INTRODUCTION

This lab concentrates on working with three LEDs, an 8x8 LED matrix, a thumbstick, and a speaker. The emphasis is on managing digital outputs and ensuring precise timing of multiple tasks. To accomplish this, we will manipulate hardware registers and bits without relying on existing libraries. Our first task involves setting up LEDs on specific pins and creating a sequence of flashing lights. Following this, we will create a task capable of generating a square wave at a predetermined frequency. Next, we will tackle concurrent tasks by defining and controlling three separate tasks. We will demonstrate how these tasks can operate simultaneously and execute in a specific sequence.

We will then implement an interactive LED display matrix that responds to thumb stick inputs. We will write a function to control a single LED at a specific row and column. Lastly, we will develop a sketch that can play a tune through the speaker while simultaneously allowing control of the LED matrix.

The main objective of this lab is to introduce direct register access and bit manipulation, laying the groundwork for hardware functions, task coordination, and interactive display creation using Arduino.

METHODS AND TECHNIQUES

In this lab, I applied several methods and techniques to achieve my objectives. For the manipulation of hardware registers and bits, I bypassed the use of existing libraries. This involved configuring three LEDs on specific pins and programming a sequence of flashing lights. I wrote code that directly accessed the hardware registers and bits, controlling the state of the LEDs. In the study of the 16-Bit Timer/Counter, I used the datasheet and lecture materials to understand the necessary registers and bits. I then wrote a task that set up the Timer/Counter to generate a square wave at a specific frequency.

For the concurrent tasks, I defined three separate tasks in the code. I used a round-robin scheduling approach in the Arduino loop() function to control the execution of these tasks. I used global variables as flags to signal the start and stop of each task. For the interactive LED display matrix, I wrote a function that directly manipulated the hardware registers and bits to control a single LED at a specific row and column. I connected a thumbstick to the Arduino and used the analogRead() function to read the X and Y values from the thumbstick.

Finally, for the sketch that plays a tune through the speaker while controlling the LED matrix, I created my own clock by using the delay() function in the loop() function. This allowed me to synchronize the playing of the tune with the control of the LED matrix based on the thumbstick input.

