COMPUTATION BOOK

NAME

Grant Abella

COURSE ECE 322 02 Lab Book B

TOPS™

1001 Rialto Road, Covington, TN 38019. Made in USA.

0 25932 35061 0

Table of Contents

Laboratory #2: External Interrupt Control of switched LEDs

Grant Abella
Brandon Largent

8/30/18

Laboratory Objective: Develop ~~an~~ external interrupt control interface techniques for switched control of the LEDs on the Atmega128A based STK128/64 starter kit.
Advisor: Dr. Gutschlag
Equipment used:

- Atmega 128A kit #27 EQ-2974
- Tektronics MSO 2014B oscilloscope EQ-3017
- Breadboard and wires
- 8-bit pin headers (x2)

*All implementation code as well as explanations are located on pages 10-12.

1. After connecting LEDs (L_i) to PORTA and push-button switches (K_i) to PORTB, we loaded the program from part 3 of Laboratory #1 into Atmel Studio and programmed the board. We tested that each key being pressed turned on each respective LED and found that everything functioned as intended.

2. In order to investigate the stability of switches K0 and K1, we routed the first 4 pins from PORTB to a breadboard using an 8-bit pin header so that we could split the signal for use with the oscilloscope probes. We did the same with the first 4 pins from PORTA because we wanted to see if the bouncing of the switches would affect the voltage readings on the LED headers as well. Unfortunately we were not able to capture a noticeable reaction on the LED port, so we decided to remove those traces from the reading for clarity. The captures from our oscilloscope are shown in the following two figures.

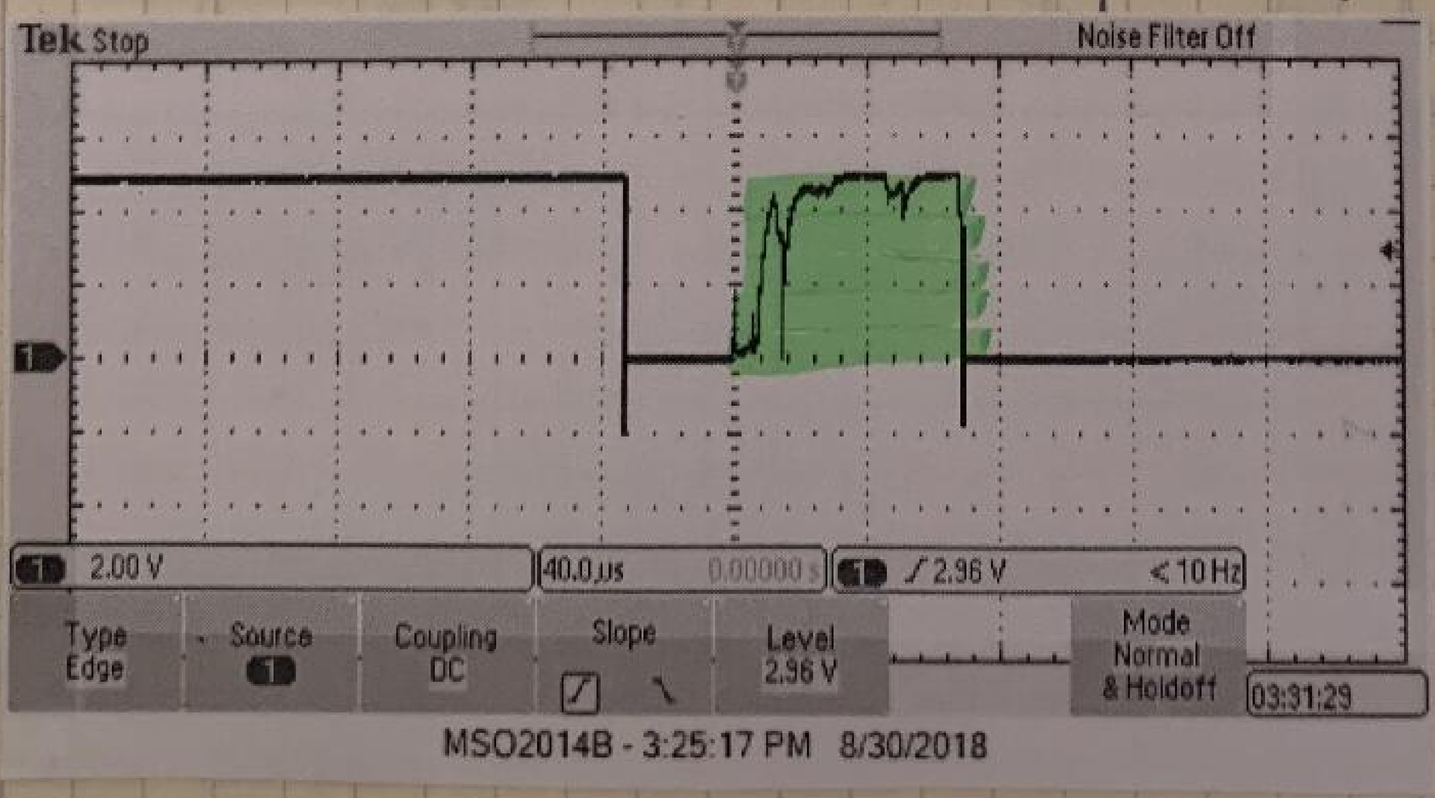


Figure 2-1: Voltage reading of switch K0 being pressed and exhibiting bounce.

The jagged and unstable portion of the signal (highlighted in green) is the result of the spring in key switch K0 bouncing after the key has been pressed.

8/30/18
3:32 PM

Laboratory #2: External Interrupt Control of Switched LEDs

Grant Abella
Brandon Lampert

8/30/18

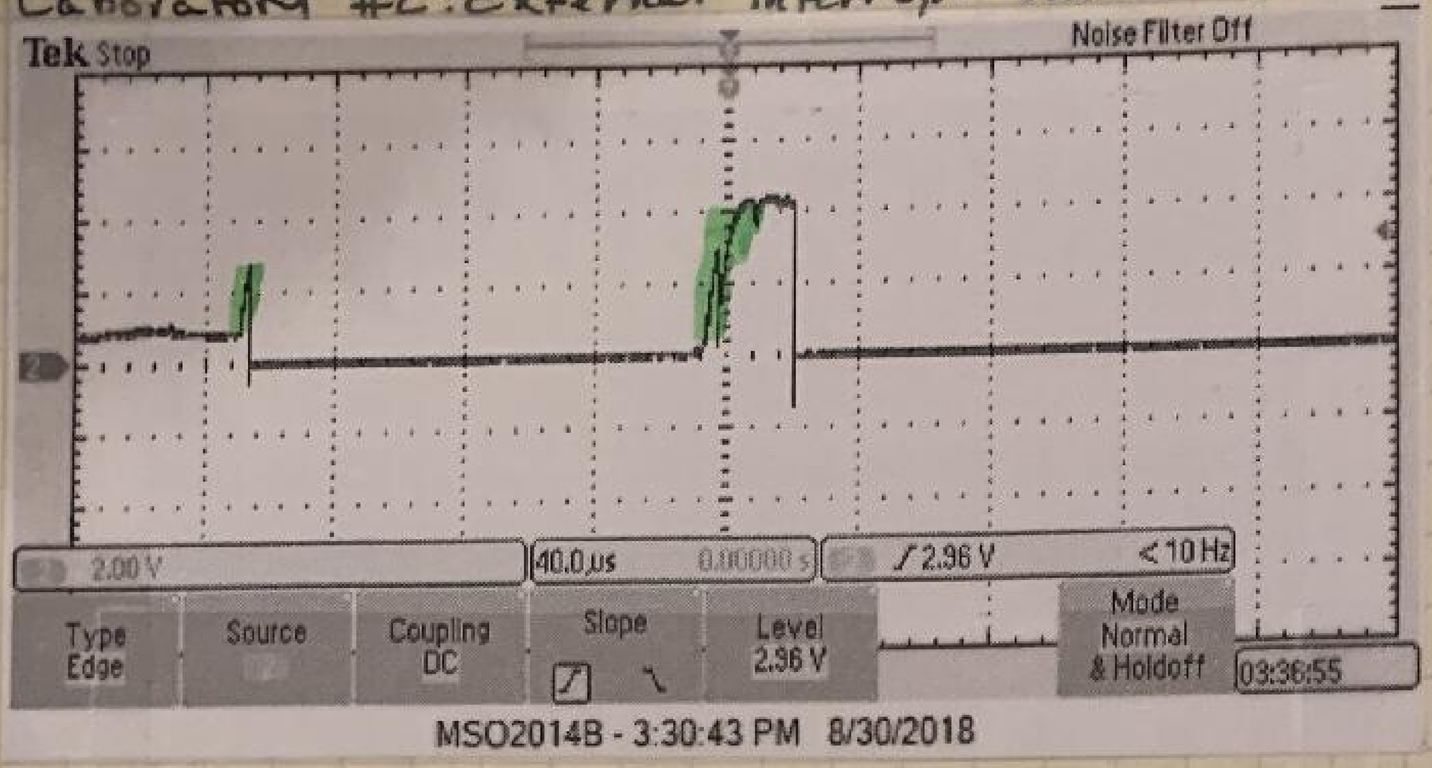


Figure 2-2: Voltage reading of switch K1 being pressed and exhibiting bounce.

Similarly, K1 displayed bounce as shown in figure 2-2 above. In the case of this laboratory, bouncing is not a huge issue as the LEDs still light up, but in future labs and in the field, bouncing is a serious issue that can cause many problems. This is because one button press could falsely be interpreted as multiple button presses and cause unintended or dangerous results.

3. We replaced the onboard switch K0 with a (momentary) push-button normally-open switch by connecting the positive end of the switch to pin 0 on PORTB, and the negative end to the ground pin next to PB0. This ground hookup is important in order for the circuit to be complete. In using the onboard ground, we ensured no floating voltage was present for the switch. We also split the positive end into two wires so that we could connect a probe from the oscilloscope to the switch in order to observe any bouncing.

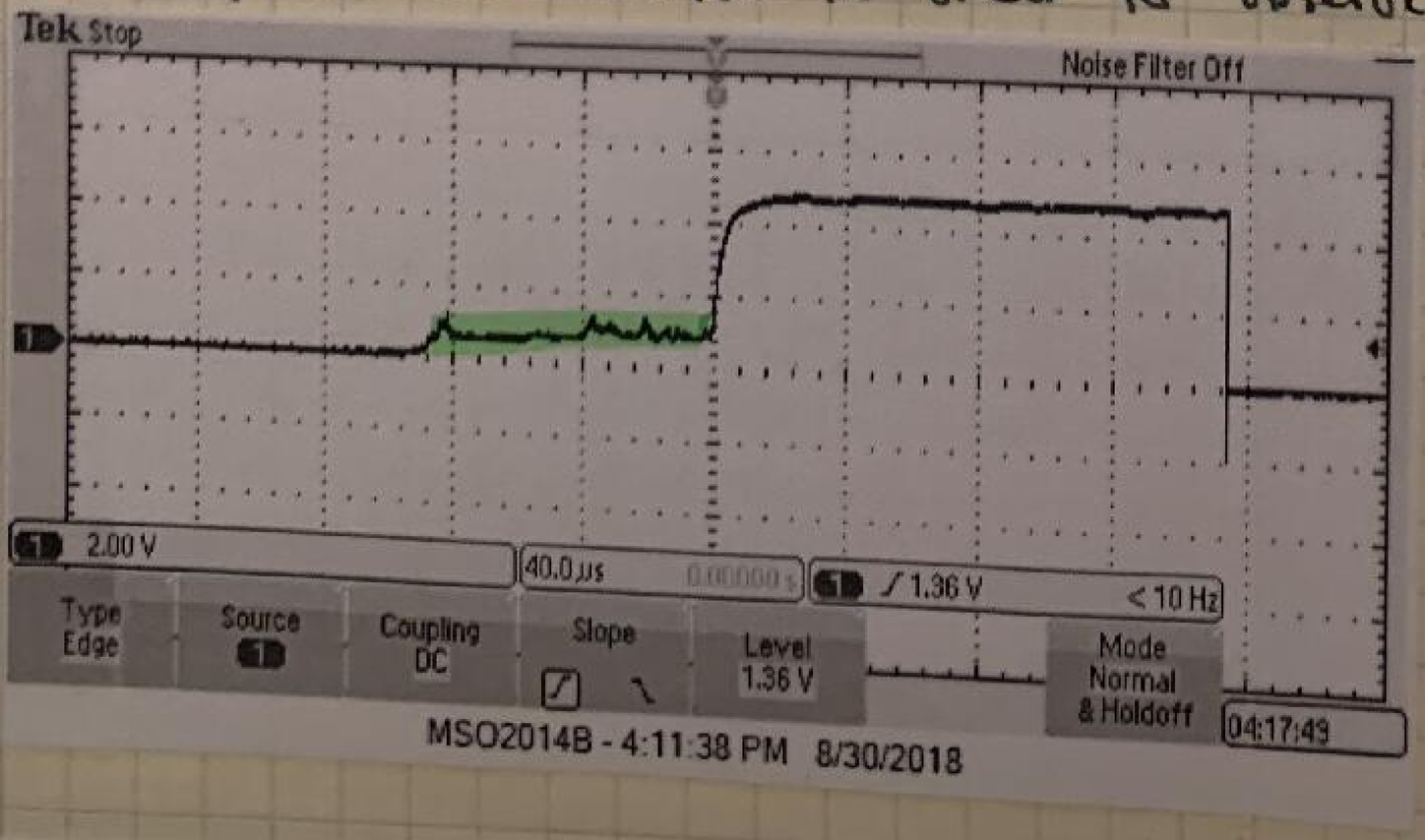


Figure 2-3: Voltage reading of momentary push-button switch displaying bounce.

8/30/18
4:15 PM

Laboratory #2: External Interrupt Control of Switched LEDs

Grant Abella
Brandon Lampert
8/30/18

The portion of the signal that is highlighted in green in Figure 2-3 is the section in which bounce is causing the voltage to rise. We noticed that the bouncing found with this external switch was much greater than what we found in the onboard switches K0 and K1. This could be a result of the external switch having more wear or being older than the onboard switches, or it could simply be that the onboard switches are of a higher quality than the external switch.

Ga
8/30/18
4:46 PM

4.a) In order to toggle LED LO every time key switch K0 is pressed using an interrupt, we will have to run a wire from pin 0 on PORTB to pin LO from the LED header, and a wire from pin 0 on PORTD to a breadboard with a 510 Ω resistor, to pin 0 of the key switch pin header.

By wiring like this, INTO (PD0) will be able to see a voltage change in pin K0 when ~~and~~ the key is pressed, and the LED will be able to be toggled using PORTB.

