the instruction register, then bit masking away the most significant bit.

*char readContents;*

*//int addyMask = 0111 1111;*

*int addyMask = 0x7F; //mask to get rid of msb*

*//loop until the LCD cntrller isnt busy:*

*while( busyLCD() );*

*//GET ADDY:*

*//read from the instruction register at curr address:*

*readContents = readLCD( IR ); //7 lsb's are addy of cursor*

*readContents = readContents & addyMask; //store only addy in 'readLCD'*

After that, I check if we’re past the end of a line, and if so reset the cursor to the next line through using macros previously displayed in the header file.

*//if curr addy tween last addy of 1st line and 1st addy of 2nd line:*

*if( (LINE\_1\_END < readContents) && (readContents < LINE\_2\_START) )*

*{*

*//set cursor to start of line 2:*

*writeLCD( IR, (LINE\_2\_START | DDRAM\_CNTRL\_BIT) );*

*}*

*//if curr addy past last addy of 2nd line:*

*if( LINE\_2\_END < readContents )*

*{*

*//set cursor to start of line 1:*

*writeLCD( IR, (LINE\_1\_START | DDRAM\_CNTRL\_BIT) );*

*}*

Finally, I use a switch statement with the passed in character as its argument to either handle a control character, or if there are none, output to the LCD Screen.

*switch( currChar ) //use switch for future expansion*

*{*

*case '\r': //move cursor to beginning of line*

*//if cursor on first line:*

*if( (LINE\_1\_START <= readContents) && (readContents < LINE\_2\_START) )*

*{*

*//set cursor to start of line 1:*

*writeLCD( IR, (LINE\_1\_START | DDRAM\_CNTRL\_BIT) );*

*}*

*//if cursor on 2nd line:*

*if( LINE\_2\_START <= readContents )*

*{*

*//set cursor to start of line 2:*

*writeLCD( IR, (LINE\_2\_START | DDRAM\_CNTRL\_BIT) );*

*}*

*break;*

*case '\n': //move cursor to start of next line*

*//if cursor on first line:*

*if( (LINE\_1\_START <= readContents) && (readContents < LINE\_2\_START) )*

*{*

*//set cursor to start of line 2:*

*writeLCD( IR, (LINE\_2\_START | DDRAM\_CNTRL\_BIT) );*

*}*

*//if cursor on 2nd line:*

*if( LINE\_2\_START <= readContents )*

*{*

*//set cursor to start of line 1:*

*writeLCD( IR, (LINE\_1\_START | DDRAM\_CNTRL\_BIT) );*

*}*

*break;*

*default: //normal char*

*//WRITE CHAR:*

*writeLCD( DR, currChar );*

*break;*

*}*

*}*

This is where I realized that you can’t do what I call ‘double-conditionals’ in C. For example:

*if( LINE\_1\_END < readContents < LINE\_2\_START ) //cant do in C*

vs.

*if( (LINE\_1\_END < readContents) && (readContents < LINE\_2\_START) ) //works*

The only time we use the data register seems to be when we’re writing a character to the LCD Screen, otherwise the instruction register is used for all other operations such as checking the busy flag and changing/reading the screen address.

**Listing 6. busyLCD():**

The above listing utilizes ‘busyLCD()’, ‘writeLCD()’, and ‘readLCD()’ so those are the next functions we’ll look into. Within busyLCD(), we bit-mask away the seven least significant bits of what’s read from the LCD Controller as the address and busy flag. That leaves us with just the busy flag in the most significant bit position. Now, we could directly return ‘readContents’, but I’m new to treating characters as mere ASCII integers, so I returned either 0 if the busy flag wasn’t set or 1 if it was.