EXPERIMENTAL RESULTS

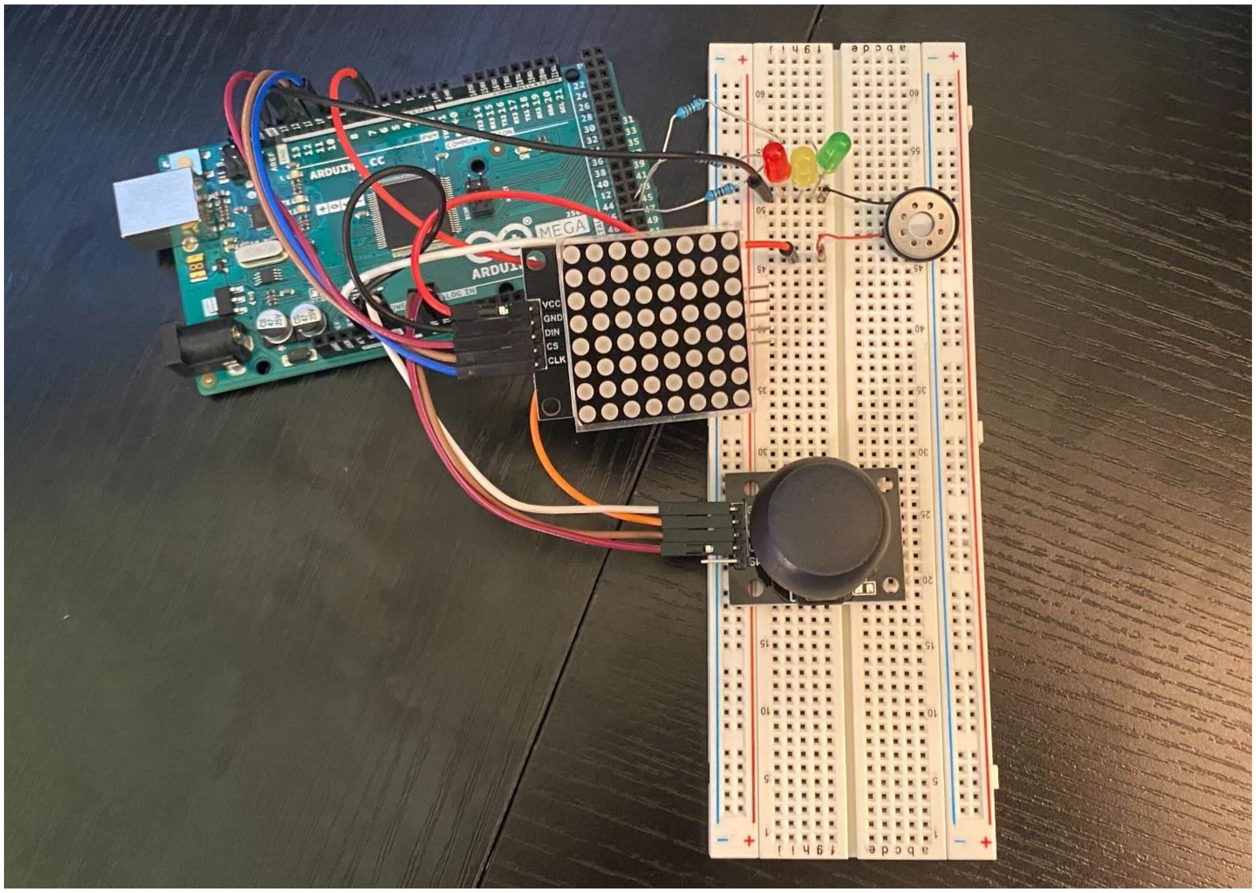

Figure 1 The experimental setup for this lab

Part 1.4

In the initial part of the lab, I began by initializing the required pins (47, 48, and 49) to enable them to output a signal. This was accomplished through direct register access, a method that allows for more precise control over the microcontroller's hardware. Following this, I utilized the void loop() function to control the state of each individual pin. By setting a specific LED to HIGH (on) and the previous one to LOW (off), I was able to create a sequence of LED flashes. To achieve a frequency of 1Hz for each LED and ensure a sequential toggle, I incorporated the delay() function into the code. This function pauses the program for a specified amount of time, allowing for control over the timing of the LED flashes. To ensure that the LEDs completed a full sequence in one second, I passed the value 333 to the delay() functions. This value was chosen because it divides one second into three equal parts, allowing each LED to flash for an equal duration within the one-second cycle.

Upon running this code, I observed the correct sequence of LED flashes. The timing was accurate, with one full sequence of flashes occurring exactly every second. This confirmed the successful implementation of the direct register access method and the correct use of the delay() function to control the timing of the LED flashes.

Part 2.4

In this part of the lab, I developed a task named generate_frequency(int frequency_in_Hz, int seconds), which accepts two arguments. The first argument specifies the frequency to be generated, and the second argument determines the duration, in seconds, for which the frequency should be generated.

To accurately generate the Pulse Width Modulation (PWM) signal needed to drive the speaker on pin OC4A (pin 6), I utilized the Timer/Counter 4 (TCTN4) and the Output Compare Register A (OCR4A) compare flag. The timer was set to Clear Timer on Compare Match (CTC) mode, which resets the timer each time a compare flag is raised. Given the frequency, I calculated a half period. This was necessary because in CTC mode, the state of the pin can only be changed on the compare, meaning we need two compares to complete a full period. I also implemented a time and counter variable to keep track of the duration each frequency is played for. This allowed for precise control over the duration of each tone.

Finally, in the loop() function, I called the generate_frequency() task for each frequency that needed to be played. This approach allowed for the generation of a sequence of tones at specified frequencies and durations. Upon running this code, I compared the generated tones with those of a provided example. The tones matched exactly, confirming that the code worked as intended.