A flowchart describing the behavior of this program is shown below:

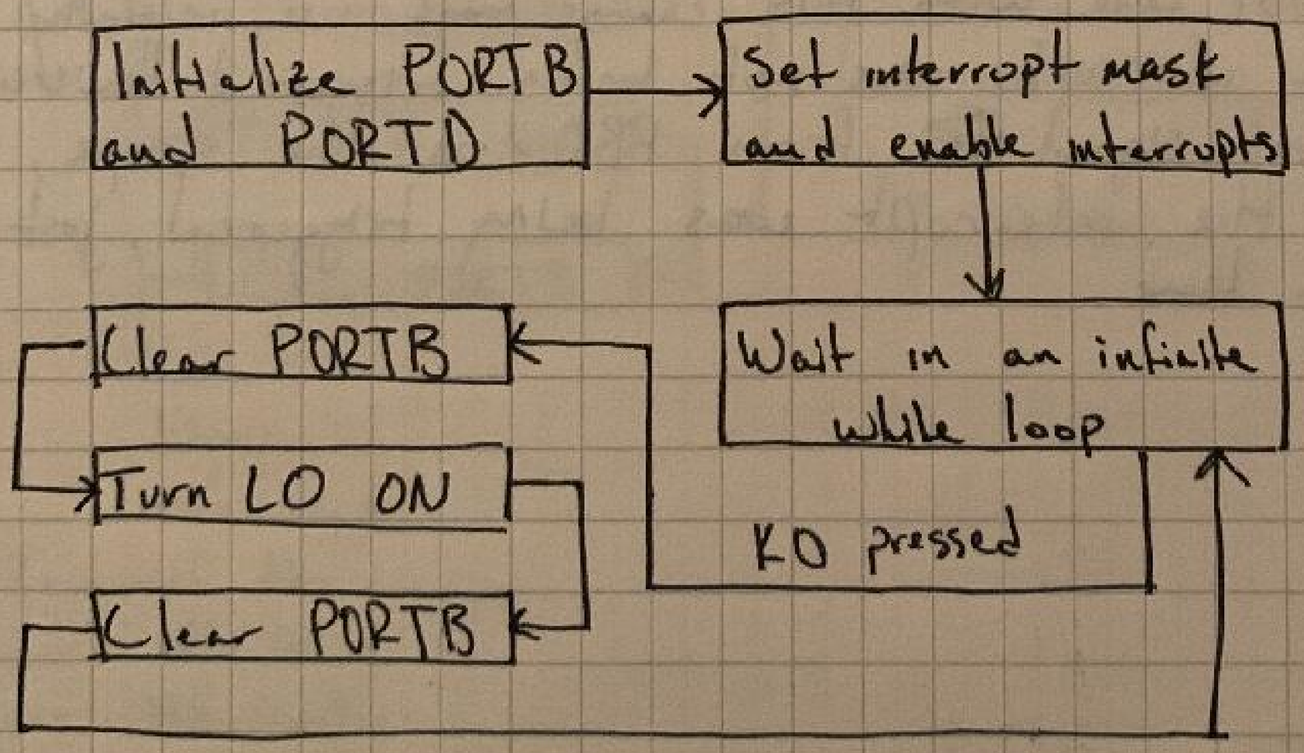


Figure 2-4: Flowchart for program that uses a single interrupt to toggle LED LO.

Ga
9/4/18
2:30 PM

We will need to be sure to both enable interrupts using the sei function, and set the external interrupt mask to enable INTO as a low-level trigger.

JF
9:46 PM

Laboratory #2: External Interrupt Control of Switched LEDs

Grant Abella
Brandon Lampert

9/4/18

When we implemented the code for this program onto the board, we found that the program initially did not function correctly. This was because we had omitted the line of code to enable INT0. Once we added this into the code, the program functioned perfectly.

All of the actual logic for LED toggling was performed in the ISR function for the interrupt, and the program waits in an infinite empty while loop until the interrupt is triggered by a keypress.

A low-level trigger is beneficial in cases where a trigger will depend on a signal that is naturally high but will switch to low. In this case, the key switches being high when not pressed meant that the LED was on when the switch was pressed. -2pts no circuit diagram

5.a) The only difference between this part and the previous part ~~#4~~ is that a negative edge trigger will be used. This is a simple matter of ~~the~~ changing the EICRA value to 0x02 (as found in the datasheet on page 88). The flowchart for this program is the same as the one shown in Figure 2-4. - Ding! Pgms

When we ran our code with this change made, we initially thought that the interrupt was not being triggered. However after viewing the signal ~~from~~ from PB0 on the scope, we found that the interrupt was being triggered, just for a very brief time.

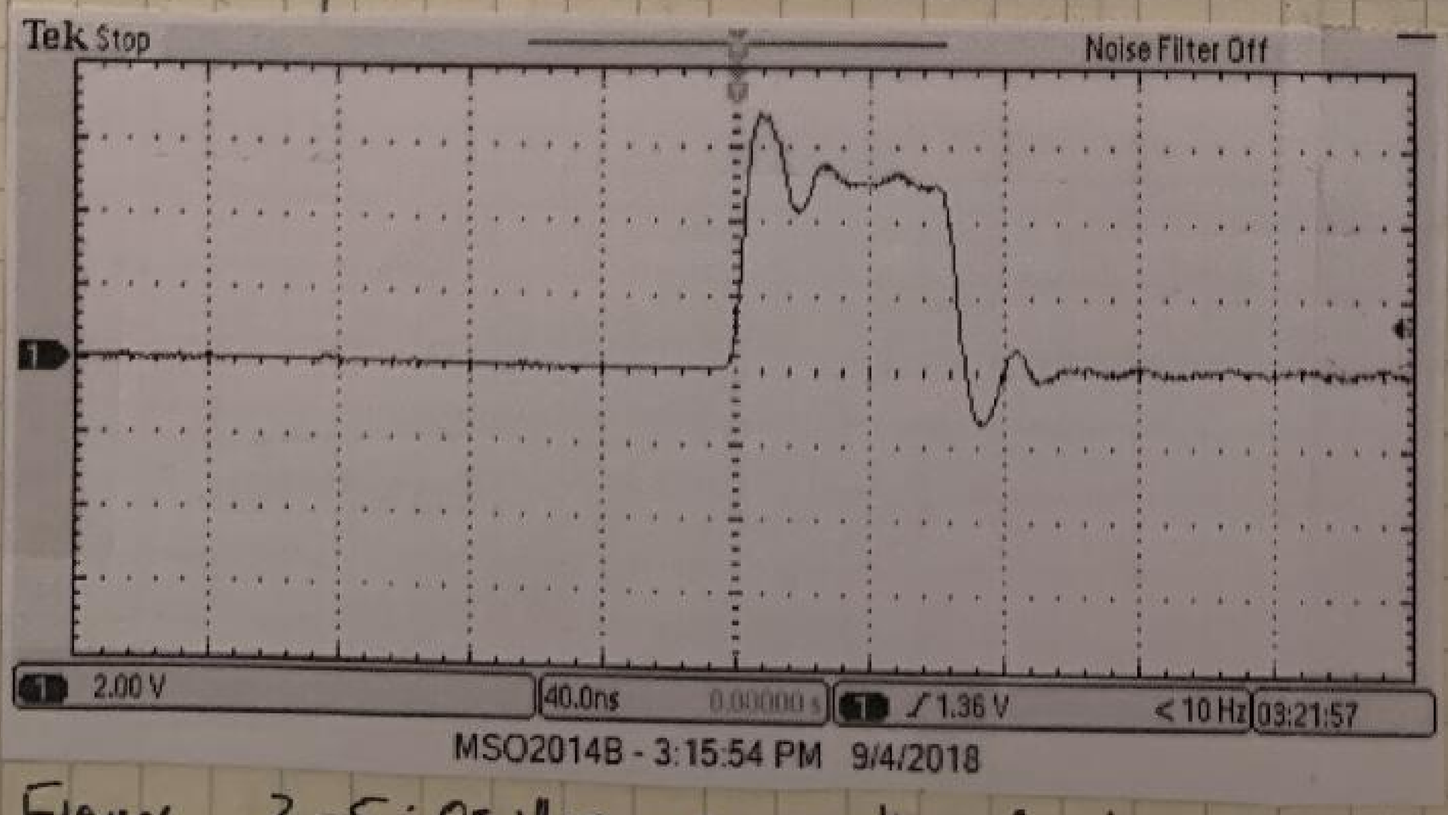


Figure 2-5: Oscilloscope reading of time spent in interrupt.

9/4/18
3:24 PM

Laboratory #2: External Interrupt Control of Switched LEDs

Grant Abella
Brandon Campbell

9/4/18

Due to the nature of the switches naturally being at a high level and INT0 being configured as a negative edge trigger, the time spent in the interrupt will naturally be a very short time. In this case, we had to set the time/division on the scope to 40 ns to even see the time spent in the ISR.

These edge triggered interrupts are useful in cases where you are looking in changes in high/low state and want to execute certain operation only for that short time that the state is being changed.

-2pts no circuit diagram

6. a) The functionality of a program that uses a negative-edge trigger to toggle LEDs L0-L3 will be essentially the same as the previous part, except using 4 interrupts and 4 LEDs.

9/4/18
4:46PM

Time Out
4:49PM
f.D.T.

The flowchart for this program will be similar to the ones from parts 4 and 5, however 4 interrupts and 4 LEDs will be utilized.

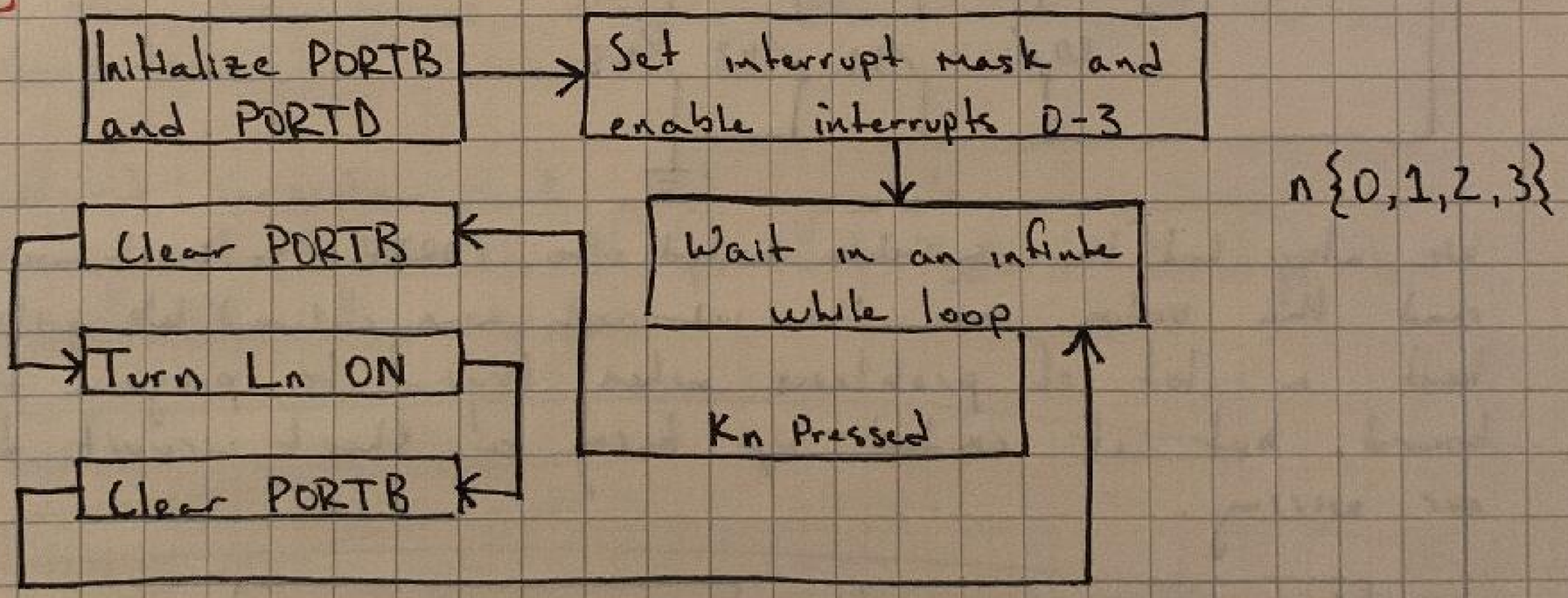


Figure 2-6: Flowchart describing behavior of a program that uses 3 key switches tied to interrupts to toggle LEDs.

b) Because the program must use a negative edge trigger, a similar difficulty will be encountered when testing if the program enters the interrupt. The LEDs will be toggled so quickly that we will not be able to visually see it happening. The scope will need to be used for this.

Also, as always, a 510 Ω resistor pack will be put in series with the key switches to protect the board pins.

9/6/18
2:30PM

Finally, the last problem that could arise is when two

Laboratory #2: External Interrupt Control of Switched LEDs

Grant Abella
Brandon Lampert
9/5/19

or more switches are pressed at one time. Program behavior is not defined for this event, so any results will be unexpected.

As expected, the program functioned as intended. We did need to attach the scope to the interrupt pins like in part 5 because of the short timing, but once we verified that each key switch triggered an interrupt we demonstrated the program to a TA and got signed off.

-2pts no circuit diagram

7.a) To achieve the same effect as in part 6 but using a single interrupt, the signals from switches K0-K3 will need to be channeled into a single signal attached to INT0. After Dr. 6.'s demonstration on diodes, we constructed a circuit like so:

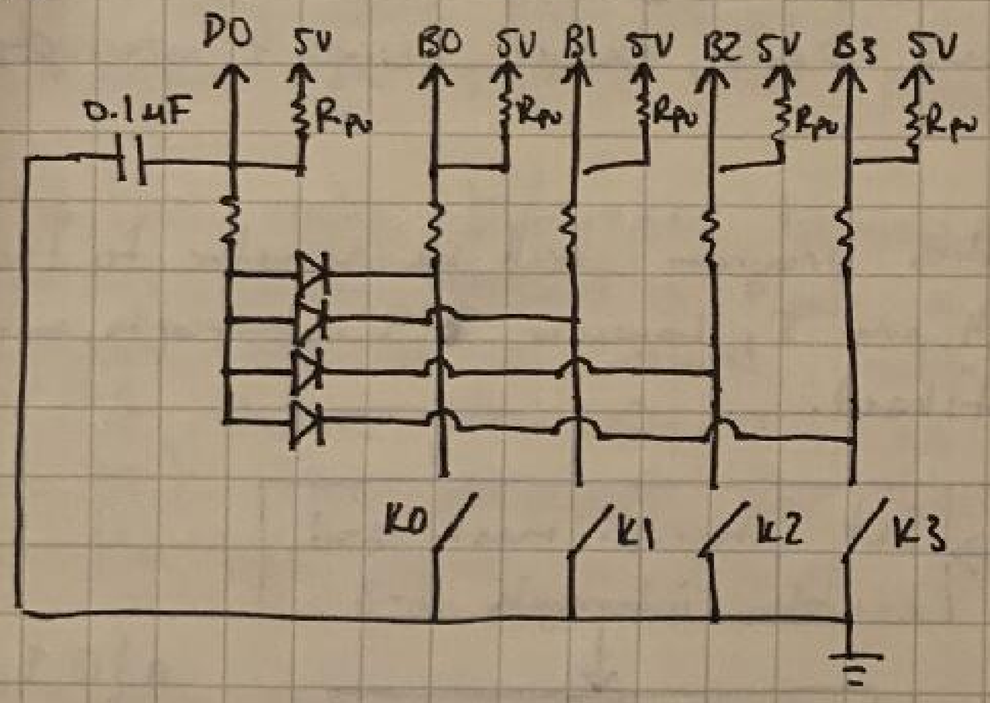
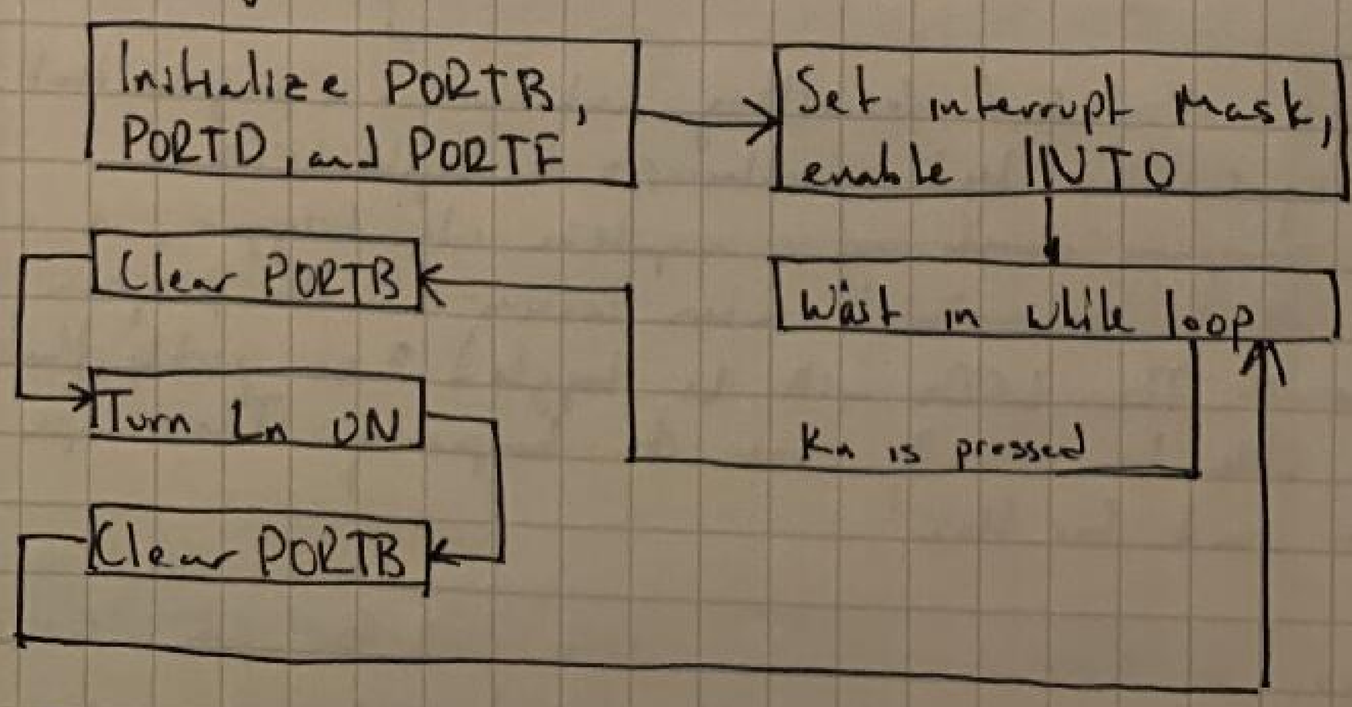


Figure 2-7: Single interrupt driven circuit with four key switches.