*char readContents;*

*//int busyFlagMask = 0b1000 0000; //mask to just find the busy flag*

*// int bitmask?? w/ char? ye*

*int busyFlagMask = 0x80;*

*//read the IR to get the busy flag and addy cntr:*

*readContents = readLCD( IR );*

*//mask away the addy cntr bits:*

*readContents = readContents & busyFlagMask;*

*//if busy flag not set:*

*if( readContents == 0x00 )//== busyFlagMask )*

*{*

*return 0; //not busy*

*}*

*else*

*{*

*return 1; //busy*

*}*

Since we used readLCD() in this listing, we’ll look at its internals next. I initially configured this function incorrectly so that it always output a 0, for not busy, which resulted in a random assortment of characters being output to the LCD Screen. This was because of violating setup and hold timing requirements for the handshaking between the PMP and the LCD Controller, since the LCD Controller sometimes wasn’t ready yet when the PMP passed over new data or requested it.

**Listing 7. readLCD():**

This was another function provided to us by the lab handout. It uses peripheral library functions to read the LCD’s data or instruction register. We have to be sure that the busy flag is down before we use this function since it doesn’t check for it.

*char readLCD(int addr)*

*{*

*PMPSetAddress(addr); // Set LCD RS control*

*mPMPMasterReadByte(); // initiate dummy read sequence*

*return mPMPMasterReadByte();// read actual data*

*}*

This function utilizes a dummy read sequence first before reading the actual data to complete the handshake required for asynchronous communication syncing.

**Listing 8. writeLCD():**

WriteLCD() was also provided to us. This function checks if the busy flag is low before setting the register select line and writing the passed in character/new address to the LCD Controller.

*void writeLCD(int addr, char c)*

*{*

*while(busyLCD()); // Wait for LCD to be ready*

*PMPSetAddress(addr); // Set LCD RS control*

*PMPMasterWrite(c); // initiate write sequence*

*}*

**Listing 9. LCD\_delay():**

This final function listing is merely the pure software delay we used for the previous labs with a new name, so I don’t think displaying the internals is necessary.

**Listing 10. Testing:**

To test my code for the TA, I implemented the following in main() after initializing the system, PMP, and LCD:

*//test strings to write to LCD display:*

*char string1[] = "Does Dr J prefer PIC32 or FPGA??";*

*char string2[] = "Answer: \116\145\151\164\150\145\162\041";*

*while(1)*

*{*

*//alternate display of the test strings: (task 1)*

*LCD\_puts( string1 );*

*LCD\_delay( 5000 ); //delays 2s, should be 5s for testing*

*LCD\_puts( string2 );*

*LCD\_delay( 5000 ); //delays 2s, should be 5s for testing*

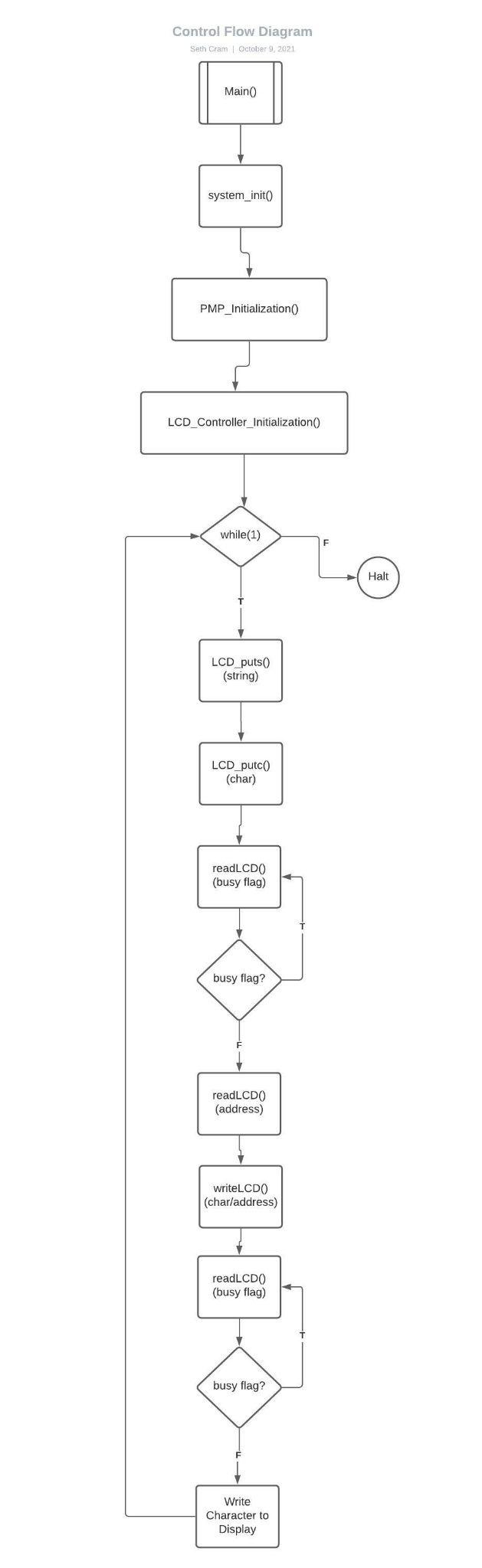
*//repeatedly write 'D' to LCD using "writeLCD()": (task 2)*