Part 3.2

In this section of the lab, we were tasked with combining the first two parts to run the LED sequence while simultaneously playing a set of tones on the speaker. This required a different approach from the previous sections. Initially, I attempted to use TCTN4 to generate two PWM signals. However, I found this method overly complex. Instead, I opted to create a pseudo-timer, as demonstrated in class, using the delay() function in the loop() function to act as a synchronizer.

To ensure the tasks could run simultaneously, I modified both the LED sequence task and the generate_tone task to be non-blocking. This meant that these tasks could run independently of each other without causing the program to wait for one task to be completed before starting the other. I also developed a helper function that sets various global flags. These flags signal to the functions in the loop() function when it is appropriate to turn on and off. I used global variables to initialize multiple timers and counters, facilitating synchronization among all the tasks.

Upon compiling and running this code, I observed that the tones played correctly, and the LED sequence ran as intended, both when the tones were playing and when they were not.

Part 3.3

In this section of the lab, the task was similar to the previous one, but with an added requirement to play the tune "Mary Had a Little Lamb" while the LEDs flashed simultaneously. To implement this, the notes for the tune were provided in an integer array, melody[], and the frequencies of the notes were defined in the .ino file header.

However, the melody[] array contained significantly more elements than the frequency array used in the previous section. To accommodate this, with the assistance of ChatGPT, I modified the generate_frequency function to play an array of frequencies of any length. This allowed for the generation of a sequence of tones that matched the melody of "Mary Had a Little Lamb." The rest of the code remained unchanged. Upon compiling and running the code, the speaker played the familiar tune, which was easily recognizable.

LED Matrix Without Tone

In this section of the lab, we implemented code to control a dot on an 8x8 SPI LED matrix using a joystick. The joystick was connected to analog pins A0 and A1, which were used for x-axis and y-axis measurements respectively. These pins feature an Analog-to-Digital Converter (ADC), enabling them to interpret the analog signals from the joystick.

Starting with the provided helper code, I first modified it to refresh the matrix to an 'all off' state after the initialization functions were called in the setup() function. This ensured that the matrix was clear before we started controlling the dot. In the loop() function, I used the analogRead() function to read the analog signals from the joystick on pins A0 and A1. I then used the map() function to convert these values to a range of 0 through 8. This was done because the joystick lost sensitivity at the edges of its travel. By mapping to a range of 0 through 8 and then using an if statement to set any value of 8 to 7, I was able to increase the sensitivity at the edges of the joystick's travel.

Finally, I used the provided spiTransfer function to send this data to the LED dot matrix. This allowed us to control the position of the dot on the matrix using the joystick. Upon running this code, I was able to effectively control the dot on the matrix using the joystick. The behavior I observed matched that of the example video provided, confirming the successful implementation of the joystick control for the LED matrix.

LED Matrix With Tone

In the final part of the lab, we were tasked with playing the tune "Mary Had a Little Lamb" while simultaneously controlling the LED dot matrix using the joystick. To accomplish this, I reused the code from previous sections and applied the same timing scheme as in section 3.3. However, to control the dot matrix, I replaced the delay function used for refreshing with an if statement. This if statement checked whether the ms_counter had reached 50 before proceeding with the code to control the LED dot matrix. Because I was using a delay of 100 microseconds in the loop() function, the tune was able to play without distortion. This was due to the fact that the loop() function would only enter into the if statement controlling the dot matrix once every 50 cycles. This frequency is so high that it is imperceptible to the human ear, ensuring that the tune played smoothly.

Upon running this code, I was able to successfully demonstrate fluid control over the dot matrix, as in the previous section, while simultaneously playing "Mary Had a Little Lamb". This successful outcome confirmed the effectiveness of the methods used to synchronize the playing of a tune with the control of the LED dot matrix.