We also had the switches wired to PORTF so that we could read the value once the interrupt was entered. We initially had a lot of problems when trying to program the board, but it ended up being a short circuit due to our wiring.



da
9/6/19
3:37PM

Figure 2-8: Flowchart for single-interrupt driven LED toggle program.

Grant Ahuller
Brandon Lampert
9/6/18

Laboratory #2: External Interrupt Control of Switched LEDs

After rewiring the breadboard, we were able to program the board, and found that the program functioned perfectly, with the .01 μF capacitor acting as a debouncer.

Postlab:

1. The software design approach for this lab was the same as last lab: start with the problem, create a flowchart, write the code and program the board, then troubleshoot. The only thing I would do differently is saving each part of the lab code separately, but for this lab it wasn't a big deal.

2. The only major problem we had was not being able to program our board for part 7. This was eventually found to be due to a short circuit, but only after we had restarted the PC, swapped our board, and rewired every thing.

When using different edge triggers, we found that negative edge triggers were the hardest to see in action because the program is only in the interrupt for a very short time.

3. Standard resistor pull and grounding techniques were used throughout this lab to prevent board damage due to unexpected voltage spikes.

9/6/18
4:28~~75~~ PM

9-6-18
4:30 pm

Laboratory #2: External Interrupt Control of Switched LEDs

Grant Abella
Brandon Lampert
9/6/18

Laboratory Code:

Part 4:

```
/*
 * main.c
 * Program file containing main function
*/
#include "lab2.h"
int main(void)
{
    init_ports();
    init();
    sei ();
    while (1)
    {
    }
}
```

```
/*
 * lab2.h
 * Header file containing function prototypes
 */
#ifndef LAB2_H_
#define LAB2_H_

#define F_CPU 16000000UL //set board clock to 16MHz
#include <avr/io.h> //avr header for project
#include <util/delay.h> //
#include <avr/interrupt.h>

void init (void);
void init_ports(void);

#endif
```

```
/*
 * init.c
 * Program file containing function definitions
 * for functions declared in lab2.h
 */

#include "lab2.h"

void init (void)
{
    DDRA=0xFF; // set PORTA as an output
    DDRB=0x00; // set PORTB as an input port
}

void init_ports(void)
{
    EICRA = 0x01;
    EIMSK = 0x00; // low level trigger
}

ISR (INT0_vect)
{
    PORTB ^= 0x01;  // toggle LED0
}
```

Figure 2-9: Code for implementing logic to toggle LED0 when K0 is pressed using INT0 (low-level trigger).

The main.c file for this program contains only the main function which calls the init_ports function, init function, sei function to enable global interrupts, and sits in an infinite while loop.

Init.c contains definitions for init(), which sets the data directions of ports A and B, and init_ports, which enables INT0 and sets it to trigger upon detecting a low level on the input. The ISR for INT0 is also contained here and simply toggles LED0. Prototypes for these are found in lab2.h.

Part 5:

```
/*
 * init.c
 * Program file containing function definitions
 * for functions declared in lab2.h
 */

#include "lab2.h"

void init (void)
{
    DDRA=0xFF; // set PORTA as an output
    DDRB=0x00; // set PORTB as an input port
}

void init_ports(void)
{
    EICRA = 0x01;
    EIMSK = 0x02; // negative edge trigger
}

ISR (INT0_vect)
{
    PORTB ^= 0x01;  // toggle LED0
}
```

9/6/18
5:20PM

Figure 2-10: Code for implementing logic to toggle LED0 when K0 is pressed using INT0 (negative edge trigger).

Laboratory #2: External Interrupt Control of Switched LEDs

Grant Abella
Brandon Lampert

9/6/18

Since the only change made to the project files for this lab was to mit.c, only this file has been included to save space.

To change the type of trigger for the interrupt, the value for the interrupt mask was changed from 0x00 to 0x02. This value was found on page 90 of the Atmega 128A datasheet.

Part 6:

```
/*
 * init.c
 * Program file containing function definitions
 * for functions declared in lab2.h
 */

#include "lab2.h"

void init (void)
{
    DDRA=0xFF; // set PORTA as an output
    DDRB=0x00; // set PORTB as an input port
}

void init_ports(void)
{
    EICRA = 0x02;   // falling edge trigger
    EIMSK = 0x0f;
}

ISR (INT0_vect)
{
    PORTB ^= 0x01;  // toggle LED0
}

ISR (INT1_vect)
{
    PORTB ^= 0x02;  // toggle LED1
}

ISR (INT2_vect)
{
    PORTB ^= 0x04;  // toggle LED2
}

ISR (INT3_vect)
{
    PORTB ^= 0x08;  // toggle LED3
}
```

Figure 2-11: Code for implementing logic to toggle LEDs L0-L3 when K0-K3 are pressed using INT0-INT3 respectively.

Again, only mit.c was changed when implementing this program.

Because 4 interrupts are used, 4 separate ISRs had to be defined. *Each ISR contains a bitwise operation to toggle an LED depending on which interrupt is responsible for each key switch and LED. The value for EIMSK was changed to 0x0F to enable interrupts 0-3, EICRA was kept at a value of 0x02 for a falling edge trigger.

Part 7: (continued on next page)

GA
9/6/18
5:46 PM

Laboratory #2: External Interrupt Control of Switched LEDs

Grant Abella
Brandon Lampe
9/6/18

```
/*
 * init.c
 * Program file containing function definitions
 * for functions declared in lab2.h
 */

#include "lab2.h"

void init (void)
{
    DDRA=0xFF; // set PORTA as an output
    DDRB=0x00; // set PORTB as an input port
}

void init_ports(void)
{
    EICRA = 0x02;   // falling edge trigger
    EIMSK = 0x01;
}

ISR (INT0_vect)
{
  PORTF ^= ~PINB;  // toggle LED0-LED4
}
```

Figure 2-12: code for implementing logic to toggle LEDs L0-L3 when key switches K0-K3 are pressed using a single interrupt.

A falling edge trigger was used for this, as well as a single ISR function for INT0 which all 4 switches were wired to using diodes.

GA
9/6/18
5:59PM

As instructed by the lab handout, PORTF was toggled inside the ISR to observe toggling.

Scope pics : -0 6/6
Discussion : -0 10/10
Flowchart/code : -0 8/8
Circuit diagram : -6 5/11
Post lab : -0 5/5

34/40

Mom Jan

Grant Abella
Brandon Lampert
10/11/18

Laboratory #5: Liquid Crystal Display Interface: Part 1

Advisor: Dr. Gutchlag
Laboratory Objective: Develop interface techniques for control of the Liquid Crystal Display (LCD) on the Atmega128A based STK129/64 starter kit.
Equipment Used: Atmega starter kit #20
MSO 2014B oscilloscope EQ-3011
510 Ω resistor pack
Bat41 shottky diode
Breadboard
wires
pin headers

Time Out
4:49 PM
1.0.5c

1. In order to successfully write and implement a function to send a nibble (4bits) of data to the LCD, we know that we will first need to understand the procedure required to send a signal and have the LCD interpret it. *The flowchart for this is on page 19.

From the LCD data sheet, we understand this procedure to be as follows:

① Set the register select (RS) bit
- 0 for instruction register
- 1 for data register

② Set the R/W bit
- 0 for write
- 1 for read

③ Set the enable bit high
④ Write the desired data (4-bits in our case)
⑤ Set the enable bit low

It is important that timings are correct for this procedure or the LCD will not be able to properly process what is being sent to it. The LCD's datasheet is very helpful in this regard, and a timing diagram for writing to the LCD is included on the next page.

Table 13-1: Item specifications for timing tables in figure 14-1.

Item	Symbol	Min	Typ	Max	Unit
Enable cycle time	t_{CYCE}	500	—	—	ns
Enable pulse width (high level)	PW_{EH}	230	—	—	
Enable rise/fall time	t_{Er}, t_{Ef}	—	—	20	
Address set-up time (RS, R/$\overline{W}$ to E)	t_{AS}	40	—	—	
Address hold time	t_{AH}	10	—	—	
Data set-up time	t_{DSW}	80	—	—	
Data hold time	t_{H}	10	—	—	

10/11/18
4:14 PM

Grant Abella
Brandon Lampert
10/11/18

Laboratory #5: Liquid Crystal Display Interface: Part 1

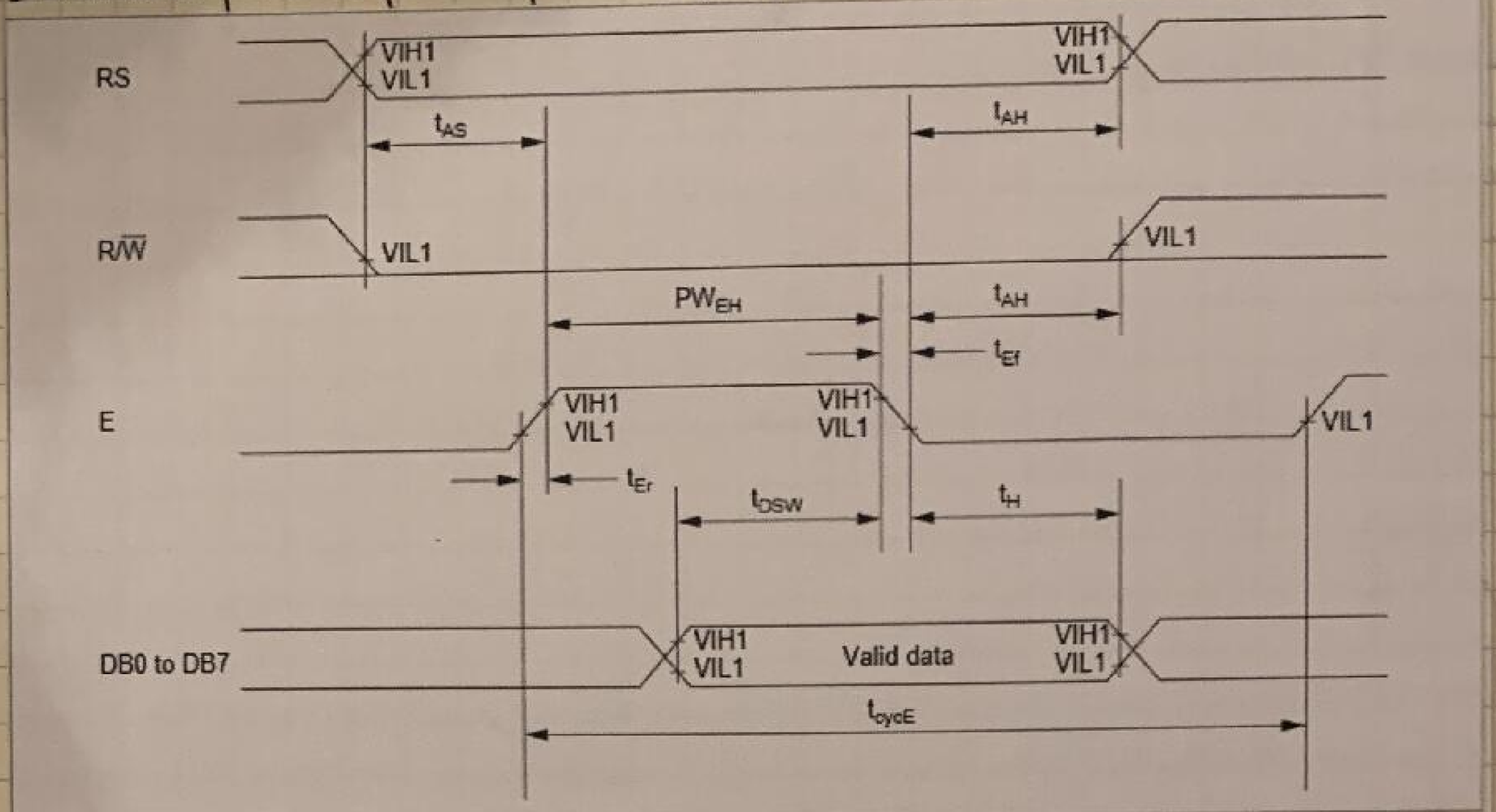


Figure 14-1: Timing diagram for performing write operation to the LCD, screenshotted from the LCD datasheet page 58.

The timing specifications for the above figure are located in table 14-1. All of these timings are important to follow, but they are all on a nanosecond scale. Our microcontroller does not perform delays at this small of a scale. Because of this, we will be using the standard delay functions (on a scale of µs and ms) that we have used in previous labs. This is fine to do since the times in between operations have a minimum period that we must wait before setting other data bits, but waiting longer will be fine.

In order to ensure that our timings are correct, we will attach our three non-data bits and one data bit wires to the oscilloscope so that we can inspect and troubleshoot the timings.

