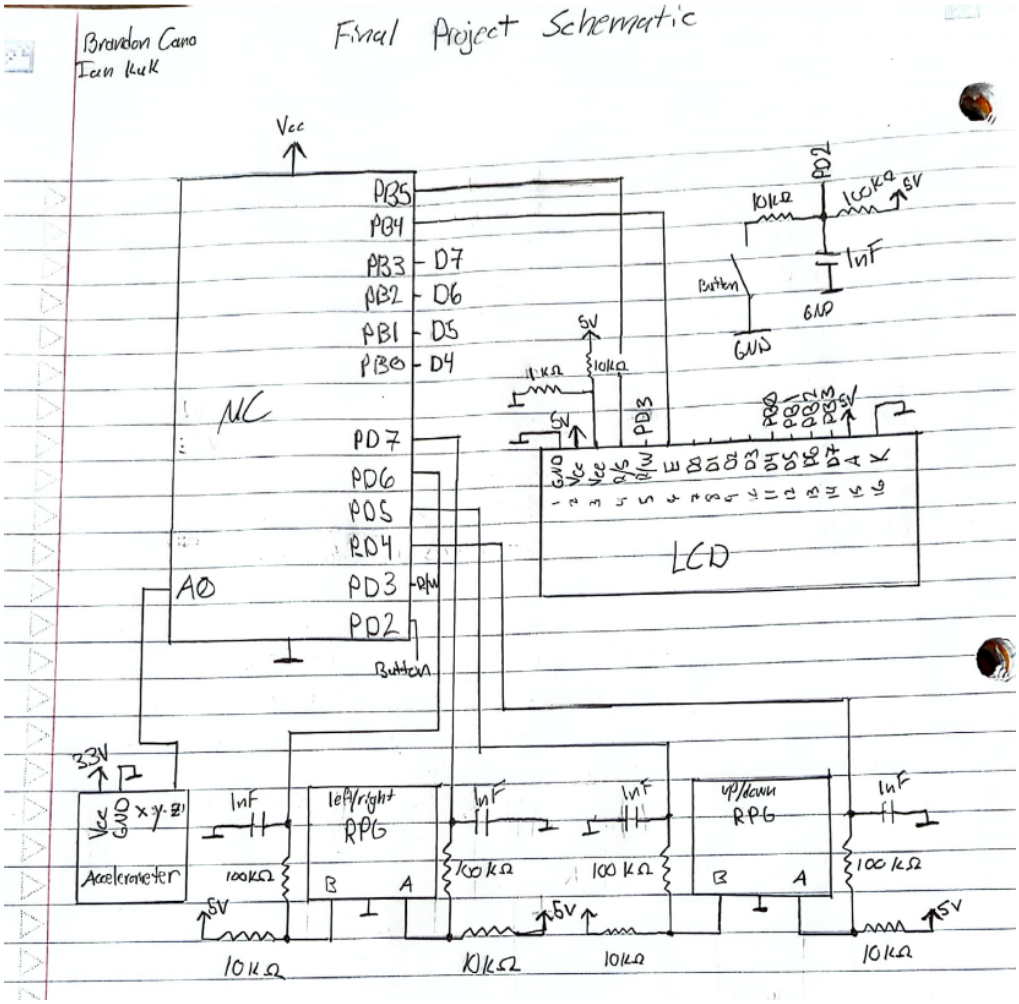
Project Title: Digital “Etch a Sketch”
Authors: Brandon Cano, Ian Kuk
Course Name: ECE:3360 Embedded Systems - Professor Beichel
Post Project Report

1. Introduction

For our project, we decided to create an "Etch-a-Sketch" inspired device using five main components: two rotary pulse generators (RPGs), a button, an LCD, and an accelerometer. The left RPG controlled horizontal direction and the right RPG controlled vertical direction for drawing on the LCD. The button was used to save the drawing on the screen in case of power loss, while the accelerometer detected shaking to clear the LCD.

2. Implementation

To make this project we had a few different items we had to implement and test to ensure we had a working product. For the sake of this report we will split off into two different sections, the hardware and the software. The schematic for the project is drawn below. Very similar to previous labs for the button and RPG setups, the same resistors and capacitors were utilized.



For the button, a physical debounce was used. The oscilloscope measured a debounce period of twenty nanoseconds so we used a 100kΩ pull up resistor and a 1nF capacitor. This roughly creates a period of 100 ns of debounce. We choose these values for the components because it's an extremely fast time to humans, and it more than ensures there will be no unnecessary bouncing. We had the wire from the button wired to PD2.