INTENTIONALLY LEFT BLANK

CODE DOCUMENTATION

Part 1.4

```
/*
*University of Washington
*ECE 474,  7/10/23
*Jason Bentley
*
*Lab 2 - Part 1.4
*
*Acknowledgments: N/A
*/
void setup() {
  DDRL |= (1 << PORTL2); //initialize pin 47
  DDRL |= (1 << PORTL1); //initialize pin 48
  DDRL |= (1 << PORTL0); //initialize pin 49
}

void loop() {
  PORTL &= ~(1 << PORTL0); //turn off pin 49
  PORTL |=  (1 << PORTL2); // turn on pin 47
  delay(333);
  PORTL &= ~(1 << PORTL2); // turn off pin 47
  PORTL |=  (1 << PORTL1); //turnb on pin 48
  delay(333);
  PORTL &= ~(1 << PORTL1); //turn off pin 48
  PORTL |=  (1 << PORTL0); //turn on pin 49
  delay(333);
}
```

Part 2.2

```
/*
* University of Washington
* ECE 474,  7/14/23
* Jason Bentley
*
* Lab 2 - Part 2.4
*
* Acknowledgments:
* Prof. Makhsous in class example in 7/11/2023, forum.arduino.cc,
* stackexchange.com, ChatGPT code interpreter plugin
*/

#include <math.h>
#define PIN_OC4A PH3
```

```
#define CLOCK_SPEED 16000000

/**************************************************
 * void generate_frequency(int, int)
 *    Argument 1 - the frequency in Hz of the tone you wish to generate.
 *                 Pass in 0 for silence.
 *    Argument 2 - the length of time in seconds that the tone will play for
 *
 *    Returns - no returns, this is a void function
 *
 *    The function takes your desired frequency and the length of time for
 *    which you would like to play the tone. Based on this it then calculates
 *    what value to set for the OCR4A flag. If 0 is passed for frequency then
 *    a while loop will iterate a counter by the number of seconds passed in
 *    and do nothing for that period of time. Else, another while loop will
 *    change state of the pin every time the timer reaches the OCR4A flag.
 *
 *    Acknowledgments: Prof. Makhsous in class example in 7/11/2023, ChatGPT code
interpreter plugin
*/
void generate_frequency(int frequency_in_Hz, int seconds){

  double period_halved = (1.0 / frequency_in_Hz) / 2.0;
  int flag = round(period_halved / (256.0 / CLOCK_SPEED));
  int time = seconds * 2 * frequency_in_Hz; //time is how long the while loop
runs by calculating the number of expected state transitions for a given period
of time at a given frequency
  int counter = 0; //counts how many state transistion occur within while loop

  TCCR4B |= (1 << WGM42); //enable CTC mode
  TCCR4B |= (1 << CS42); //set prescaler 256
  TCNT4 = 0; // initialize timer 4 to 0

  if(frequency_in_Hz == 0){

    OCR4A = 62500 * seconds; // 1 / (256/16*10^6)

    while(counter != seconds){
      if (TIFR4 & (1 << OCF4A)){
        TIFR4 |= (1 << OCF4A);
        counter++;
      }
    }

  } else {
    OCR4A = flag;

    while(counter < time){
      if (TIFR4 & (1 << OCF4A)){  // if timer interupt register bit 1 is high for
timer 4
        TIFR4 |= (1 << OCF4A);    // then toggle that bit to 0
        PORTH ^= (1 << PIN_OC4A); // and set the state of the pin to opposite of
what it is now
        counter++;
      }
    }
  }
}


int main(){
  DDRH |= (1 << PIN_OC4A);

  while(1){
    generate_frequency(400,1);
    generate_frequency(250,1);
    generate_frequency(800,1);
    generate_frequency(0,1);
  }
  return 0;
}
```