*//writeLCD( DR, string1[0] );*

*}*

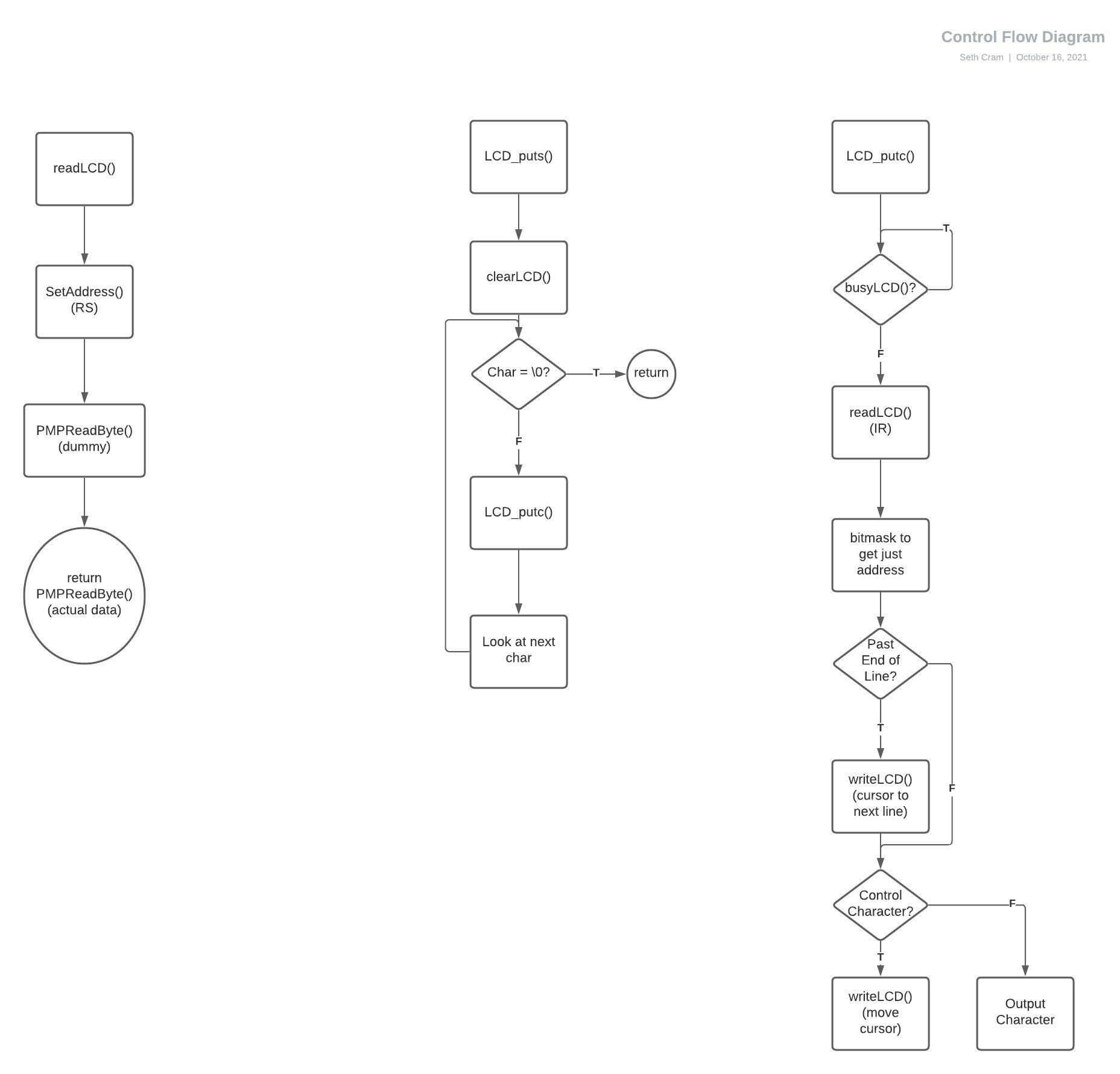
The program displayed the first string on the screen, then displayed the second after a 5 second interval, waited another 5 seconds and repeated. I then commented out everything in the while loop, and uncommented the ‘writeLCD()’ line for testing timing constraints.