To debounce the RPGs we looked to the data sheet where it recommended a 10kΩ pull up resistor with a .01µF capacitor. We encountered a problem of not having a capacitor at that capacitance. To counteract this problem we used a 1nF capacitor in place and replaced the pullup resistor with a 100kΩ resistor instead. By decreasing the capacitor and increasing the resistor by the same magnitude we were able to have the factory recommended debounce solution. The wires for the RPGs used PD4 through PD7

The LCD setup was relatively simple. To start, pins one and five were wired to ground. Pin one was wired to ground because it's the LCD's ground pin, but pin five was wired to ground because it decides if the user is reading or writing to the LCD, zero being writing and one being reading. In this lab we had no use reading from the LCD so it was wired directly to ground so it would be permanently in write mode. Pin two was wired directly to VCC, and pin three was wired to a 10k resistor that connected to VCC and a 1k resistor that connected to ground. This pin is in charge of the contrast of the LCD so this setup allows the user to see the characters of the LCD better. Pin four, which is in charge of instructions, and register select was wired to PB5. Pin six, which controls the enable feature that allows information to be sent to the LCD, was wired to PB3. Pins seven through ten were left untouched, this was done because we only used the LCD in 4-bit mode and in 4-bit mode only pins eleven through fourteen are necessary. The pins were left unwired and not wired to ground, because wiring them to ground could cause more harm by accidentally sending a signal, whereas if left unwired there is no possibility of a false signal being sent. Pins eleven through fourteen were wired to PB0 through PB3 respectively. And finally pin fifteen was wired to VCC and pin sixteen was wired to ground. This was done in order to turn on the backlight of the LCD.

The accelerometer setup was extremely simple. Pin 1 of the accelerometer was wired to 3.3 volts as specified in the data sheet, and pin 2 was wired to ground. We decided to only wire pin 5, and that went to PC0. The reasoning for deciding PC0 was that it is an analog input, and the accelerometer generates analog values. We decided to only wire pin 5 of the accelerometer because it reads the Z axis. For our project we decided that any significant shaking that would mirror an actual "Etch-a-sketch" would only be shaking up and down, so the x and y axis were inconsequential.

For the software, I will explain each item as it was implemented into the project, starting with the uart_io. We implemented uart_io so that we can use 'printf' into the serial monitor for initial testing of our RPGs and accelerometer. In order to efficiently test different components with their respective code segments, we used the serial monitor to see how each item was working, to do this we had to enable the I/O for UART. Similar to lab 5 we create the uart_putchar and uart_getchar functions in order for printf to be seen from the serial monitor, along with the initial setup from the main function. These items are no longer needed with the current implementation of the project, but were left in to show how we went about testing through the implementation process.

Next we set up the code for RPG detection. In general, we decided against using interrupts as the sequence detection was a lot less reliable as we found out in lab 4. To do this, we took our assembly code from lab 2 and converted it to C which had the exact same behavior as before. How this works is that we first read in the 8 bit value from PIND and use a bitwise-and operation to only check the 2 left most bits since those are the only ones we care for, and it is where the first RPG is hooked up to on the arduino. Then we left shift to make them the 2 rightmost bits, then we check to see if it is different from the previously stored value. If not, we skip and don't do anything. If the bits are different we shift the bits to the left by 2, and bitwise-or the old and new values. This will continue until either the clockwise or counterclockwise patterns are achieved, and then the correct move function is called. The same idea is applied to the second RPG as well, with both being in the infinite loop.

Next we added in the code to read the value from the accelerometer. To do this we created a function called 'check_accel' which will also call another function called 'read_accel'. To read the value in, we had to set up

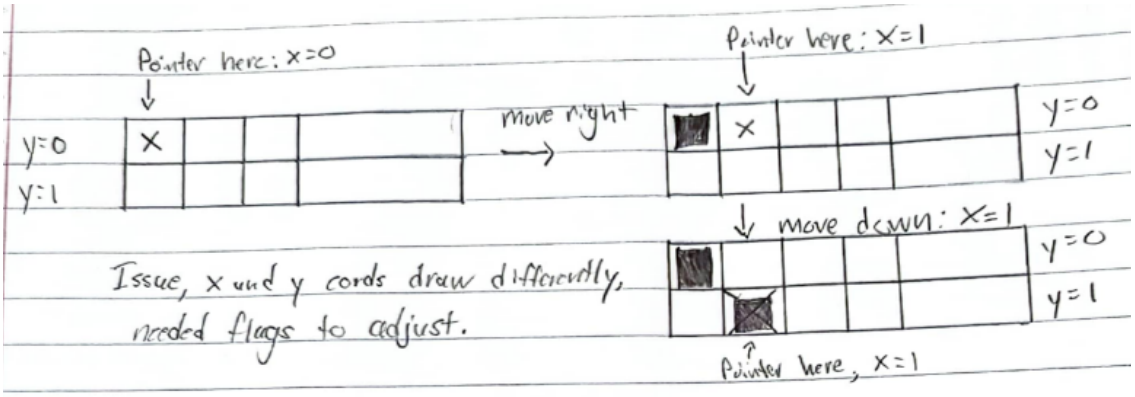
our ADC input to be able to read in the analog inputs. Once that was set up we checked the pin the accelerometer was wired to and read the value converting it to 8-bit mode. When the value has been converted we then check to see the magnitude of the value, and if it is high enough to be considered a 'shake' we want to actually reset the screen. After some testing in the serial monitor we gathered the average resting value to be roughly 1.800. Using this as the benchmark, we then tested with different values, settling on a value of 2.250 or greater to reset the display. This value will require the user to give a good shake and stop any movements across the table from being able to reset.