After much troubleshooting, we have captured the following on the oscilloscope:

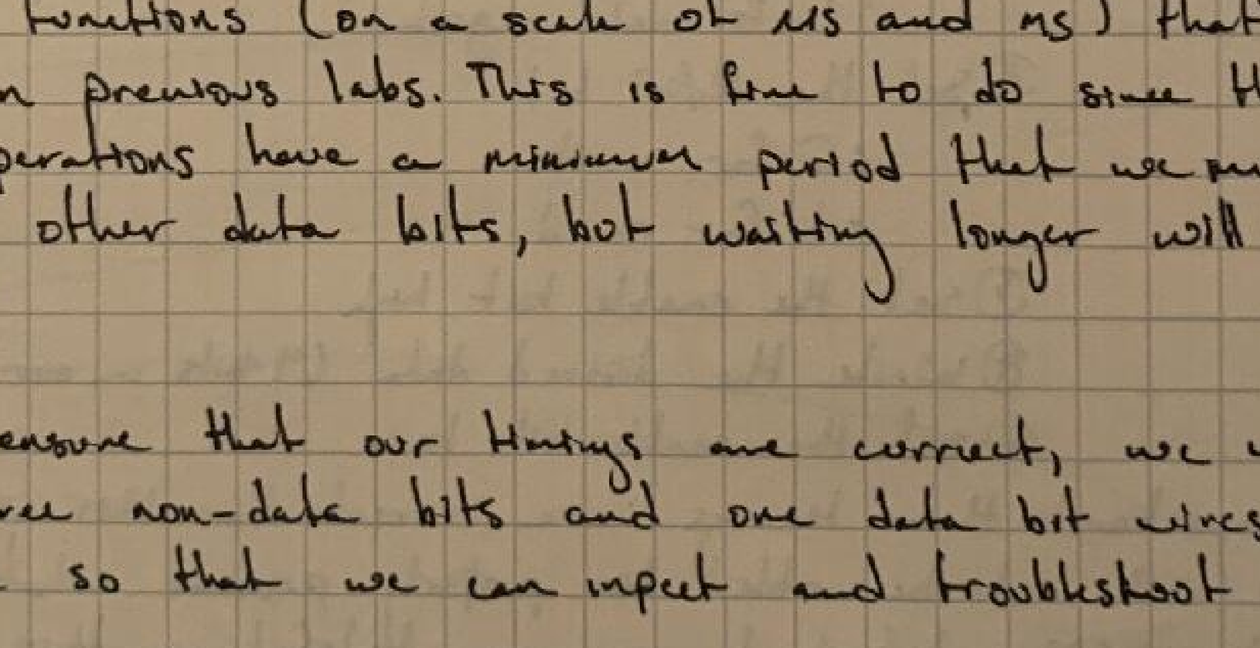


Figure 14-2: Diagram showing timings for RS, RW, E, and DB7 signals to be sent to the LCD.

10/11/18
5:10 PM

10-18-18
3:41 PM

Grant Abella
Brandon Lampe
10/16/18

Laboratory #5: Liquid Crystal Display Interface: Part 1

As mentioned before, these times are longer than those specified in the datasheet, however they are in the correct order and function just as well.

Our function for writing a nibble to the LCD is shown below:

```
// Function to write a nibble of data to the LCD.
void WriteNibbleToLCD(uint8_t SelectedRegister, uint8_t nibble)
{
    _delay_ms(1);
    uint8_t lower = nibble & 0x0F;                    // extract lower nibble
    if (SelectedRegister == COMMAND_REGISTER)
    {
        // Register select bit
        LCD_OUT = LCD_OUT | 0x00;
        _delay_ms(1);
        // R/W
        LCD_OUT = LCD_OUT | 0x00;
        _delay_ms(1);
        // Enable bit high
        LCD_OUT = BV(ENABLE);
        _delay_ms(1);
        // Write valid data
        LCD_OUT = LCD_OUT | lower;
        _delay_ms(1);
        // Enable bit low
        LCD_OUT = LCD_OUT & 0x0F;
        _delay_ms(1);
    }
    if (SelectedRegister == DATA_REGISTER)
    {
        // Register select bit
        LCD_OUT = BV(RS);
        _delay_ms(1);
        // R/W
        LCD_OUT = LCD_OUT | 0x00;
        _delay_ms(1);
        // Enable bit high
        LCD_OUT = BV(ENABLE);
        _delay_ms(1);
        // Write valid data
        LCD_OUT = LCD_OUT | lower;
        _delay_ms(1);
        // Enable bit low
        LCD_OUT = LCD_OUT & 0x0F;
        _delay_ms(1);
    }
}
```

Figure 15-1: C code for a function to write a nibble of data to either the command or data register on the LCD.

10/16/18
3:15PM

This function begins by extracting the lower part of the variable named nibble. From here, it checks which register to write to. For both registers, the process is generally the same, with the only difference being the setting of the register select bit. Following the write procedure, we first set the register select bit. After this, the R/W bit is set to 0 since we will be writing. Next, the enable bit is set high to prepare for us to write. After this, the 4 bits from nibble are sent to the LCD. Finally, the enable bit is set low, and the action is complete.

Grant Abelle
Brandon Layert
10/16/18

Laboratory #5: Liquid Crystal Display Interface: Part 1

Using this function, we can now move on to the next part of this lab.

2. The following table describes the instructions that can be sent to the LCD. Many of these will be used in our initialization for the LCD to meet the specifications outlined in the lab.

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (max) (when f_{cp} or f_{osc} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*

I/D = 1: Increment
I/D = 0: Decrement
S = 1: Accompanies display shift
S/C = 1: Display shift
S/C = 0: Cursor move
R/L = 1: Shift to the right
R/L = 0: Shift to the left
DL = 1: 8 bits, DL = 0: 4 bits
N = 1: 2 lines, N = 0: 1 line
F = 1: 5 × 10 dots, F = 0: 5 × 8 dots
BF = 1: Internally operating
BF = 0: Instructions acceptable

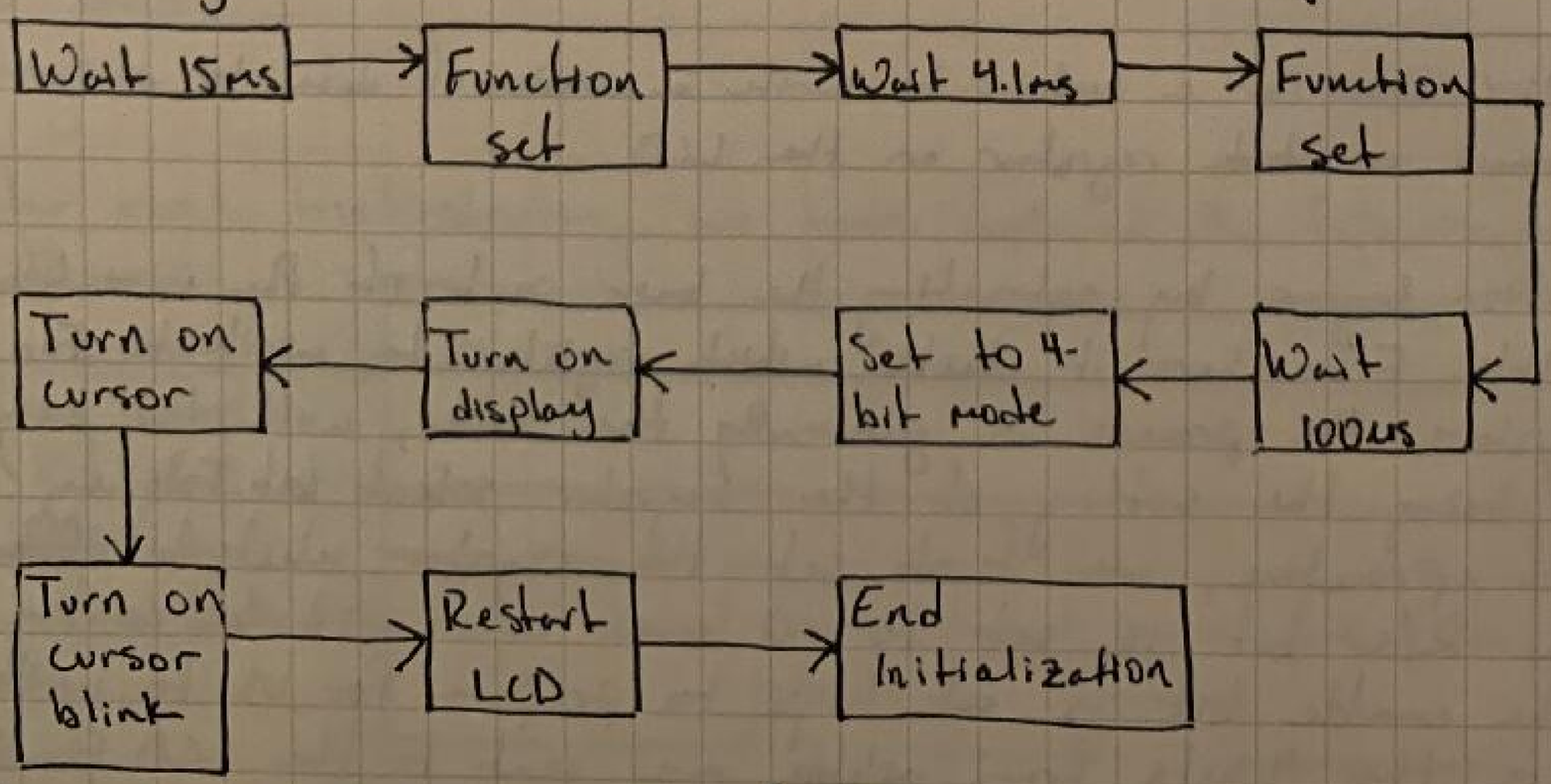
DDRAM: Display data RAM
CGRAM: Character generator RAM
ACG: CGRAM address
ADD: DDRAM address (corresponds to cursor address)
AC: Address counter used for both DD and CGRAM addresses

Execution time changes when frequency changes
Example:
When f_{cp} or f_{osc} is 250 kHz,
$37\ \mu s \times \frac{270}{250} = 40\ \mu s$

Note: — indicates no effect.

Table 16-1: list of instructions that can be sent to the LCD with descriptions and execution times for each command.

After reading through the LCD datasheet we have concluded that the following flowchart will describe initialization logic for the LCD:



10/16/18
4:10 PM

Figure 16-1: Initialization logic for the LCD.

Grant Abella
Brandon Lapp
10/16/18

Laboratory #5: Liquid Crystal Display Interface: Part 1

As previously, the timings in this part are important, and while there are minimum wait times that we must consider, waiting for a longer time will not matter.

The code for our function that utilizes the logic shown in figure 16-1 is shown below:

```
// Function to initialize the LCD
void initializeLCD(void)
{
    _delay_ms(20);

    WriteNibbleToLCD(COMMAND_REGISTER, 0x03);   // function set
    _delay_ms(5);

    WriteNibbleToLCD(COMMAND_REGISTER, 0x03);   // function set
    _delay_ms(5);

    WriteNibbleToLCD(COMMAND_REGISTER, 0x03);   // function set

    WriteNibbleToLCD(COMMAND_REGISTER, 0x02);

    WriteNibbleToLCD(COMMAND_REGISTER, 0x02);   // 4-bit mode
    WriteNibbleToLCD(COMMAND_REGISTER, 0x08);

    WriteNibbleToLCD(COMMAND_REGISTER, 0x00);
    WriteNibbleToLCD(COMMAND_REGISTER, 0x0F);   // display on, cursor on, blinking

    WriteNibbleToLCD(COMMAND_REGISTER, 0x00);  // restart LCD
    WriteNibbleToLCD(COMMAND_REGISTER, 0x01);

    WriteNibbleToLCD(COMMAND_REGISTER, 0x00);  // end of initialization
    WriteNibbleToLCD(COMMAND_REGISTER, 0x06);
}
```

Figure 17-1: C code implementation of initialization logic for the LCD.

Our code follows the flowchart logic pretty closely. We begin with 3 function set commands as outlined in the datasheet, proper wait times are used. After this, we set the LCD to interface with the board in 4-bit mode. 5x8 font and 2-line mode are set. The cursor is enabled and set to blink. Finally, the LCD is cycled on and off, and initialization is finished. With these settings, the cursor begins at the home position and auto-increment is in effect.

The only issue we had during this implementation was waiting 100 us after the second function set command. We had first set the wait time to this value exactly, but changed it when we had issues. We do not know why the 100 us wait was causing problems.

10/16/18
4:45 PM

The only reason for this that comes to mind is that the LCD possibly just needed time to catch up on processing

Grant Abella
Brandon Lampert
10/16/18

Laboratory #5: Liquid Crystal Display Interface: Part 1

Now that we have successfully achieved LCD initialization using this function, we are ready to move on to the final part of this lab.

3. The process for designing a function to send an entire byte will be simple since we already have the function to send half of a byte. This function will need to have the same two arguments: one for the register select, and another for the byte.

A general flowchart for our WriteByteToLCD function is shown below:

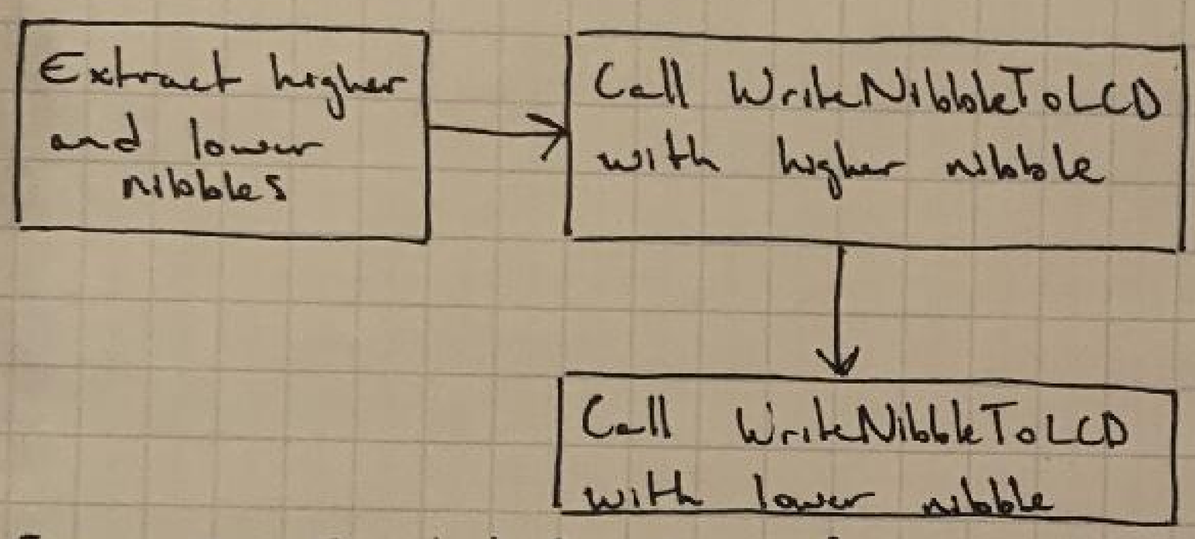


Figure 18-1: Flowchart for logic of the WriteByteToLCD function.

The writing of this function was very quick, our code is shown below:

```
// Function to send a full byte to the LCD
void WriteByteToLCD(uint8_t SelectedRegister, uint8_t byte)
{
    uint8_t higher, lower;
    higher = byte >> 4;
    lower = byte & 0x0F;
    WriteNibbleToLCD(SelectedRegister, higher);
    WriteNibbleToLCD(SelectedRegister, lower);
}
```

Figure 18-2: Code for our function to write a byte to the LCD.

This function extracts the higher 4 bits of the byte using a bitwise shift of 4 to the right. The lower 4 bits are extracted using an AND operation.

Lee
10/18/18
2:19 PM

Using this function, we can now reuse our LCD initialization function.

Our reused function is shown in figure 19-1 on the next page.

Grant Abella
Brandon Leggett
10/18/18

Laboratory #5: Liquid Crystal Display Interface: Part 1

```
// Function to initialize the LCD
void initializeLCD(void)
{
    _delay_ms(20);

    WriteNibbleToLCD(COMMAND_REGISTER, 0x03);   // function set
    _delay_ms(5);

    WriteNibbleToLCD(COMMAND_REGISTER, 0x03);   // function set
    _delay_ms(5);

    WriteByteToLCD(COMMAND_REGISTER, 0x32);     // function set
    WriteByteToLCD(COMMAND_REGISTER, 0x28);     // 4-bit mode
    WriteByteToLCD(COMMAND_REGISTER, 0x0F);     // display, cursor on, blinking
    WriteByteToLCD(COMMAND_REGISTER, 0x01);     // clear display
    WriteByteToLCD(COMMAND_REGISTER, 0x06);     // entry mode set, ready for data

}
```

Figure 19-1: Revised initializeLCD function utilizing WriteByteToLCD.