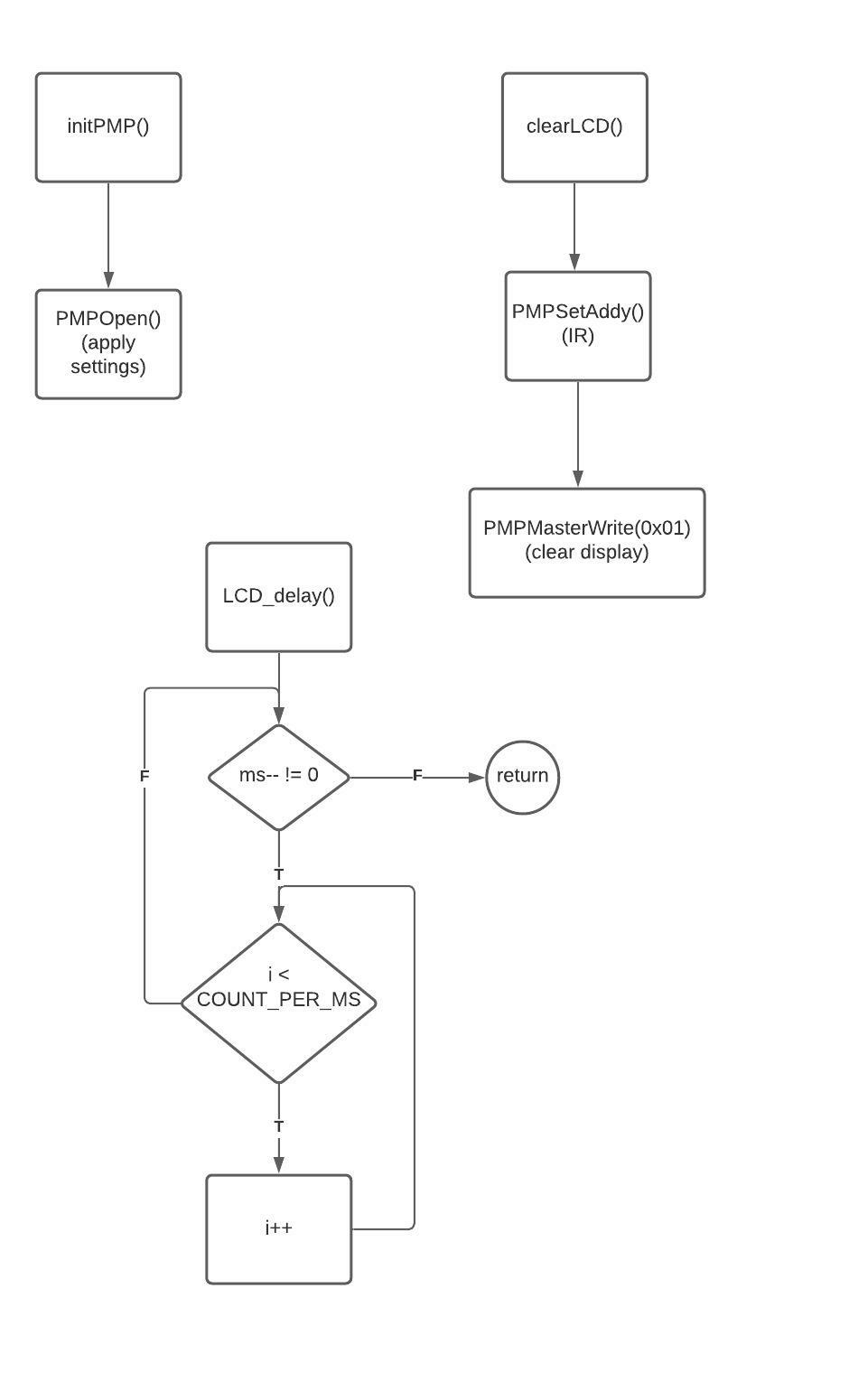
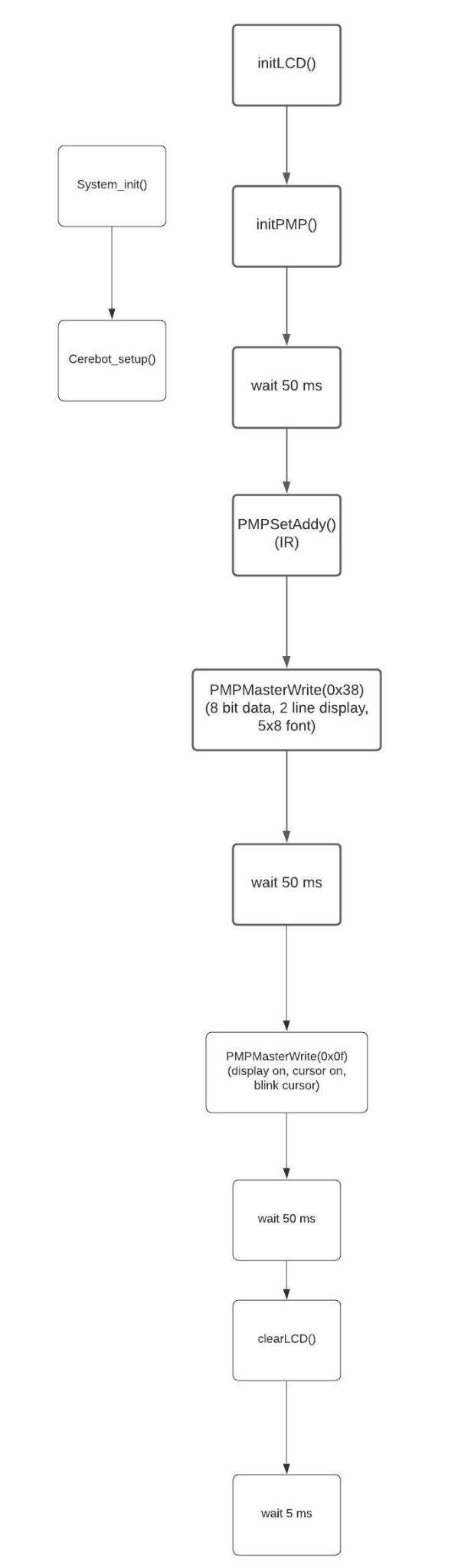
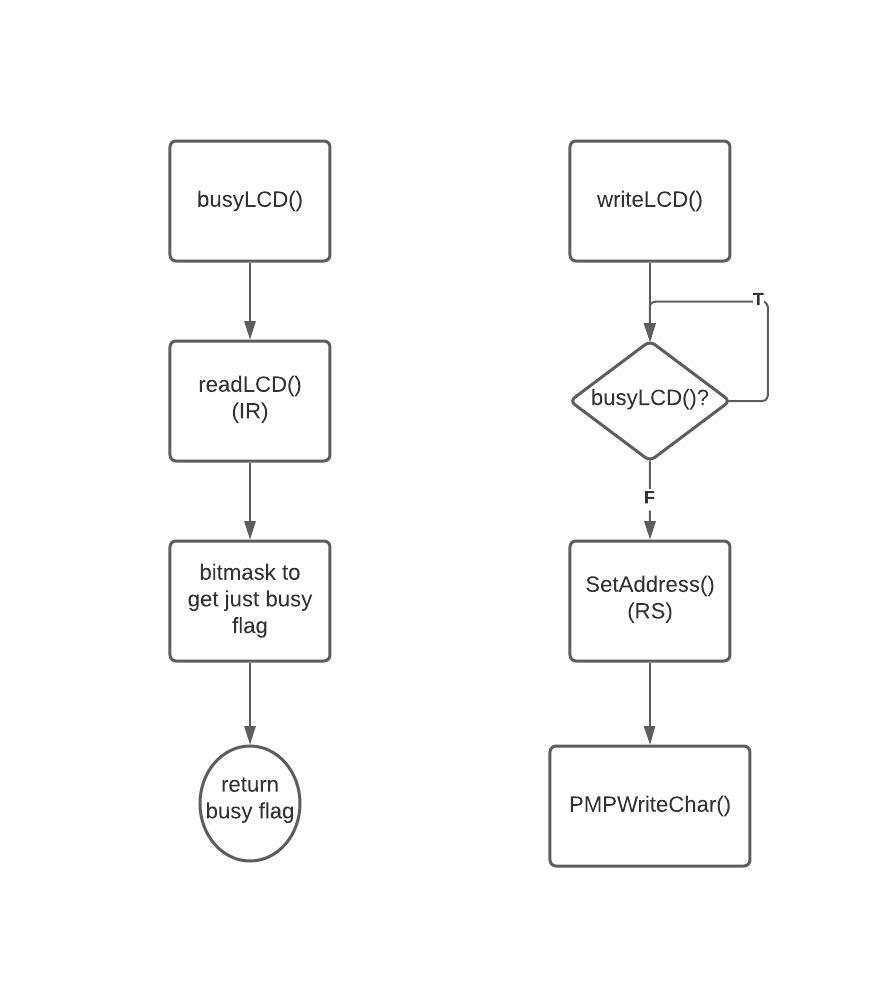
Finally, my control flow diagram models the behavior of the above specified listings:



As a result of using so many functions not called in main(), but rather called by other functions, the overall control flow diagram has a tough time of clearing illustrating a clear purpose of each function. So, it makes sense that Dr. J would want a CFD for each function.

*– Include CFDs for each function in LCDlib.c .*

**

**

*– Justify your choice for the number of wait states (beginning, middle and end of a transmission) for the handshaking signals.*

The handshaking signals were performed during readLCD(), seen with the single state ‘dummy’ read before reading and returning the actual data. We do this because the PMP is referred to as ‘pipelined’, so to access the desired data we have to first perform a read to clear data present from the previous operation, and then read to get our desired data. There is no checking of the busy flag during our readLCD() operation, so we have to make sure the busy flag isn’t high before calling this function.

Handshaking occurs every time a read or a write operation happens. Looking into writeLCD(), we can see that here we actually make sure the busy flag is down before writing. This is because we want to take extra care not to overwrite characters already present on the LCD Screen. We don’t need to do a dummy write because we aren’t retrieving information from the PMP, but rather, using it to send information for us.

**Testing and Validation:**

To test my project for the TA’s, I needed to write another function called ‘clearLCD()’. This function would clear the LCD between writing strings to the screen. Otherwise, the test strings would overlap and it wouldn’t be clear where one started and the other ended. Following the lab handout, I output the first test string, delayed for 5 seconds, then output the second test string and waited for 5 more seconds, then looped.

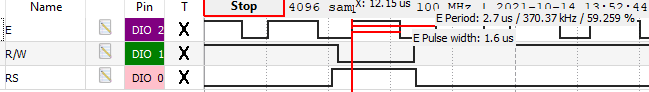
Asked in the lab handout, the LCD busy flag and cursor address is read twice because the PMP is pipelined. So, in order to get the current information and correctly perform a handshake, we need to implement a dummy ready before our second read, which returns the desired data.