The next item we needed to get working was the LCD, to do this we utilized a library that had support for the HD44780 controller that is in our LCD. In this library there were the hd44780.h, lcd.h, and defines.h header files along with the hd44780.c and lcd.c, C code files. The first thing we had to do was configure the pins that we used on the LCD, and frequency of the microcontroller. The pins we configured were the R/S, R/W, E, and D4 pins. After setting these definitions, we had to call the lcd_init() function for the initial setup. We had to use the I/O stream for writing to the LCD so that we can call fprintf() for writing full strings. If that method is not used we can call the lcd_putchar method to write an individual character, which is used a lot for writing the different characters we had, like the full 5x8 block at 0xFF.

Though we ended with a 16x4 drawing board we started with the simpler 16x2 to help get us proof of concept and have some logic that we could then expand upon. Since not everything used here was used later on only the parts that were utilized in both will be discussed here and the changes later on. The first thing we did was define our coordinate system for the display, x (0-15) and y (0-1), this came directly from the size of the LCD display. For all four moveable directions we had a function for it, move_{left/right/up/down}. Then we added two char arrays to act as boards, both of size 16. This was done because the LCD has a 24 character gap between the end of the first line and the start of the second line, so it was easier to have two separate arrays that were determined by the y-coordinate. For both the left and right movements, we checked their respective bounds making sure that the user couldn't draw "off the screen". For the up and down movements we simply checked if the y-value was 0 or 1 to determine whether to edit to top or the bottom of the LCD.

Before moving to the next section of a more "viable" board to draw on, we needed to add some custom characters that we could write to the LCD. Since each block is a 5x8 rectangle we had to define some 5x4 half blocks. To do this we made some char arrays that had 8, 5-bit binary values indicating which pixels should be on (1) or off (0). To add the characters to the LCD memory, we had to add a function to our basic LCD library that we were using. We created a function called 'lcd_build_char' which took a location, from 0-7 (LCD only allows up to 8 custom characters), and the char array. We then write the command 0x40+(location*8), this will indicate where in the memory to write the characters to, then we plug in all 8 binary values from the char array, then exit the writing mode. We then call this function during the initialization of the program.

Now for the final drawing implementation. To start, the x and y coordinate system, two boards, and direction functions were all kept but had some modifications. Instead of switching the y-range to 0-3 we added another way to detect which level it should be at by using a char variable which holds either 0x00 or 0x01, these are the custom half blocks that we had to make and upload to the LCD. We also had to add flags to detect which movement we had previously. This was done in case we switch which axis we are moving on, we are able to correct the x-cord issue we had when designing our system, which is that after we write to the spot we change the x-value. Shown in the image below was our original problem with vertical movement after we moved either right or left. This is why the flags came into play, so we could avoid this diagonal direction that was wrong. Since this issue was found after the fact, going back and rewriting the logic would have been too much work so we added flags to adjust.