We replaced most of the instructions that were originally inside this function with the new function by combining pairs of nibble commands into a single byte command.

We implemented this new function to ensure that everything worked correctly and we were pleased that it did. The LCD initialized as expected, ensuring that the new function works as intended.

We encountered no difficulties during this implementation.

Flowchart for part 1: The logical flowchart for our function to write a nibble to the LCD was as follows:

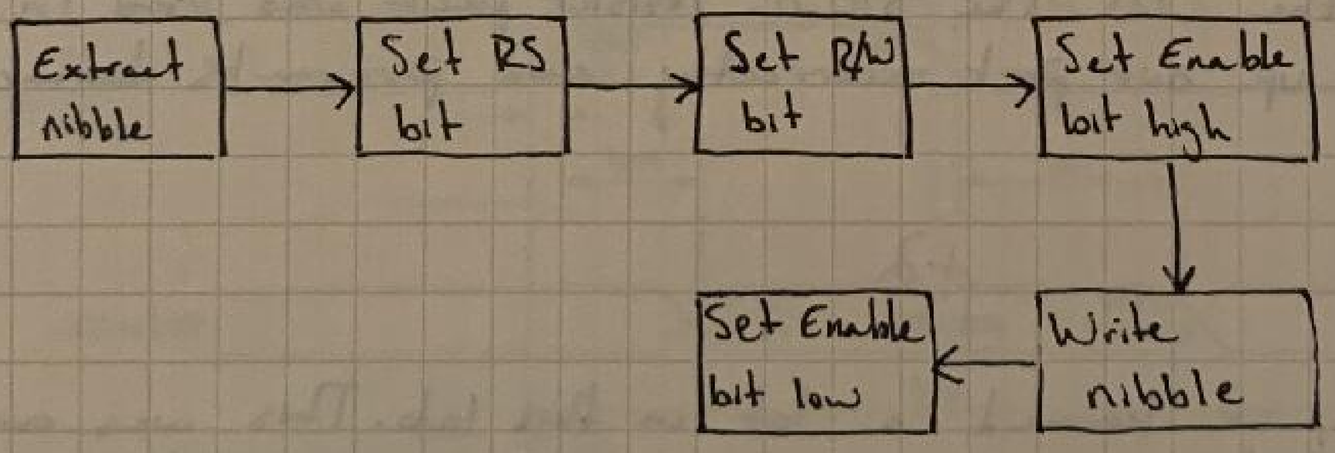


Figure 19-2: Flowchart describing the logic to write a nibble of data to the LCD.

The logical flow of this diagram is based on the timing diagram in figure 14-1.

10/18/18
3:00 PM

PostLab: +5

1. Our approach to designing the software in this lab was to first

Grant Abella
Brandon Lampert
10/18/18

Laboratory #5: Liquid Crystal Display Interface: Part 1

understand the timings and signals associated with the LCD. The reason for this was that we wouldn't be able to diagnose any software issues if we didn't fully understand how the program speaks with the LCD. The way that we verified our timings was discussed earlier in this lab, but basically attached each of the bits responsible for signaling the LCD to the osculloscope. This allowed us to debug our function easily because we could & see the order and timings of these signals on the scope.

In the future, it would help to utilize the LEDs on the board to debug things like messages going to the LCD. In this lab, it could have saved us a little time had we done this. We will be utilizing this for the other parts of this LCD lab.

2. The only real difficulty we had during this lab was interfacing with the LCD, and this was ultimately a result of the timings discussed above. Once we had read through the datasheet and fully understood the process of writing an instruction to the LCD, things went more smoothly.

3. In order to protect the hardware in this lab, we utilized a 510Ω resistor pack and a diode, as we have in previous labs. The purpose of the diode was to protect the LCD in case the header was plugged in backwards. This was not necessary, but was a good precaution because this mistake could happen and would fry the LCD. The 510Ω resistor pack was used in case we set up our port incorrectly, and prevents damage to the LCD.

Conclusion:

In conclusion, we learned a lot in this lab. This was our first time interfacing with an LCD, and we learned many helpful techniques that will make future applications much easier to implement and debug. Additionally, with LCDs being such a common peripheral, we are bound to use this newfound knowlege again in the future.

GC
10/18/18
5:00PM

+2

Code : ~ 5/5
Flowchart : ~ 2/2
Diagram : ~ 1/1
Discussion : ~ 2/2

+9 19/10

Grant Abella
Brandon Lampert
10/23/18

Laboratory #5: Liquid Crystal Display: Part 2

Advisor: Dr. Gutschlag

Laboratory Objective: Continue to develop interfacing techniques for control of the Liquid Crystal Display (LCD) on the Atmega128A based STK128/64 starter kit.

Equipment Used:
- Atmega starter kit #34
- MSO 2014B Osulloscope EQ-3011
- 510Ω resistor pack
- Bat41 shottky diode
- Breadboard
- Wires
- Pin headers

1. Similar to Part 1 of this lab, just as we had to first get our timings correct for the writing to the LCD, we know that in order to successfully read information from the LCD we will need to have the correct timings programmed.

Below is the timing diagram describing the order that operations must occur for a read operation:

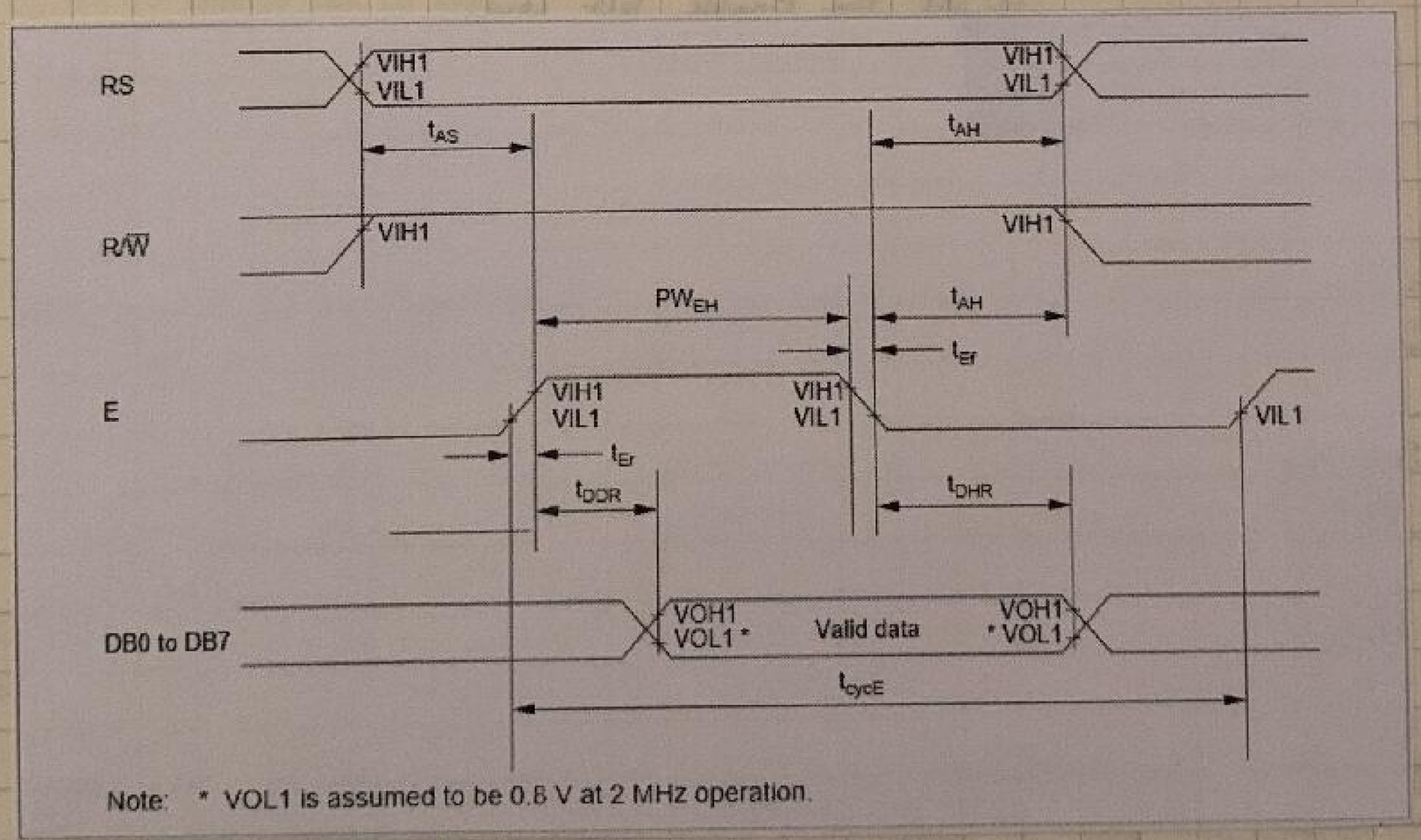


Figure 21-1: Timing diagram for read operation on the LCD. Taken from page 58 of the datasheet

GA
10/23/18
2:40 PM

The following table defines the times of each operation on an order of nanoseconds.

Grant Mackie
Brendon Lampert
10/23/18

Laboratory #5: Liquid Crystal Display Interface: Part 2

Table 22-1: Time definitions for the figure on the previous page.

Item	Symbol	Min	Typ	Max	Unit	Test Condition
Enable cycle time	t_{CYCE}	500	—	—	ns	Figure 26
Enable pulse width (high level)	PW_{EH}	230	—	—		
Enable rise/fall time	t_{Er}, t_{Ef}	—	—	20		
Address set-up time (RS, R/$\overline{W}$ to E)	t_{AS}	40	—	—		
Address hold time	t_{AH}	10	—	—		
Data delay time	t_{DDR}	—	—	160		
Data hold time	t_{DHR}	5	—	—		

So, according to this information, the order of operations for reading from the LCD in 4-bit mode is:

1. Set the Register Select bit
   - 0 for instruction
   - 1 for data
2. Set the R/W bit
   - 0 for write
   - 1 for read
3. Set the Enable bit high
4. read the data (4-bits)
5. set the Enable bit low

* It is very important to note that since we are operating in 4bit mode, two read operations will need to be performed to read the full byte and keep the LCD functioning.

Given this procedure, our flowchart for the ReadNibbleFromLCD function is as follows:

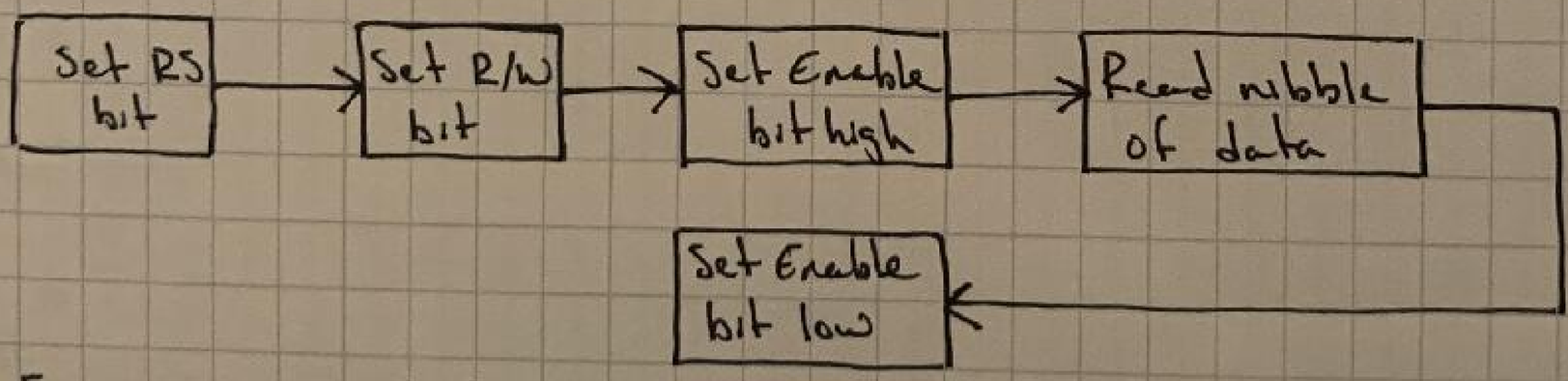


Figure 22-1: Flowchart describing logic to read a nibble of data from the LCD.

Our function implementing the logic from the flowchart above is shown in figure 23-1 on the next page.

GA
10/23/18
3:20 PM

Grant Abella
Brandon Lampert
10/23/18

Laboratory #5: Liquid Crystal Display Interface: Part 2

```
// Function to read a nibble of data from either the
// COMMAND or the DATA register of the LCD
uint8_t ReadNibbleFromLCD(uint8_t selectedRegister)
{
    _delay_us(1);
    uint8_t data = 0x00;
    if (selectedRegister == COMMAND_REGISTER)
    {
        // R/W
        LCD_OUT = BV(RW);
        _delay_us(1);
        // Change DDR to input
        LCD_DDR = 0xF0;
        LCD_OUT |= 0x0F;
        _delay_us(1);
        // Enable bit high
        LCD_OUT |= BV(ENABLE);
        _delay_us(1);
        // Read valid data
        data = LCD_IN & 0x0F;
        _delay_us(1);
        // Enable bit low
        LCD_OUT ^= BV(ENABLE);
        _delay_us(1);
        // Change DDR to output
        LCD_DDR = 0xFF;
    }
    if (selectedRegister == DATA_REGISTER)
    {
        // Register select bit
        LCD_OUT = BV(RS);
        _delay_us(1);
        // R/W
        LCD_OUT |= BV(RW);
        _delay_us(1);
        // Change DDR to input
        LCD_DDR = 0xF0;
        LCD_OUT |= 0x0F;
        _delay_us(1);
        // Enable bit high
        LCD_OUT |= BV(ENABLE);
        _delay_us(1);
        // Read valid data
        data = LCD_IN & 0x0F;
        _delay_us(1);
        // Enable bit low
        LCD_OUT ^= BV(ENABLE);
        _delay_us(1);
        // Change DDR to output
        LCD_DDR = 0xFF;
    }
    return data;
}
```

Figure 23-1: C code implementation of logic shown on page 22 for reading a nibble of data from the LCD.

In order to test whether or not this function works, we followed the suggestion made in the lab handout and attempted to read the address counter. After a return home command is passed to the LCD, the address counter should read zero. So, to test the function, we wrote several random bytes to the LCD, outputted the resulting instruction address to the LEDs, then did the same after a return home command was sent to the LCD. We found that the function worked because the LEDs showed a value of 43 before the return home, and then 0 after the command.

10/23/18
4:00 PM

Grant Abelle
Brandon Lampert
10/23/18

Laboratory #5: Liquid Crystal Display Interface Part 2

2. Now that we have our ReadNibbleFromLCD function working, we can begin working on the WaitForLCD function.

The basic logic for this function is that when the LCD is busy processing a command, the busy flag (DB7) of the LCD will have a value of 1. This flag can be read when the RS value of the LCD is 0 and the R/W value is 1. This is the case when a read of the command register is made.