Part 3.2

```
/*
* University of Washington
* ECE 474,  7/14/23
* Jason Bentley
*
* Lab 2 - Part 3.2
*
* Acknowledgments:
* Prof. Makhsous in class example in 7/11/2023, forum.arduino.cc,
* stackexchange.com, ChatGPT code interpreter plugin
*/

#include <math.h>

#define PIN_OC4A PH3
#define LED_PIN47 PL2
#define LED_PIN48 PL1
#define LED_PIN49 PL0

volatile int LED_state   = 0;

volatile int tone_counter  = 0;
volatile int tone_state    = 0;

volatile int uS_counter  = 0;
volatile int ms_counter  = 0;
volatile int sec_counter = 0;

volatile bool LED_flag  = false;
volatile bool tone_flag = false;


/***************************************************
 * void generate_pattern()
 *    Argument 1 - No arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function watches the sec_counter global variable and sets flags
 *    and controls the orders of operations for blinking the LEDS
 *    and controlling the tones played by the speaker.
 *
 *    Acknowledgments: N/A
*/
void generate_pattern(){
    if (sec_counter < 2){
        LED_flag = true;
    } else if (sec_counter >= 2 && sec_counter < 6){
        LED_flag  = false;
        tone_flag = true;
    } else if (sec_counter >= 6 && sec_counter < 16){
        LED_flag = true;
    } else if (sec_counter >= 16){
        LED_flag = false;
        tone_flag = false;
    }
    generate_LED();
    generate_tone();
}


/**************************************************
 * void generate_LED()
 *    Argument 1 - No arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function watches the ms_counter global variable and the LED_flag
 *    in order to control the sequence of the LEDs. This task is hard coded for
 *    a 3Hz blinking sequence.
 *
 *    Acknowledgments: Prof. Makhsous in class example in 7/11/2023
*/
void generate_LED(){
    if(LED_flag){
        if (ms_counter == 333){
            PORTL &= ~(1 << LED_PIN49); //turn off pin 49
            PORTL |=  (1 << LED_PIN47); // turn on pin 47
            LED_state = 1;
        } else if (ms_counter == 666){
            PORTL &= ~(1 << LED_PIN47); // turn off pin 47
            PORTL |=  (1 << LED_PIN48); //turn on pin 48
            LED_state = 2;
        } else if (ms_counter == 999){
            PORTL &= ~(1 << LED_PIN48); //turn off pin 48
            PORTL |=  (1 << LED_PIN49); //turn on pin 49
            LED_state = 0;
        }
    }

    // Turn off LEDs
    if(!LED_flag){
        PORTL &= ~(1 << LED_PIN49);
        PORTL &= ~(1 << LED_PIN48);
        PORTL &= ~(1 << LED_PIN47);
        LED_state = 0;
    }
}


/***************************************************
 * void generate_tone()
 *    Argument 1 - No arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function watches the tone_flag global variable and controls the
 *    sequence of speaker tones. The tones are hard coded to be 400Hz,
 *    250Hz, 800Hz and 0Hz. Additionally, tones are hard coded to play
 *    for one second each.
 *
 *    Acknowledgments: N/A
*/
void generate_tone(){
    static int time;
    static int half_period;

    int frequency[] = {400, 250, 800, 0};

    if(tone_flag){
        time++;

        // Depending on tone_state value, calculate the 1/2 * period of the
signal
        if (tone_state == 0)
            half_period = round((5000/ frequency[0]));
        else if (tone_state == 1)
            half_period = round((5000 / frequency[1]));
        else if (tone_state == 2)
            half_period = round((5000 / frequency[2]));
        else if (tone_state == 3){
            half_period = 5000;
        }

        // Toggle the state of the OC4A pin on each half period
        if (time == half_period){
            PORTH ^= (1 << PIN_OC4A);
            tone_counter++;
            time = 0;
        }

        // Calcualte how long the tone has been playing and change the tone state
        // when after the tone has been playing for set period of time
        if (tone_counter == 2 * frequency[0] && tone_state == 0){
            tone_counter = 0;
            tone_state = 1;
        } else if (tone_counter == 2 * frequency[1] && tone_state == 1){
            tone_counter = 0;
            tone_state = 2;
        } else if (tone_counter == 2 * frequency[2] && tone_state == 2){
            tone_counter = 0;
            tone_state = 3;
        } else if (tone_counter == 2 && tone_state == 3){
            tone_counter = 0;
            tone_state = 0;
        }
    }

    // Reset tone_state and tone_counter to zero so that
    // tones restart at beginning of sequence.
    if(!tone_flag){
        tone_state = 0;
        tone_counter = 0;
    }

}


void setup(){
  DDRL |= (1 << LED_PIN47); //initialize pin 47
  DDRL |= (1 << LED_PIN48); //initialize pin 48
  DDRL |= (1 << LED_PIN49); //initialize pin 49
  DDRH |= (1 << PIN_OC4A);  //initialize pin 6
}

void loop(){
    //uS counter increments every 100uS
    uS_counter++;

    //delay function is 100 uS, therefore 100uS * 10 = 1ms
    if(uS_counter == 10){
        ms_counter++;
        uS_counter = 0;
    }
    if(ms_counter == 1000){
        sec_counter++;
        ms_counter = 0;
    }
    if (sec_counter == 17){
        sec_counter = 0;
    }
    generate_pattern();
    delayMicroseconds(100); //sync delay function acts as the global clock
}
```