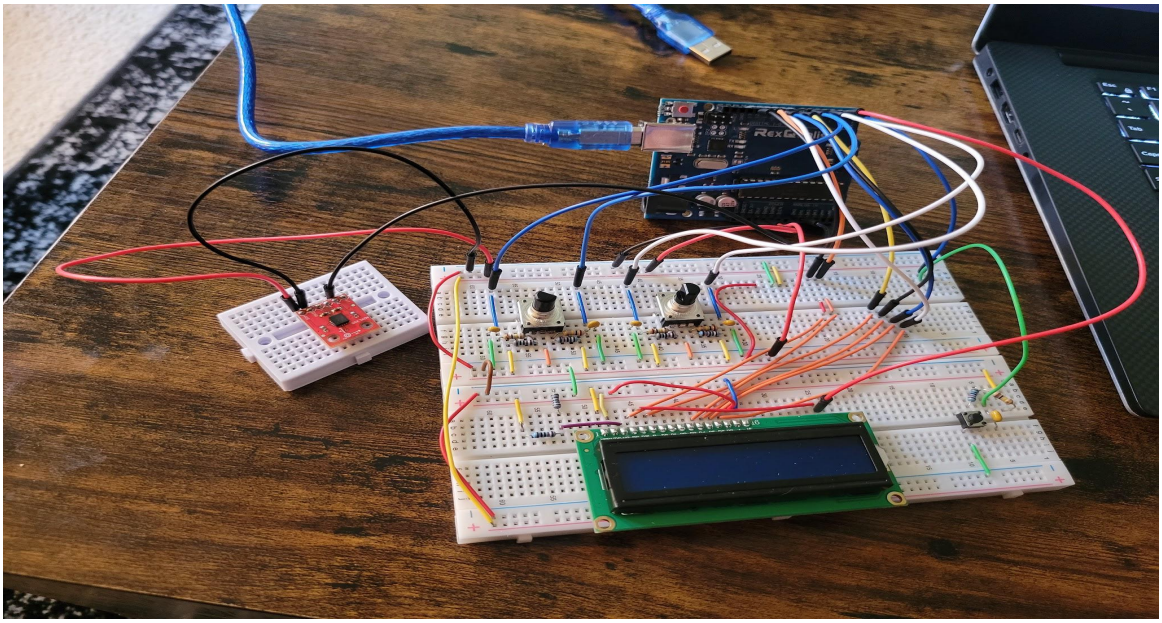
Following this, we have to know which of the 3 options to draw each block to be, top half, bottom half, or full block. This inherently places us at an issue of tracking every possible case for moving up and down, since left and right is determined by the 'block' variable. To help break this down, there is an image below showing all cases for moving down, moving up is mirrored horizontally. The table below shows where the pointer (' ') is at for the vertical motions, this one is specifically for the downward motion. This had the relevant information of what the different cases that could occur and what actions should follow.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0-0x00	' '	'T'	' '	'F'												
0-0x01			B	F	' '	' '	' '	' '								
1-0x00						T		F	' '	'T'	' '	'F'		T		F
1-0x01							B	F			B	F	' '	' '	'B'	'F'

We started by breaking the conditionals into the two different states of y, 0 and 1. From that, we checked the board values to help us understand where we want to draw and what our surroundings are, so that we know if there was no change at all, or which half block or full block to draw. At the end of each directional function we set the flags. If we moved up, then that flag is set to 1, while the moved_down is set to 0 (vice versa) and moved_right is set to -1, meaning we don't care about this currently. If we move right or left then we set the moved_right flag to 1 or 0 respectively and the up/down flags are set to -1, since we don't care about these at the current state.

The last item we added was a way to save the drawing the user had created. The way to do this was by writing and reading to the EEPROM memory on the microcontroller. In order to do this, we utilized a library included in the AVR directory, <avr/eeprom.h>. We utilized the read and write/update functions that were provided. At the start of the program, we called our function 'load_board' which has an uint8_t array of length 32, since it's a 16x2 LCD screen. We use the function eeprom_read_block(), the first parameter is the array that we load the data into, the second is which address to start from, we choose 12, and the last is the size or amount of items to read from (32). After we load in the data, we use a for loop to place it into both board arrays that we have. To save the strings to EEPROM we have a similar function called 'save_board'. Essentially, we reverse the process that we did with loading, we write the data from the board arrays into the one uint8_t array and then use eeprom_update_block() to then write the data to EEPROM. We used update instead of write because update will first check the contents of the memory before writing to it, if it's the same then it won't waste a write, which helps give it a longer lifespan. We also chose block instead of byte or word because we had a fair bit of data to be loaded and written to so it was easier on the constraints. The last part of this function checks to see if the board is being

cleared or not, if it is then we display a message denoting that the user saved it, if they clear the board then we don't display the message but save the cleared board.