A flowchart describing the general logic for this function is shown below:

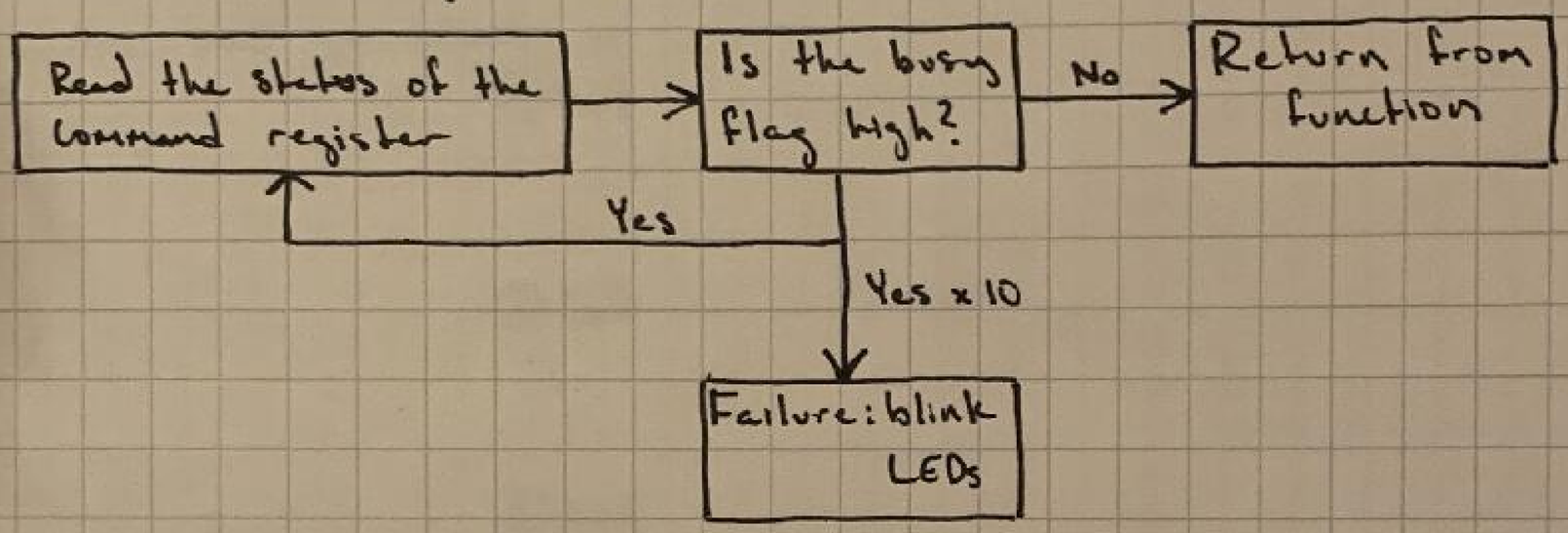


Figure 24-1: Flowchart for WaitForLCD logic.

The function polls the LCD's busy flag, if the flag is low, the LCD is ready for another operation, so the function exits. Otherwise the LCD is polled again. If after 10 polls of the busy flag, the LCD is still busy, the function stops polling the device and blinks the board's LEDs indefinitely, denoting failure. Our code for this function is shown below:

```
// Function to poll the LCD busy flag until the LCD is
// ready for more operations. The flag is polled at most
// ten times, failure occurs if still busy after this.
void waitForLCD(void)
{
    uint8_t status, i;
    for(i=0; i<10; i++)
    {
        status = ReadByteFromLCD(COMMAND_REGISTER);
        if((status & 0x80) == 0x80)      // if busy
        {
            _delay_us(10);
        }
        else
        {
            i = 11;                      // break the loop
        }
    }
    if (i==10)                  // if busy for 10 iterations
    {
        while(1)
        {
            // flash LED0 indicating failure
            _delay_ms(500);
            LED_OUT ^= 0x01;
        }
    }
}
```

Figure 24-2: C code for the WaitForLCD function.

10/25/18
2:09PM

Grant Abulka
Brandon Lampert
10/25/18

Laboratory #5: Liquid Crystal Display [redacted] Interface: Part 2

It is important to note that initially we incorporated two calls to the readNibble function, with the status being saved on the second read, but this was later changed to a call to the ReadByte function for convenience. When we implemented this function, we were pleased to find that it worked. Our system of testing consisted of calling the WaitForLCD function continuously, and then unplugging the LCD from the board. When we performed this test, the LED on the board began blinking, indicating the failure to communicate with the LCD.

3. In order to read a full byte of data from the LCD, we will just need to design a function to read a nibble of data twice, and then combine the nibbles into a single byte. It is possible that a small delay may be necessary in between calls to the ReadNibble function. A flowchart representing this logic is shown below:

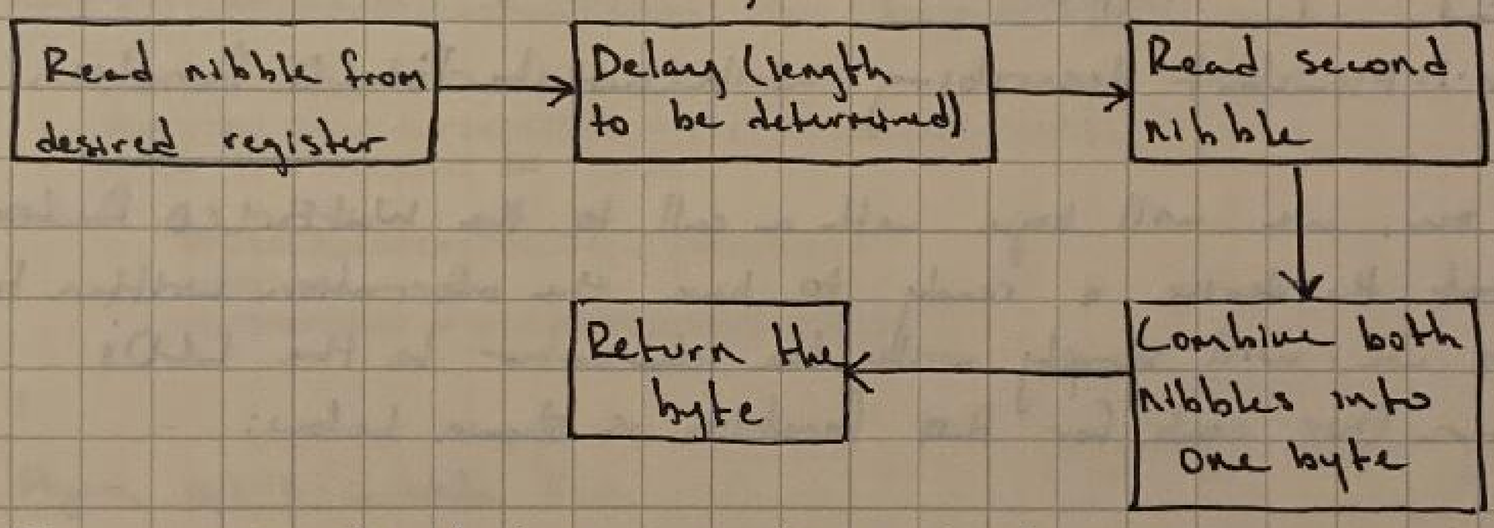


Figure 25-1: Flowchart for ReadByte function logic.

When first testing this function, we experimented with different delays between nibble reads. After much trial and error, we successfully extracted an accurate ~~byte~~ byte of data from the LCD by utilizing a delay of 125 us. Our function is shown below:

```
// Function to read a byte from the LCD
uint8_t ReadByteFromLCD(uint8_t selectedRegister)
{
    uint8_t byte = 0x00;
    byte = ReadNibbleFromLCD(selectedRegister) << 4;
    _delay_us(125);
    byte |= ReadNibbleFromLCD(selectedRegister);
    return byte;
}
```

Figure 25-2: C code for ReadByteFromLCD function.

10/25/18
2:40PM

As explained above, once we got the delay time determined, the function ran as intended. We used the same technique for testing as we did for our readNibble function. The address counter was read accurately several times with this function to determine that it worked correctly.

Grant Abella
Brandon Lapetina
10/25/18

Laboratory #5: Liquid Crystal Display Interface: Part 2

We followed the logic shown in the flowchart pretty closely with the only difference being that we utilized a bit shift in order to orient the nibbles in the correct order within the returned byte.

4. The process for writing a character to the LCD will be very ~~sssss~~ simple. While most of the functions we have created don't utilize the LCD's data register, this one will, since we are dealing with writing something to the screen, rather than executing a command. Our flowchart for this functional logic is as follows:

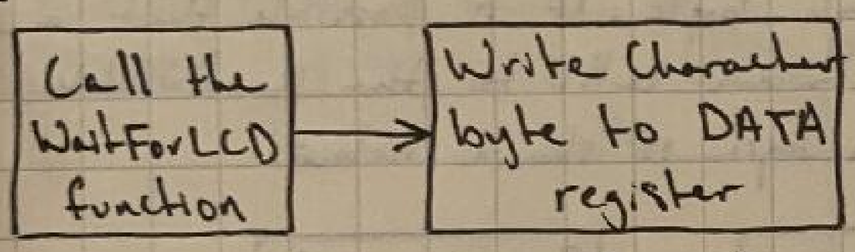


Figure 26-1: Flowchart describing WriteCharacterToLCD function.

As you can see, we will begin with a call to the WaitForLCD function, to ensure that the device is ready to have the information written to it. After this we will simply write the character to the LCD's DATA register. Our code for this function is shown below:

```
// Function to write a character to display on the LCD
void WriteCharacterToLCD(unsigned char character)
{
    waitForLCD();
    WriteByteToLCD(DATA_REGISTER, character);
}
```

Figure 26-2: Code for writing an ASCII character to the LCD.

Initially we were concerned that we would need to perform a cast on the character, however this was not necessary since the character is automatically converted into the ASCII value. With this, we tested the function by passing several characters to it. All of the tests were successful.

5. We are now able to move on to writing a function to output an entire string of characters. This function will essentially consist of calling the writeCharacter function for each character within the string. With this said, we will need to find a way to know when we have reached the end of the string so we can stop writing. There are a couple ways to do this, and we will be using a method that checks for the string's null character denoting the end of the string. The flowchart for this function is shown on the next page.

10/25/18
3:33 PM

Grant Ahrens
Brandon Lampert
10/25/18

Laboratory #5: Liquid Crystal Display Interface: Part 2

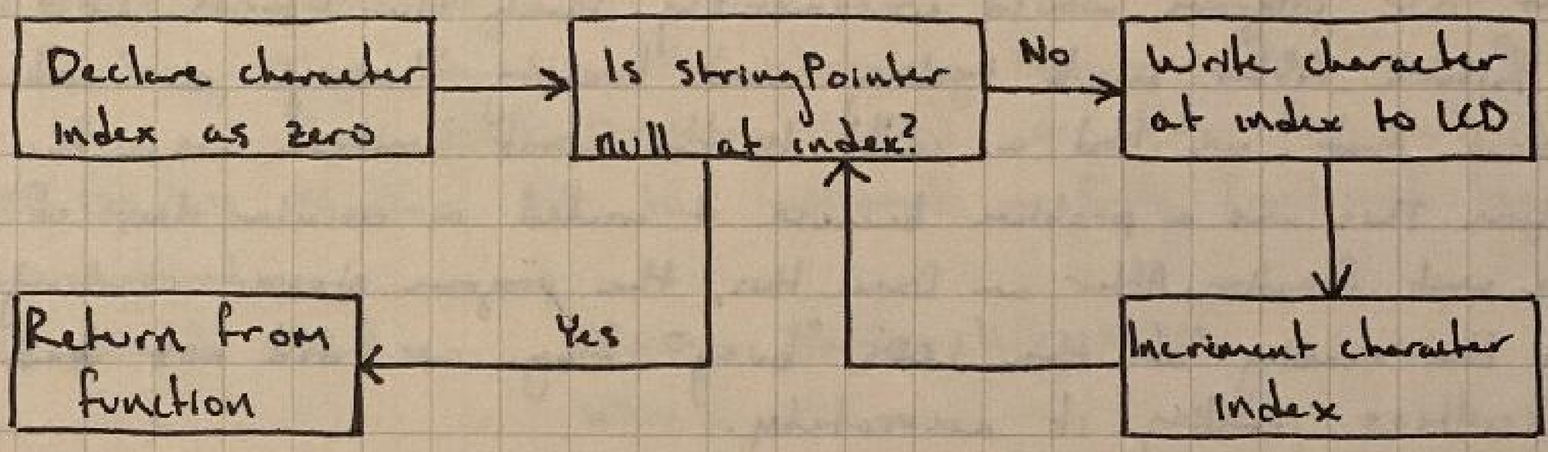


Figure 27-1: Flowchart describing logic for printing a string to the LCD.

This logic should be fairly simple to implement, however we won't know until we begin writing it.

Our function's c code is shown below:

```
// Function to write a string of characters to the LCD
void WritePointerStringToLCD(const char *stringPointer)
{
    uint8_t count = 0;
    unsigned char currentChar = stringPointer[count];
    while (currentChar != '\0')
    {
        WriteCharacterToLCD(currentChar);
        count++;
        currentChar = stringPointer[count];
    }
}
```

Figure 27-2: C code for implementation of the logic described above.

After writing this function, we tested it by outputting the traditional "Hello World" text to the screen. We found that the function performed as expected when the full string appeared on the screen. We were pleased to find that no indexes were being accessed that were outside the bounds of the character array as this was an initial concern that we had.

+5

PostLab:

1. Our approach to the software design for this lab was the same as in the first part of this lab. We began by thinking about the logic associated with what we needed to do. After this we drafted flowcharts and designed functions to reflect the logic from the flowcharts. This time around, we utilized the LEDs more than we did in the previous part. One example of this was in testing the read functions, we output the address counter to the LEDs to ensure that the functions were getting an accurate reading.

10/27/18
3:00PM

2. In this lab we did not encounter any hardware difficulties, but we did experience some software issues. In particular, we

Grant Abella
Brandon Lampert
10/27/18

Laboratory #5: Liquid Crystal Display Interface: Part 2

found that our program would continuously flash the board's LED indicating failure. After much testing and tracing through our code we realized that we had a call to the wait function inside our read function. This was a problem because it created a circular loop of calls to the wait function. After we fixed this, the program stopped crashing. As far as the reading of the LCD's "busy" flag, we did not find any difficulties reading it accurately.

3. We used the standard method of placing a 510 Ω resistor pack in between the board pins and the LCD to protect the board. This was important during the writing of the read functions because we were changing data directions for the board port.

Conclusion: In conclusion, we learned a lot in this lab regarding the performing of read operations on the LCD. It was similar to performing write operations, however it was more useful because we were able to utilize reads to gather information from the device. Reading the busy flag was a useful function as well because we could tell if the LCD was ready for more commands. This avoids issues such as the LCD crashing or incorrect read or write operation executions.

10/27/18
4:30PM

Code ! -0 6/6
Discussion ! -0 4/4
+7 17/10

Grant Abella
Brandon Lanfort
11/8/18

Laboratory #5: Liquid Crystal Display Interface: Part 3

Advisor: Dr. Gutschlag

Laboratory Objective: Use the interfacing techniques developed for control of the Liquid Crystal Display (LCD) on the Atmega128A-based STK128/64 Starter Kit to implement a fully functional keypad and LCD system.

Equipment Used: Atmega Starter Kit #34
Wires

1. A function to clear the LCD will be very simple to design. As there is already a command that can be sent to the LCD's command register to clear the screen, the function will take advantage of this.