Part 3.3

```
/*
* University of Washington
* ECE 474,  7/14/23
* Jason Bentley
*
* Lab 2 - Part 3.3
*
* Acknowledgments:
* Prof. Makhsous in class example in 7/11/2023, forum.arduino.cc,
* stackexchange.com, ChatGPT code interpreter plugin
*/

#include <math.h>
#define NOTE_C 261
#define NOTE_D 294
#define NOTE_E 329
#define NOTE_F 349
#define NOTE_G 392
#define NOTE_A 440
#define NOTE_B 493
#define NOTE_C 523
#define NOTE_R 5


int melody[] = { NOTE_E, NOTE_R, NOTE_D, NOTE_R, NOTE_C, NOTE_R, NOTE_D, NOTE_R,
NOTE_E,
NOTE_R,NOTE_E, NOTE_R,NOTE_E, NOTE_R,NOTE_D, NOTE_R,NOTE_D, NOTE_R,NOTE_D,
NOTE_R,NOTE_E,
NOTE_R,NOTE_G,NOTE_R,NOTE_G, NOTE_R,NOTE_E, NOTE_R,NOTE_D, NOTE_R,NOTE_C,
NOTE_R,NOTE_D,
NOTE_R,NOTE_E, NOTE_R,NOTE_E, NOTE_R,NOTE_E, NOTE_R,NOTE_E, NOTE_R,NOTE_D,
NOTE_R,NOTE_D,
NOTE_R,NOTE_E, NOTE_R,NOTE_D, NOTE_R,NOTE_C, NOTE_R,NOTE_C };

// uncomment to hold each note longer
// int melody[] = { NOTE_E, NOTE_E, NOTE_R, NOTE_D, NOTE_D, NOTE_R, NOTE_C,
NOTE_C, NOTE_R, NOTE_D, NOTE_D, NOTE_R, NOTE_E, NOTE_E,
// NOTE_R,NOTE_E, NOTE_E, NOTE_R,NOTE_E, NOTE_E, NOTE_R,NOTE_D, NOTE_D,
NOTE_R,NOTE_D, NOTE_D, NOTE_R,NOTE_D, NOTE_D, NOTE_R,NOTE_E, NOTE_E,
// NOTE_R,NOTE_G, NOTE_G,NOTE_R,NOTE_G, NOTE_G, NOTE_R,NOTE_E, NOTE_E,
NOTE_R,NOTE_D, NOTE_D, NOTE_R,NOTE_C, NOTE_C, NOTE_R,NOTE_D, NOTE_D,
// NOTE_R,NOTE_E, NOTE_E, NOTE_R,NOTE_E, NOTE_E, NOTE_R,NOTE_E, NOTE_E,
NOTE_R,NOTE_E, NOTE_E, NOTE_R,NOTE_D, NOTE_D, NOTE_R,NOTE_D, NOTE_D,
// NOTE_R,NOTE_E, NOTE_E, NOTE_R,NOTE_D, NOTE_D, NOTE_R,NOTE_C, NOTE_C,
NOTE_R,NOTE_C, NOTE_C };

#define PIN_OC4A PH3
#define LED_PIN47 PL2
#define LED_PIN48 PL1
#define LED_PIN49 PL0

volatile int LED_state   = 0;

volatile int tone_counter  = 0;
volatile int tone_state    = 0;

volatile int uS_counter  = 0;
volatile int ms_counter  = 0;
volatile int sec_counter = 0;

volatile bool LED_flag  = false;
volatile bool tone_flag = false;


/***************************************************
 * void generate_pattern()
 *    Argument 1 - No arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function watches the sec_counter global variable and sets flags
 *    and controls the orders of operations for blinking the LEDS
 *    and controlling the tones played by the speaker.
 *
 *    Acknowledgments: N/A
*/
void generate_pattern(){

    if (sec_counter < 2){
        LED_flag = true;
    } else if (sec_counter >= 2 && sec_counter <= 6){
        LED_flag  = false;
        tone_flag = true;
    } else if (sec_counter > 6 && sec_counter < 16){
        LED_flag = true;
    } else if (sec_counter >= 16){
        LED_flag = false;
        tone_flag = false;
    }

    generate_LED();
    generate_tone();

}


/**************************************************
 * void generate_LED()
 *    Argument 1 - No arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function watches the ms_counter global variable and the LED_flag
 *    in order to control the sequence of the LEDs. This task is hard coded for
 *    a 3Hz blinking sequence.
 *
 *    Acknowledgments: Prof. Makhsous in class example in 7/11/2023
*/
void generate_LED(){