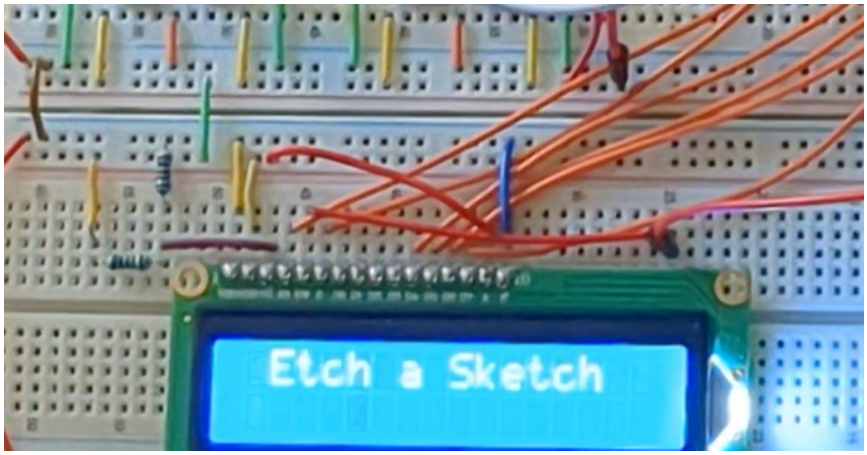


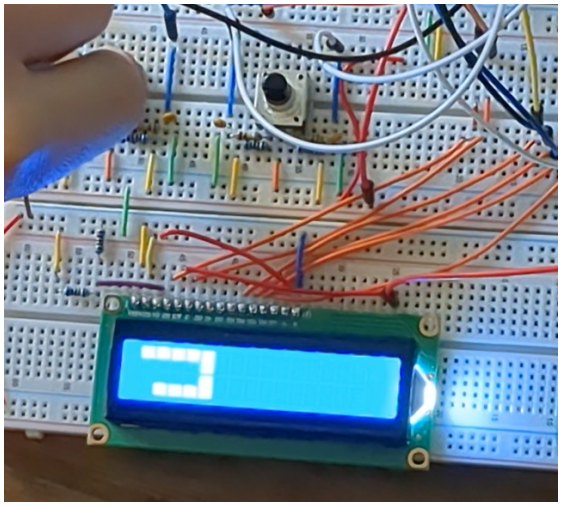
3. Experimental Methods

While we were developing the project we had many different ways to test each component and pieces of code to ensure proper behavior and to find bugs. The first items we tested were the RPGs and accelerometer, to do this we used what we learned in lab 5 to print items to the Arduino serial monitor and test if the accelerometer function will print a value once a value of 2 or greater was achieved. Same idea was applied to the RPGs, for both clockwise and counterclockwise motions we printed out strings to see if the logic was working appropriately. Another big method of testing the drawing mechanic was starting with the easier 16x2 version and getting a base idea how the logic works for that, before expanding to the more complex 16x4 iteration.

4. Results

Here are some images showing the project working, for a more in depth demo refer to the demo video submitted on icon.





5. Discussion of Results

WIth the current implementation the device performs fairly well. The custom characters we uploaded to the LCD work well, but don't allow for many creative drawings. Saving and loading from EEPROM memory works as intended from the push of a button. Drawing to the screen works, but there are small inconsistent glitches that will show themselves every so often. We believe these could be from accidentally hitting wires while using the RPGS, or RPGs not always detecting movement or detecting an incorrect movement. Clearing the LCD works without any issues. It has been tuned so there are no accidental resets, and that it needs substantial vertical movement in the z-direction to detect a reset.

Initially we wanted to use a graphical LCD to better simulate the “Etch-a-Sketch” machine, but there were many issues that we ran into with libraries not working and lack of information relating to the controllers of our GLCD. Due to these issues we had to make a change with design goals. We switched back to the basic LCD we used in lab 4. To make up for the lack of difficulty we created and uploaded custom characters to the LCD, and utilized EEPROM memory on the microcontroller to save what the user has on the screen from a push of a button. These revised goals were all met and working as intended.

Some alternative implementations we could have used would have been a larger sized LCD, whether that's a graphical or a OLED screen just to add some more space to draw on. Potentially the logic to draw could have been done in a more elegant way, the movement functions were all a little clunky and had small glitches. Also remembering where the cursor was could with the EEPROM memory could have been another item to consider. This also leads into some current limitations as well since there is not a lot of room for drawing since it's a 16x4 board. With the current LCD it also isn’t possible to have a blinking cursor for what we would need it for, so it can be hard to tell where you are currently located on the board.

Some potential ideas for improvement in the future could be to use a graphical LCD screen which has more flexibility with drawing around and could make it similar to an actual “Etch-a-Sketch” machine. We could also add extensions to the RPGs, or use different parts so that it is easier to draw, and more similar to an actual “Etch-a-Sketch”. Another change could be having some sort of cursor that can help the user know exactly where they are if they start drawing through repeated areas.

6. Conclusion

In the end we were able to get a fully working product that allows users to draw, erase, and save images onto an LCD. We learned how to create and write custom characters to an LCD, how to read and write to EEPROM, how to read and use values from an accelerometer, and how to do all of the previously mentioned accomplishments in C.

7. References

Datasheets:

Accelerometer (datasheet under documents): https://www.sparkfun.com/products/9269

Software Libraries:

LCD Library: https://github.com/avrdudes/avr-libc/tree/main/doc/examples/stdiodemo

- defines.h
- hd44780.h
- hd44780.c
- lcd.h - *added our own function:* lcd_build_char
- lcd.c - *added our own function:* lcd_build_char

I2C Serial Communication Library (used for testing early on): https://github.com/alx741/avr_i2c