LCD Bus Timing Results:

| **Parameter** | **Symbol** | **Min** | **Max** | **Units** | **Test Pin** | **Measured** |
| --- | --- | --- | --- | --- | --- | --- |
| **Enable Cycle Time** | t\_c | 500 | N/A | ns | Enable | 2700 |
| **Enable High Pulse Width** | t\_w | 220 | N/A | ns | Enable | 1600 |
| **RS, RW set up time** | t\_su | 40 | N/A | ns | RS / RW | 400 |
| **RS, RW hold time** | T\_h | 10 | N/A | ns | RS / RW | 400 |

Reference Waveforms:

|  |
| --- |
| T\_c: |



Since t\_c is the enable cycle time, which is equivalent to the enable period, T\_c = 2.7 us.

| t\_w: |
| --- |
|  |

Enable pulse width is 1.6 us.

| T\_su: |
| --- |
|  |

| t\_h: |
| --- |
| Both setup and hold time are 400 ns. We measure setup time from when both RS and R/W are stable, to when enable is high. We measure hold time from when Enable goes low, to when R/W or RS changes. |

The reason why the measured values are so much bigger than the minimums is because of the alterations we made to the initialization of the PMP, given to us by the lab-handout, at the behest of Dr. J. Specifically, this was changing the PMP setup from

*int cfg2 = PMP\_DATA\_BUS\_8 | PMP\_MODE\_MASTER1 | PMP\_WAIT\_BEG\_1 | PMP\_WAIT\_MID\_2 | PMP\_WAIT\_END\_1;*

Into

*int cfg2 = PMP\_DATA\_BUS\_8 | PMP\_MODE\_MASTER1 | PMP\_WAIT\_BEG\_4 | PMP\_WAIT\_MID\_15 | PMP\_WAIT\_END\_4;*

This change was made to satisfy minimum timing constraints for different workstation differences.

Each measurement doesn’t have a maximum because we could theoretically make them as long as we wished.

**Conclusion:**

In conclusion, we learned how to implement our first communication protocol during this lab. It was parallel and asynchronous, and utilized the Parallel Master Port in conjunction with the LCD module. It required handshaking because of the data transfer between two clock domains of wildly different operating frequencies, which was handled by the PMP. We constructed a library header and source file to take advantage of in the future. Our newfound ability to display messages on the LCD Screen greatly increases the testability for future projects, when compared to our initial testing of merely blinking LEDs.

Limitations to our design include the 32 characters we can write to the LCD Screen. Although writing more and implementing a scrolling feature is possible, we didn’t explore such avenues. Other limitations involve the timing constraints of the handshake performed during data transfers to and from the PMP and LCD Controller, discussed below.

*– 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?*

The minimum time that the read/write operation can be repeated is restricted by the enable cycle time: t\_c. So, the maximum rate at which the LCD can receive characters is 1/t\_c. Using the minimum t\_c, this would be 1/(500 ns) = 2 MHz, which is a character per 500 ns. Using the measured t\_c, this would be 1/( 2700 ns) = 370.37 KHz, which is a character per 2.7 us.

To completely erase the display, based on the method we performed during this lab: we’d need to write to the LCD Controller. So, erasing the display would take t\_c. Which is 500 ns using minimum delays From the delay found during our testing: 2.7 us.

We’ll need to add this clear display time to the time taken to rewrite the visible display area. The visible display area is 32 characters, so would ideally require 33 write operations, with the extra one used to move the cursor from the end of line 1 to the start of line 2. But, since we have to constantly check whether we’re at the end of the line or not, every character write would turn into first a reading of the address, and then a character write if deemed valid. Therefore, working under the assumption that our timing is correct and we don’t have to wait for the busy flag to lower, we need to perform 65 read/write operations to rewrite the visible display. So, 65 \* t\_c = time to write the entire visible display. For the ideal situation: 65 \* 500 ns = 32.5 us. Using our measurements: 65 \* 2700 ns = 175.5 us.

Adding erase display and write display times, the ideal situation would take 500 ns + 32.5 us = **33 us**. Using our implementation, it would take 2700 ns + 175.5 us = **178.2 us**.

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

Yes, the PMP peripheral is more efficient than bit-banging the LCD because less software is required for the LCD read and write operations, since extra hardware is being used for the PMP. With bit-banging, all the I/O operations use purely software, which is inherently slower than hardware.