    if(LED_flag){
        if (ms_counter == 333){
            PORTL &= ~(1 << LED_PIN49); //turn off pin 49
            PORTL |=  (1 << LED_PIN47); // turn on pin 47
            LED_state = 1;
        } else if (ms_counter == 666){
            PORTL &= ~(1 << LED_PIN47); // turn off pin 47
            PORTL |=  (1 << LED_PIN48); //turn on pin 48
            LED_state = 2;
        } else if (ms_counter == 999){
            PORTL &= ~(1 << LED_PIN48); //turn off pin 48
            PORTL |=  (1 << LED_PIN49); //turn on pin 49
            LED_state = 0;
        }
    }

    if(!LED_flag){
        PORTL &= ~(1 << LED_PIN49);
        PORTL &= ~(1 << LED_PIN48);
        PORTL &= ~(1 << LED_PIN47);
        LED_state = 0;
    }
```

```
}


/**************************************************
 * void generate_tone()
 *    Argument 1 - No arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function watches the tone_flag global variable and controls the
 *    sequence of speaker tones. The tones are hard coded in a global
 *    array to play Marry Had a Little Lamb. Additionally, tones are hard
 *    coded to play for 1/5 second each.
 *
 *    Acknowledgments: ChatGPT code interpreter plug-in
*/
void generate_tone(){
    static int time;
    static int half_period;
    static int melody_length = sizeof(melody) / sizeof(melody[0]);

    if(tone_flag){
        time++;

        if (tone_state < melody_length) // Check to see position in melody array
            half_period = round((5000 / melody[tone_state])); //calculate half
period of tone
        else
            half_period = 5000;

        // Toggle the state of the OC4A pin on each half period
        if (time == half_period){
            PORTH ^= (1 << PIN_OC4A);
            tone_counter++;
            time = 0;
        }

        // Check to see if we have reached the end of the melody array. And
        // calcualte how long the tone has been playing and change the tone state
        // when after the tone has been playing for set period of time
        if (tone_counter == melody[tone_state]/5 && tone_state < melody_length){
            tone_counter = 0;
            tone_state++;
        } else if (tone_state == melody_length){
            tone_counter = 0;
```

```
      tone_state = 0;
    }
  }

  // Reset tone_state and tone_counter to zero so that
  // tones restart at beginning of sequence.
  if(!tone_flag){
    tone_state = 0;
    tone_counter = 0;
  }
}


void setup(){
 DDRL |= (1 << LED_PIN47); //initialize pin 47
 DDRL |= (1 << LED_PIN48); //initialize pin 48
 DDRL |= (1 << LED_PIN49); //initialize pin 49
 DDRH |= (1 << PIN_OC4A);  //initialize pin 6
}


void loop(){

  //uS counter increments every 100uS
  uS_counter++;

  //delay function is 100 uS, therefore 100uS * 10 = 1ms
  if(uS_counter ==10){
    ms_counter++;
    uS_counter = 0;
  }

  if(ms_counter == 1000){
    sec_counter++;
    ms_counter = 0;
  }

  if (sec_counter == 17){
    sec_counter = 0;
  }

  generate_pattern();

  delayMicroseconds(100); //sync delay function acts as the global clock
```

```
}
```