- twimaster.h
- I2cmaster.c

EEPROM Library: *<avr/eeprom.h>*

8. Appendix - Source Code

main.c

```
/*
 * Project.c
 *
 * Created: 4/23/2023 2:17:23 PM
 * Author : Brandon Cano, Ian Kuk
 */

#define F_CPU 16000000L
#define UART_BAUD 9600
#define DAC_ADDRESS         0b01011000
#define slave 0101100

#include <ctype.h>
#include <stdio.h>
#include <stdint.h>

#include <avr/eeprom.h>
#include <avr/io.h>
#include <util/delay.h>

#include "lcd.h"
#include "serial/twimaster.c"

int uart_putchar(char, FILE*);
int uart_getchar(FILE*);

static FILE uart_io = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);
static FILE lcd_str = FDEV_SETUP_STREAM(lcd_putchar, NULL, _FDEV_SETUP_WRITE);
```

```
static const char line_break[] = "                        ";

static const int x_max = 15; // 0 - 15

// global variables
int x = 0;
int y = 0;
int moved_right = -1;
int moved_down = -1;
int moved_up = -1;
char block = 0x00;
// 0x00 top half
// 0x01 bottom half
// 0xFF full block
char board_1[16] = { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' };
char board_2[16] = { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' };

// custom characters for LCD
unsigned char custom_char_1[8] = { 0b11111, 0b11111, 0b11111, 0b11111, 0b00000, 0b00000, 0b00000, 0b00000 };
unsigned char custom_char_2[8] = { 0b00000, 0b00000, 0b00000, 0b00000, 0b11111, 0b11111, 0b11111, 0b11111 };

// setup IO and
void main_init(void)
{
         stdout = stdin = &uart_io;
         stderr = &lcd_str;

         ADMUX |= 0b01 << REFS0;
         ADCSRA = 1 << ADEN | 0b110 << ADPS0;

         UCSR0A = 1 << U2X0;
         UBRR0L = (F_CPU / (8UL * UART_BAUD)) - 1;
         UCSR0B = 1 << TXEN0 | 1 << RXEN0;
}

// add custom characters to the LCD
void lcd_char_init(void)
{        // this function was created by us and added to the lcd.c file
         lcd_build_char(0, custom_char_1); // 0x00
         lcd_build_char(1, custom_char_2); // 0x01
         /*
         lcd_putchar(0x00, &lcd_str);
         lcd_putchar(0x01, &lcd_str);
         */
}

// load the last saved board from EEPROM to be displayed
void load_board(void)
{
         uint8_t board_from_eeprom[32];
         eeprom_read_block((void*)&board_from_eeprom, (const void*)12, 32);
         for (int i=0; i<16; i++)
         {
                  board_1[i] = board_from_eeprom[i];
                  board_2[i] = board_from_eeprom[i+16];
         }
}

// reads in the analog input from the accelerometer and converts to 8-bit
static float read_accel(uint8_t pin)
{
         ADMUX = (ADMUX & 0xf0) | pin;
         ADCSRA |= 1 << ADSC;             // conversion to 8 bit
         while (!(ADCSRA & (1 << ADIF))); // wait for conversion to complete
         ADCSRA |= 1 << ADIF;
         return 5.0f * (ADCL + (ADCH << 8)) / 1023.0f;
}
```

```
// main loop
int main(void)
{
	// initialize everything
	main_init();
	lcd_init();
	lcd_char_init();
	load_board();

	// starting screen
	fprintf(&lcd_str, " Etch a Sketch\n");
	_delay_ms(2000);

	// write board loaded from EEPROM
	write_board();

	// variables for input components
	uint8_t LR1;
	uint8_t LR2 = 0b00000011;
	uint8_t LR3 = 0b00000011;

	uint8_t UD1;
	uint8_t UD2 = 0b00000011;
	uint8_t UD3 = 0b00000011;

	uint8_t PB;

	// main loop
	while (1)
	{
		// check accelerometer
		check_accel();

		// left/right RPG
		LR1 = PIND;
		LR1 &= 0b11000000;
		LR1 >>= 6;
		if (LR1 != LR2)
		{
			LR2 = LR1;
			LR3 <<= 1;
			LR3 <<= 1;
			LR3 |= LR1;
			if (LR3 == 0b01001011)
			{   // clockwise pattern
				move_left();
			}
			else if (LR3 == 0b10000111)
			{   // counterclockwise pattern
				move_right();
			}
		}

		// up/down RPG
		UD1 = PIND;
		UD1 &= 0b00110000;
		UD1 >>= 4;
		if (UD1 != UD2)
		{
			UD2 = UD1;
			UD3 <<= 1;
			UD3 <<= 1;
			UD3 |= UD1;
			if (UD3 == 0b01001011)
			{   // clockwise pattern
				move_up();
			}
			else if (UD3 == 0b10000111)
```

```
                        {   // counterclockwise pattern
                                move_down();
                        }
                }

                // push button
                PB = PIND;
                PB &= 0b00000100;
                // check button press
                if (PB == 0b00000000)
                {       // wait for release
                        while (PB != 0b00000100)
                        {
                                PB = PIND;
                                PB &= 0b00000100;
                        }
                        // upon release we save the board
                        save_board(0);
                }
        }
}

// accelerometer clears the LCD screen
int check_accel(void)
{
        float z = read_accel(0);
        if (z > 2.000)
        {       // write clear screen flag
                lcd_putchar('\n', &lcd_str);
                // clear board arrays
                for (int i=0; i<16; i++)
                {
                        board_1[i] = ' ';
                        board_2[i] = ' ';
                }
                save_board(1);
                write_board();
        }
        return 0;
}

// same logic for both left and right motions to determine what to draw on the board
void write_left_right(void)
{
        if (y == 0)
        {
             if ((board_1[x] == 0x00 && block == 0x01) || (board_1[x] == 0x01 && block == 0x00) || (board_1[x] == 0xFF))
                {
                        board_1[x] = 0xFF;
                }
                else
                {
                        board_1[x] = block;
                }
        }
        if (y == 1)
        {
             if ((board_2[x] == 0x00 && block == 0x01) || (board_2[x] == 0x01 && block == 0x00) || (board_2[x] == 0xFF))