A high-level flowchart describing the logic for this function is shown below:

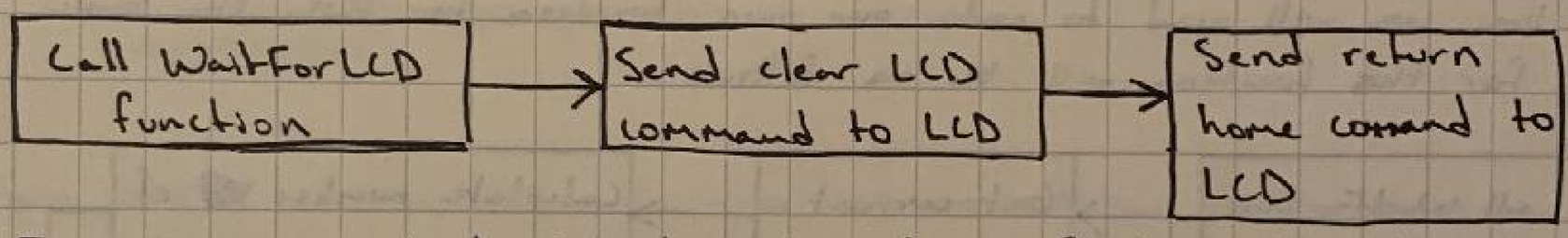


Figure 29-1: Flowchart describing logic for a function to clear the LCD screen.

The basic logic for the function is to begin by waiting until the LCD is ready to be written to. This is important because if we were to write to the LCD's command register while it is still executing something, undesired results could occur.

After this, we will send the command to clear the LCD. This will be sent to the command register. The specific command for this is 0x01 sent to the command register.

Finally, we will send the command to return home, meaning the cursor will be placed in the first position on the LCD. This is for convenience as because if the screen is cleared, most likely the user will want to be at the home position. The specific command for this is 0x02, sent to the command register. The function will return after this.

11/8/18
2:40PM

With this plan in mind, we are ready to begin writing this function.

11-13-18
4:55pm

Our code for this function is shown on the next page.

Laboratory #5: Liquid Crystal Display Interface: Part 3

Grant Abella
Brandon Lengert
11/8/18

```
// Function to clear the LCD
void ClearLCD(void)
{
    waitForLCD();                    // wait till LCD is ready
    WriteByteToLCD(COMMAND_REGISTER, 0x01); //clear LCD
    WriteByteToLCD(COMMAND_REGISTER, 0x02); //return Home
}
```

Figure 30-1: C code implementation for a clear LCD function.

We tested this function by writing several characters to the screen and calling the function above. When the screen cleared and the cursor was placed in the home position, we knew that the function worked as intended.

With this completed, we are ready to move on to the next part of this lab.

2. Since there is no built-in command to move the LCD's cursor to a new line, we will need to make our own function for this. The basic logic for this function will be as follows:

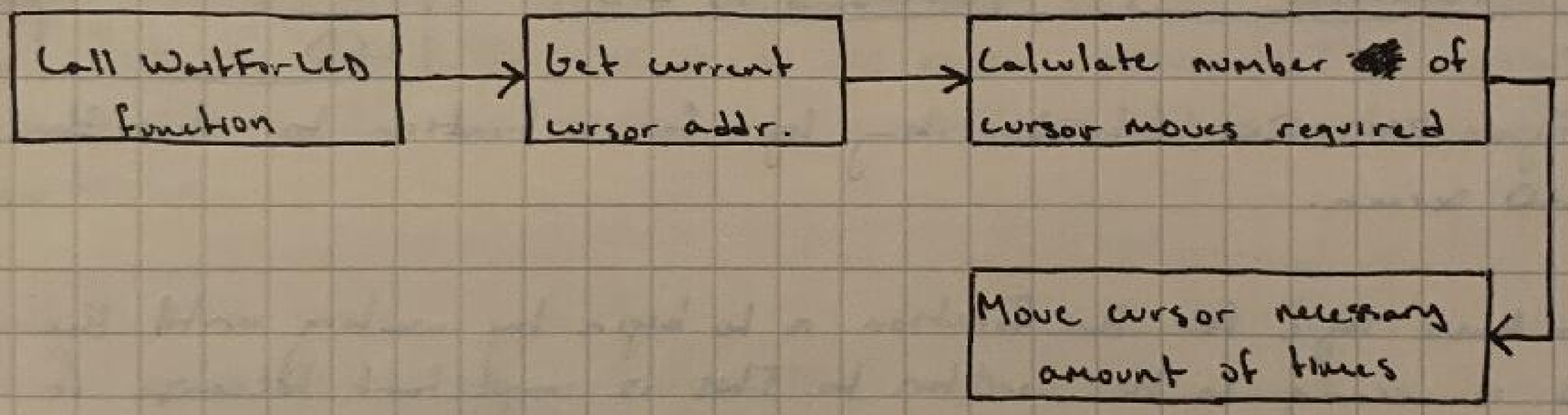


Figure 30-2: Flowchart describing logic for moving the cursor to the next line.

There are a couple of things to consider with this:

① There will need to be special handling for when a newline is called on the last line of the LCD. We will want a "rollover" effect so that the cursor is placed on the first line of the screen.

② Since the LCD can move the cursor either left or right, it may be worth considering changing the direction depending on what will be most efficient.

③ Because of the way that the controller interfaces with the LCD, the addresses for each line are not sequential. We will need to take this into account when writing the logic for this function.

DA
11/8/18
3:00 PM

With these points in mind, we will begin developing this function.

Our code for this function is shown on the next page.

Grant Abella
Brandon Lampert
11/8/18

Laboratory #5: Liquid Crystal Display Interface: Part 3

```
// Function to go to the next line on the LCD
void NewlineLCD(void)
{
    waitForLCD();                                       // wait till LCD is ready
    uint8_t currentAddress = ReadByteFromLCD(COMMAND_REGISTER);
    uint8_t upperAddress = currentAddress & 0xF0;       // upper nibble of address counter
    uint8_t direction = cursorRight;                    // sets cursor direction right
    uint8_t count=0;                                    // amount of shifts required
    currentAddress &= 0x0F;                             // lower nibble of address counter
    switch(upperAddress)                                // check the upper address, set the
    {                                                   // number of times to shift
        case row1:
            count = 40 - currentAddress;
            break;
        case row2:
            count =  24 + currentAddress;
            direction = cursorLeft; //left
            break;
        case row3:
            count = 40- currentAddress;
            break;
        case row4:
             WriteByteToLCD(COMMAND_REGISTER, 0x02); //return Home
             return;
    }
    // move the cursor the necessary amount of times
    for(uint8_t i = 0; i<count; i++)
    {
        WriteByteToLCD(COMMAND_REGISTER, direction);
        waitForLCD();
    }
}
```

Figure 31-1: C code for function to move LCD to next line of the screen.

When developing this function, we came across one problem. When calculating the number of shifts to make to go from row two to row three, our cursor was moving to an undesired place on the LCD. After much testing, we discovered that the reason for this was that instead of adding 24 to the current address, we were subtracting. Once we fixed this, our tests functioned as intended. Our tests consisted of writing several characters to each line on the LCD followed by a call to the newline function. This allowed us to make sure that we were able to adjust the cursor to the correct address no matter where the cursor was previously.

3. [redacted] When writing a function to perform a backspace on the onscreen text, we need to remember that this is really three operations:
   1. Move the cursor to the previous position
   2. Write a space character
   3. Move the cursor to the previous position

SR
11/13/18
2:14 PM

Of course, special consideration will need to be taken in the case of a backspace being applied in the first column of the LCD. This will require moving the cursor to the previous line. This logic is all shown in the flowchart on the next page.

Grant Abelle
Brandon Lampert
11/13/18

Laboratory #5: Liquid Crystal Display Interface: Part 3

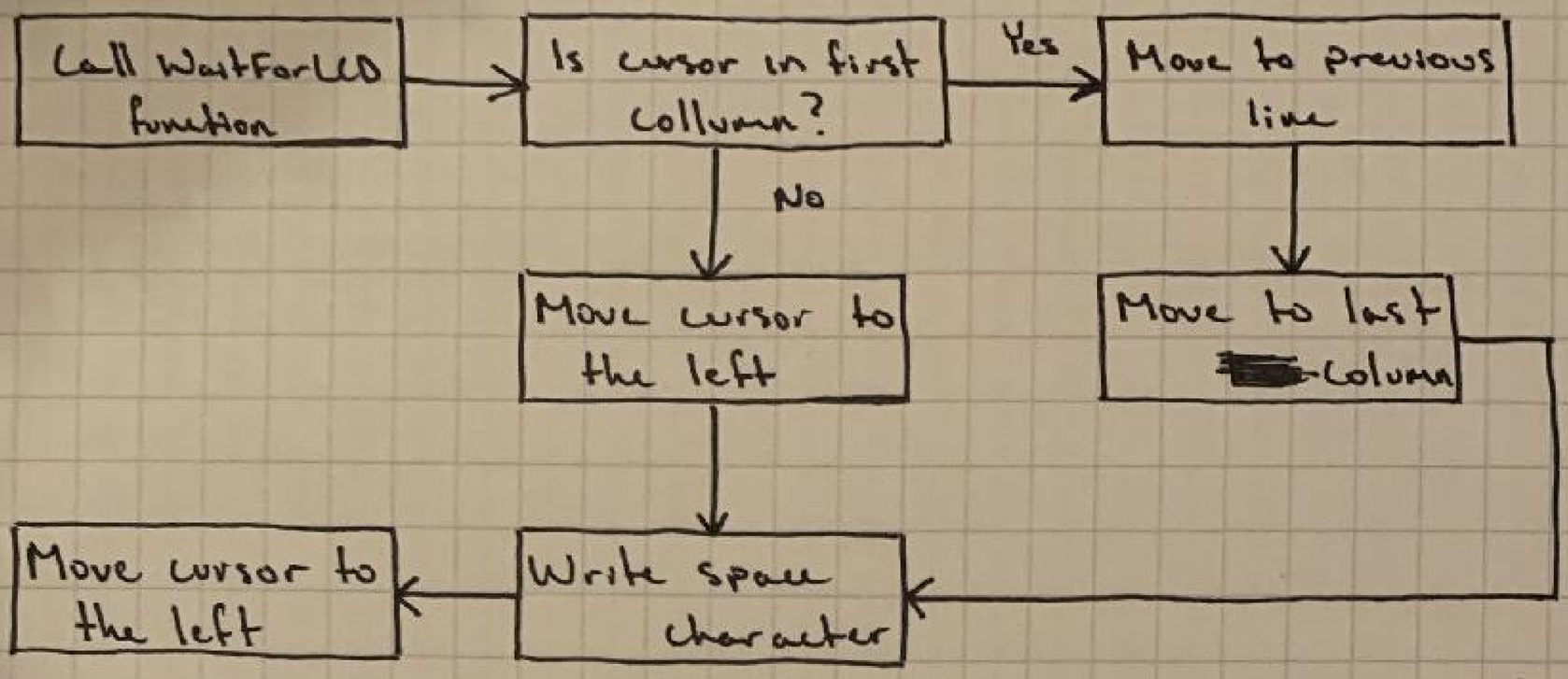


Figure 32-1: Flowchart describing logic for backspace function.

We will begin by calling a wait function to make sure the LCD is ready. After this, we will test which column the cursor is in. This is so that we know how to handle where to place the cursor. After this, we will move the cursor to the previous position, write a space, and move the cursor again.

With all this in mind, we will begin design for the function.

Our final code for the function is shown below:

```
// Function to move the cursor to the previous position and delete the
// character in that space
void BackspaceLCD(void)
{
    waitForLCD();                           // wait till LCD is ready
    uint8_t index;
    // special handling for backspace in first column of any row
    if ((ReadByteFromLCD(COMMAND_REGISTER) & 0x0F) == 0x00)
    {
        for(index = 0; index<3; index++)
            NewlineLCD();
        for(index = 0; index<16; index++)
        {
            waitForLCD();
            WriteByteToLCD(COMMAND_REGISTER, cursorRight);  //move cursor to the right
        }
        WriteByteToLCD(DATA_REGISTER, ' ');                 // write a space character
        WriteByteToLCD(COMMAND_REGISTER, cursorLeft);       // move cursor to that space
    }
    WriteByteToLCD(COMMAND_REGISTER, cursorLeft);
    WriteByteToLCD(DATA_REGISTER, ' ');
    WriteByteToLCD(COMMAND_REGISTER, cursorLeft);
}
```

Figure 32-2: C code for function to perform a backspace operation on the LCD.

11/13/18
3:30PM

The code follows the logic outlined in the flowchart. We begin with a wait so that the LCD is ready. After this, we test to see if the cursor is in the first column of the screen. If it is, we move to the last column of the previous row. If not, we move the cursor left. Both

Grant Ahrens
Brandon Lampert
11/13/18

Laboratory #5: Liquid Crystal Display Interface: Part 3

rows of the comparison end up end up writing a space and moving the cursor left.

Our method of testing this function consisted of filling the LCD screen with characters until the cursor was in the last position of the last line. After this, within our while loop in our main function, we called BackspaceLCD continuously. To see the results easier, we followed this with a 50 ms delay.

When we ran this test, we saw the cursor move across the screen, eventually erasing all of the characters that had been on the display. This is how we knew our function was working properly.

4. In order to write a function to move to a desired position on the display, we will simply need to design the logic from the flowchart below in c code:

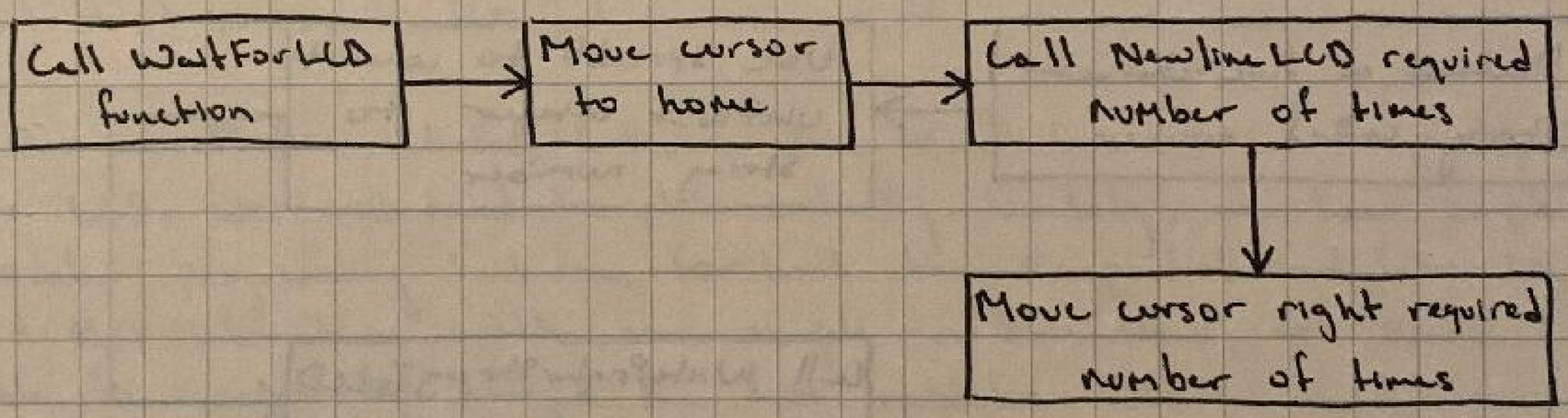


Figure 33-1: Flowchart describing logic to move cursor to desired location.

We will of course begin with a call to the wait function in order to ensure that the LCD is not busy. After this, we will return the cursor to the home position so that we know where it will be. Yes, this is not the most efficient method of knowing the cursor's position, but this way we eliminate having to move left when we move to the desired x-coordinate. This is because we know the cursor will be all the way to the left already. Finally we will shift the cursor right to the x-coordinate desired. Our code for this function is shown below:

```
// Function to place the cursor to a specific coordinate on the LCD
void MoveCursorToRowColumn(uint8_t row, uint8_t column)
{
    waitForLCD();                                   // wait till LCD is ready
    uint8_t index;
    WriteByteToLCD(COMMAND_REGISTER, 0x02);         // return home
    // move to desired row
    for (index = 0; index<(row-1); index++)
        NewlineLCD();
    // move to desired column
    for (index = 0; index<(column-1); index++)
        WriteByteToLCD(COMMAND_REGISTER, cursorRight);
}
```

Figure 33-2: Function to move cursor to a desired row and column.