                {
                        board_2[x] = 0xFF;
                }
                else
                {
                        board_2[x] = block;
                }
        }
}
```

```
void move_left(void)
{	// fix x-cord
	if (moved_down == 1 || moved_up == 1)
	{
		x--;
	}
	// don't do anything when at lower bound
	if (x < 0)
	{
		x = 0;
		return;
	}
	// determine how to write to the board
	write_left_right();
	write_board();
	x--;
	// update flags
	moved_right = 0;
	moved_down = -1;
	moved_up = -1;
}

void move_right(void)
{	// fix x-cord
	if (moved_down == 1 || moved_up == 1)
	{
		x++;
	}
	// don't do anything when at upper bound
	if (x > x_max)
	{
		x = x_max;
		return;
	}
	// determine how to draw to board
	write_left_right();
	write_board();
	x++;
	// update flags
	moved_right = 1;
	moved_down = -1;
	moved_up = -1;
}

void move_up(void)
{	// fix x-cord
	if (moved_right == 1 && x != x_max)
	{
		x--;
	}
	else if (moved_right == 0 && x != 0)
	{
		x++;
	}
	// determine how to write to the board
	if (y == 1) {
		if (board_2[x] == ' ') {
			board_2[x] = 0x01;
			block = 0x01;
			y = 1;
		}
		else if (board_2[x] == 0x00 && (board_1[x] == ' ' || board_1[x] == 0x01)) {
			board_1[x] = 0x01;
			block = 0x01;
			y = 0;
		}
		else if (board_2[x] == 0x01) {
			board_2[x] = 0xFF;
```

```
                        block = 0x00;
                        y = 1;
                }
                else if (board_2[x] == 0xFF && (board_1[x] == ' ' || board_1[x] == 0x01)) {
                        board_1[x] = 0x01;
                        block = 0x01;
                        y = 0;
                }
                else if (board_1[x] == 0x00) {
                        board_1[x] = 0xFF;
                        block = 0x01;
                        y = 0;
                }
        }
        else if (y == 0) {
                if (board_1[x] == ' ') {
                        board_1[x] = 0x01;
                        block = 0x01;
                        y = 0;
                }
                else if (board_1[x] == 0x01) {
                        board_1[x] = 0xFF;
                        block = 0x00;
                        y = 0;
                }
                else if (board_1[x] == 0xFF) {
                        block = 0x00;
                        y = 0;
                }
        }

        write_board();
        moved_right = -1;
        moved_down = 0;
        moved_up = 1;
}

void move_down(void)
{       // fix x-cord
        if (moved_right == 1 && x != x_max)
        {
                x--;
        }
        else if (moved_right == 0 && x != 0)
        {
                x++;
        }
        // determine how to write to the board
        if (y == 0) {
                if (board_1[x] == ' ') {
                        board_1[x] = 0x00;
                        block = 0x00;
                        y = 0;
                }
                else if (board_1[x] == 0x01 && (board_2[x] == ' ' || board_2[x] == 0x00)) {
                        board_2[x] = 0x00;
                        block = 0x00;
                        y = 1;
                }
                else if (board_1[x] == 0x00) {
                        board_1[x] = 0xFF;
                        block = 0x01;
                        y = 0;
                }
                else if (board_1[x] == 0xFF && (board_2[x] == ' ' || board_2[x] == 0x00)) {
                        board_2[x] = 0x00;
                        block = 0x00;
                        y = 1;
```

```
                }
                else if (board_2[x] == 0x01) {
                        board_2[x] = 0xFF;
                        block = 0x00;
                        y = 1;
                }
        }
        else if (y == 1) {
                if (board_2[x] == ' ') {
                        board_2[x] = 0x00;
                        block = 0x00;
                        y = 1;
                }
                else if (board_2[x] == 0x00) {
                        board_2[x] = 0xFF;
                        block = 0x01;
                        y = 1;
                }
                else if (board_2[x] == 0xFF) {
                        block = 0x01;
                        y = 1;
                }
        }

        write_board();
        moved_right = -1;
        moved_down = 1;
        moved_up = 0;
}

void write_board(void)
{
        for (int i=0; i<16; i++)
        {
                lcd_putchar(board_1[i], &lcd_str);
        }
        fprintf(&lcd_str, line_break);
        for (int i=0; i<16; i++)
        {
                lcd_putchar(board_2[i], &lcd_str);
        }
        lcd_putchar('\n', &lcd_str);
}

void save_board(int is_cleared) {
        uint8_t eeprom_saved_board[32];
        for (int i=0; i<16; i++)
        {
                eeprom_saved_board[i] = board_1[i];
                eeprom_saved_board[i+16] = board_2[i];
        }
        eeprom_update_block((const void*)&eeprom_saved_board, (void*)12, sizeof(eeprom_saved_board));
        if (!is_cleared) {
                fprintf(&lcd_str, "     Saved!     \n");
                _delay_ms(2000);
        }
        write_board();
}

/** TEMPORARY FOR TESTING PURPOSES **/
int uart_putchar(char c, FILE *s)
{
        // CRLF insertion
        if (c == '\n')
        {
                uart_putchar('\r', s);
        }
        while (!(UCSR0A & (1 << UDRE0)));
```

```
        UDR0 = c;
        return 0;
}

int uart_getchar(FILE *s)
{
        // wait for data to be received
        while (!(UCSR0A & (1 << RXC0)));
        uint8_t c = UDR0;
        return c;
}
```

defines.h

```
/* CPU frequency */
#define F_CPU 16000000L

/* UART baud rate */
#define UART_BAUD  9600

/* HD44780 LCD port connections */
#define HD44780_RS B, 5
#define HD44780_RW D, 3
#define HD44780_E  B, 4
/* The data bits have to be not only in ascending order but also consecutive. */
#define HD44780_D4 B, 0

/* Whether to read the busy flag, or fall back to
   worst-time delays. */
#define USE_BUSY_BIT 1
```

lcd.h

```
/*
 * Initialize LCD controller.  Performs a software reset.
 */
void     lcd_init(void);

/*
 * Send one character to the LCD.
 */
int      lcd_putchar(char c, FILE *stream);

/*
 * Send a custom character to a location to be used on the LCD.
 */
void lcd_build_char(unsigned char loc, unsigned char *p);
```

lcd.c

```
#include "defines.h"

#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

#include <avr/io.h>

#include <util/delay.h>

#include "hd44780.h"
#include "lcd.h"
```

```
static FILE lcd_str = FDEV_SETUP_STREAM(lcd_putchar, NULL, _FDEV_SETUP_WRITE);

/*
 * Setup the LCD controller.  First, call the hardware initialization
 * function, then adjust the display attributes we want.
 */
void lcd_init(void)
{

  hd44780_init();

  /*
   * Clear the display.
   */
  hd44780_outcmd(HD44780_CLR);
  hd44780_wait_ready(true);

  /*
   * Entry mode: auto-increment address counter, no display shift in
   * effect.
   */
  hd44780_outcmd(HD44780_ENTMODE(1, 0));
  hd44780_wait_ready(false);

  /*
   * Enable display, activate non-blinking cursor.
   */
  hd44780_outcmd(HD44780_DISPCTL(1, 0, 0));
  hd44780_wait_ready(false);
}

/*
 * Send character c to the LCD display.  After a '\n' has been seen,
 * the next character will first clear the display.
 */
int lcd_putchar(char c, FILE *unused)
{
  static bool nl_seen;

  if (nl_seen && c != '\n')
    {
      /*
       * First character after newline, clear display and home cursor.
       */
      hd44780_wait_ready(false);
      hd44780_outcmd(HD44780_CLR);
      hd44780_wait_ready(false);
      hd44780_outcmd(HD44780_HOME);
      hd44780_wait_ready(true);
      hd44780_outcmd(HD44780_DDADDR(0));

      nl_seen = false;
    }
  if (c == '\n')
    {
      nl_seen = true;
    }
  else
    {
      hd44780_wait_ready(false);
      hd44780_outdata(c);
    }

  return 0;
}

void lcd_build_char(unsigned char loc, unsigned char *p)
{
```

```
        unsigned char i;
        if(loc<8) //If valid address
        {
                hd44780_wait_ready(false);
                hd44780_outcmd(0x40+(loc*8));
                for(i=0;i<8;i++) {
                        lcd_putchar(p[i], &lcd_str); //Write the character pattern to CGRAM
                }

        }
        hd44780_wait_ready(false);
        hd44780_outcmd(0x80);
}
```