DA
11/12/18
4:40PM

Grant Abelle
Brandon Langwell
11/15/19

Laboratory #5: Liquid Crystal Display Interface: Part 3

We did not encounter any problems when designing this function. Our method for testing consisted of making calls to the function with several different sets of coordinates specified. We had the program hold the cursor in each position for ten seconds so that we could count the spaces on the screen and ensure that the correct location ~~or~~ change was made.

After we knew that the function worked as intended, we were able to move on to the next part of this laboratory.

5. There are several different ways we can design a function to write a 16-bit integer to the screen. The difference in methods lies in the conversion of unsigned int to string. Given this, we have opted to use an already existing function to perform this conversion. The string.h header file contains the sprintf function, which will perform the conversion. A flowchart describing the full logic for this function is shown below:

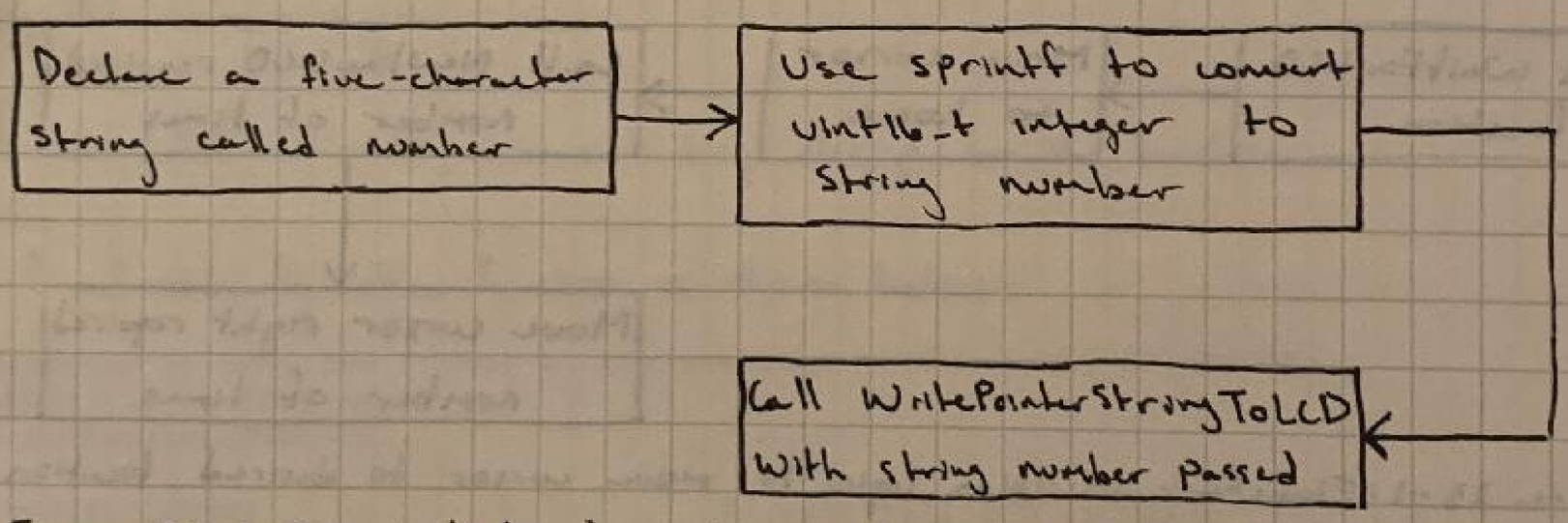


Figure 34-1: Flowchart for logic to print a 16-bit integer to the LCD.

With this, we are ready to begin writing our function.

The final version of our function is shown below:

```
// Function to write a 16-bit integer to the LCD
void WriteIntegerToLCD(uint16_t integer)
{
    char number[5];                          // max uint16_t is 65535
    sprintf(number, "%u", integer);          // convert integer to string
    WritePointerStringToLCD(number);         // print the string
}
```

Figure 34-2: C code implementation of function to write integer to the LCD.

11/15/18 2:40 PM — We came across one major problem when writing and when testing this function. Basically, when we would pass a number to this function that was below 32,767, the number would display perfectly fine and we were pleased with the results.

Laboratory #5: Liquid Crystal Display Interface: Part 3

However, when passing numbers above this value, the LCD would display negative values that were less than the value we had passed to the function.

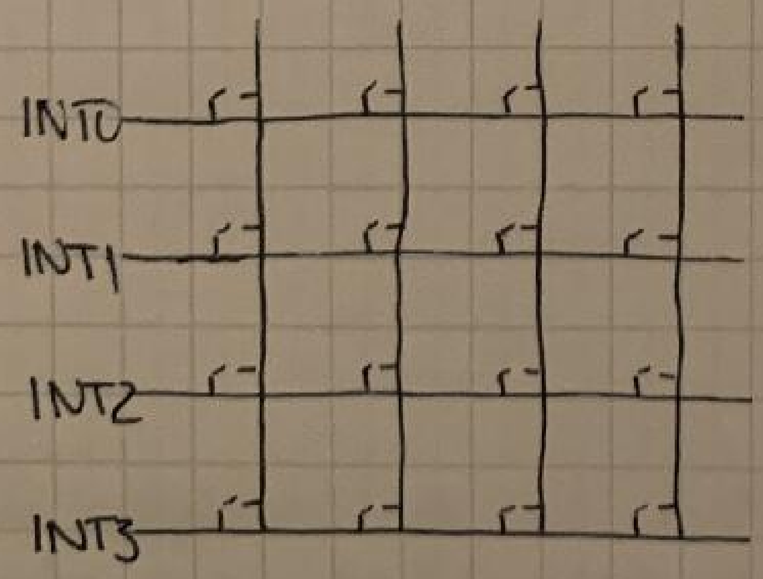
Upon further analysis of the testing and results, we realized that the threshold value of 32767 at which the results began to become undesirable, was half of the supposed capacity of an unsigned 16-bit integer. It was at this point that we realized the flag input into the sprintf function was a %d notating a signed integer. We quickly changed this flag to %u for an unsigned integer and retested the function.

We were pleased to find that numbers above the original threshold value displayed correctly, and after more testing we deemed the function fully operational.

With this accomplished, we ~~were~~ are ready to move on to the next and final portion of this lab.

6. For an effective interface between the Atmel board, a keypad, and the LCD, we will need to plan critically. Luckily, we have employed modular programming techniques for both the keypad-based laboratory and this laboratory. With some simple modifications, we should be able to successfully merge the two modules into a single program.

A circuit diagram for the keypad's interface with the board is shown below:



11/15/18
3:27PM

Figure 35-1: Circuit diagram of the external keypad using for external interrupts.

As is apparent in the above diagram, this set up will utilize four of the microcontroller's external interrupts INT0-INT3. This means that our program will need to contain four ISRs.

Grant Abella
Brandon Lapert
11/15/18

Laboratory #5: Liquid Crystal Display Interface: Part 3

Aside from this, we should not need many other major modifications to our existing code in this laboratory.

A basic high-level flowchart for this logic is shown below:

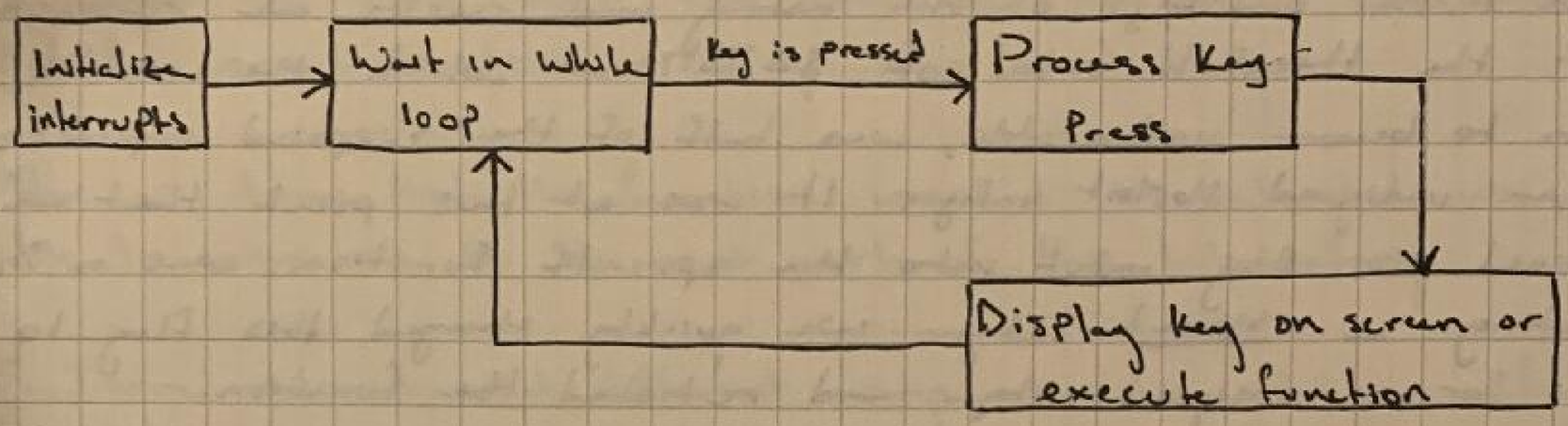


Figure 36-1: Flowchart for keypad-LCD logic.

With this logic outlined, we are ready to begin working on this program.

Our code for this functionality is shown below and extends to the next page. An explanation of the code will follow.

```
#include "lcd.h"
void interupts(void)
{
    EICRA = 0xAA; //int0-int3 falling edge
    EIMSK = 0x0F; //int0-int3 enable
}
ISR (INT0_vect) // ISR for handling 1st row
{
    char keyrow1[4] = {0x31, 0x32, 0x33,
0x41};
    char *keyOut = NULL;
    uint8_t temp = 0x00;
    for (uint8_t i = 0x00; i <= 0x03; i++)
    {
        PORTC = ~(0x01 <<i);
        _delay_us(10);
        temp = PINC;
        if (temp == 0xEF-(0x01<<i))
        {
            keyOut = keyrow1+i;
            i=0x04; //exits for loop
        }
    }
    PORTC = 0xF0;
    LCDKeypad(keyOut);
    while(1){
        temp = PINC;
        if ((temp&BV(4)) == BV(4))
        {
            _delay_ms(10);
            break;
        }
    }
}
```

```
ISR (INT1_vect) // ISR for handling 2nd row
{
    char keyrow2[4] = {0x34, 0x35, 0x36,
0x42};
    char *keyOut = NULL;
    uint8_t temp = 0x00;
    for (uint8_t i = 0x00; i <= 0x3; i++)
    {
        PORTC = ~(0x01 <<i);
        _delay_us(10);
        temp = PINC;
        if (temp == 0xDF-(0x01<<i))
        {
            keyOut = keyrow2+i;
            i=0x04; //exits for loop
        }
    }
    PORTC = 0xF0;
    LCDKeypad(keyOut);
    while(1){
        temp = PINC;
        if ((temp&BV(5)) == BV(5))
        {
            _delay_ms(10);
            break;
        }
    }
}
```

Figure 36-2: ISRs for INT0-1 for handling keypad input from the user.

The rest of our code is shown on the next page.

11/15/18
3:50 PM

Grant Abelke
Brandon Lampert
11/15/18

Laboratory #5: Liquid Crystal Display Interface: Part 3

```
ISR (INT2_vect) // ISR for handling 3rd row
{
    char keyrow3[4] = {0x37, 0x38, 0x39,
0x43};
    char *keyOut = NULL;
    uint8_t temp = 0x00;
    for (uint8_t i = 0x00; i <= 0x03; i++)
    {
        PORTC = ~(0x01 <<i);
        _delay_us(10);
        temp = PINC;
        if (temp == 0xBF-(0x01<<i))
        {
            keyOut = keyrow3+i;
            i=0x04; //exits for loop
        }
    }
    PORTC = 0xF0;
    LCDKeypad(keyOut);
    while(1){
        temp = PINC;
        if ((temp&BV(6)) == BV(6))
        {
            _delay_ms(10);
            break;
        }
    }
}
```

```
ISR (INT3_vect) // ISR for handling 4th row
{
    char keyrow4[4] = {0x46, 0x30, 0x45,
0x44};
    char *keyOut = NULL;
    uint8_t temp = 0x00;
    for (uint8_t i = 0; i <= 3; i++)
    {
        PORTC = ~(0x01 <<i);
        _delay_us(10);
        temp = PINC;
        if (temp == 0x7F-(0x01<<i))
        {
            keyOut = keyrow4+i;
            i=4; //exits for loop
        }
    }
    PORTC = 0xF0;
    LCDKeypad(keyOut);
    while(1){
        temp = PINC;
        if ((temp&BV(7)) == BV(7))
        {
            _delay_ms(10);
            break;
        }
    }
}
```

Figure 37-1: ISRs for INT2-INT3 for handling keypad input from the user.

```
// Function to handle writing character pressed on the keypad
// onto the LCD and and/or any function calls for buttons B,C,&E
void LCDKeypad(char *key)
{
    // Print numberic characters
    if ((*key >= '0') && (*key <='9'))
        WriteCharacterToLCD(*key);
    // Execute the backspace
    if (*key == 'B')
        BackspaceLCD();
    // Execute the clear
    if (*key == 'C')
        ClearLCD();
    // Execute newline
    if (*key == 'E')
        NewlineLCD();
}
```

Figure 37-2: code for function to handle writing keys ~~[illegible]~~ to the LCD or executing functions.

The contents of each ISR remained fairly unchanged from what they contained from the previous laboratory with a couple of ~~[illegible]~~ updates. We changed the row key array values from hex to ASCII to make them compatible with the LCD. In addition to this, we have a character pointer keyOut which is essentially the key that is pressed. This is passed in a call to the function shown in figure 37-2, which parses the character pointer.

GA
11/15/18
4:40 PM

Any numeric characters are sent to the WriteCharacterToLCD function to be displayed on the screen. The 'B' character leads to a call to the BackspaceLCD function. The 'C' character leads to a

Grant Moller
Brandon Lapert
11/15/18

Laboratory #5: Liquid Crystal Display Interface Part 3

call to the ClearLCD function. And finally, the 'E' character leads to a call to the newlineLCD function.

The last change made ~~to~~ to the ISRs was a short debounce block of code. This code block sits in a while loop until the value for keypad input has reached a stable value.

The program sits in a while loop and executes appropriately when keys are pressed. When we tested it initially we had some ~~[illegible]~~ bouncing issues, but after changing and testing our debounce code, we settled on the final version shown in this laboratory book.

+5

Postlab:

1. Our design approach in this lab was the same as in the previous parts of this lab. We began by thinking critically about the task at hand. After this, we would plan our design using flowcharts and numbered lists. Finally, we would write our code based on these plans and test it's performance thoroughly. Debugging utilized the onboard LEDs to display various important values.

2. While we did not encounter any hardware issues during this lab, we did have several software problems. These have both been discussed previously so I will not go into detail. The first issue was related to the NewlineLCD function in which we needed to adjust the cursor address. And this was fixed by simply thinking more critically about the cursor's current address and adjust based on that. The other big issue we had was our print integer function not printing correctly when the integer was over a certain value. This was fixed by changing the variable flag from %d to %u to denote unsigned behavior.

Conclusion: +2

Throughout this lab (all three parts), we developped our debugging and troubleshooting skills. In addition to this, we gained experience working with LCD's, which will be beneficial to us in the real world. Finally, we expanded our skills in building modular programs and even modular functions.

[signature]
11/15/18
5:23 PM

Code : -0 13/13
Discussion: -0 7/7 +7 27/20

p1: 19/10
p2: 17/10
p3: 27/20

63/40 [signature]

X5 [illegible] online