LED Matrix Without Tone

```
/*
* University of Washington
* ECE 474,  7/14/23
* Jason Bentley
*
* Lab 2 - Part 4 - LED Matrix
*
* Acknowledgments: Ishaan Bhimani wrote the majoirty of this code
*/

#define OP_DECODEMODE  8
#define OP_SCANLIMIT   10
#define OP_SHUTDOWN    11
#define OP_DISPLAYTEST 14
#define OP_INTENSITY   10

// a global array to store the columns of the LED Matrix
byte column[] = {B10000000, B01000000, B00100000, B00010000, B00001000,
B00000100, B00000010, B00000001};

//Transfers 1 SPI command to LED Matrix for given row
//Input: row - row in LED matrix
//       data - bit representation of LEDs in a given row; 1 indicates ON, 0
indicates OFF
void spiTransfer(volatile byte row, volatile byte data);

// change these pins as necessary
int DIN = 12;
int CS =  11;
int CLK = 10;

byte spidata[2]; //spi shift register uses 16 bits, 8 for ctrl and 8 for data

void setup(){

  //must do this setup
  pinMode(DIN, OUTPUT);
  pinMode(CS, OUTPUT);
  pinMode(CLK, OUTPUT);

  digitalWrite(CS, HIGH);
  spiTransfer(OP_DISPLAYTEST,0);
  spiTransfer(OP_SCANLIMIT,7);
  spiTransfer(OP_DECODEMODE,0);
  spiTransfer(OP_SHUTDOWN,1);

  //clear the matrix
  for (int i = 0; i < 8; i++)
    spiTransfer(i,B00000000);

}

void loop(){

  /*
  * X AXIS
  */
  int x_axis = analogRead(A0);

  //map 0-1023 to 0-8. 8 is set as max because the joystick looses sensitivity
towards
  //the edges of its travel distances in X and Y directions
  x_axis = map(x_axis, 0, 1023, 0, 8);

  //Increase the sensitivity of the edges of the joysticks travel
  if (x_axis == 8)
    x_axis = 7;


  /*
  * Y AXIS
  */
  int y_axis = analogRead(A1);

  //map 0-1023 to 0-8. 8 is set as max because the joystick looses sensitivity
towards
  //the edges of its travel distances in X and Y directions
  y_axis = map(y_axis, 0, 1023, 0, 8);

  //Increase the sensitivity of the edges of the joysticks travel
  if (y_axis == 8)
    y_axis = 7;

  spiTransfer(y_axis, column[x_axis]);
  delay(50);
  spiTransfer(y_axis,B00000000); //reset the LED in the matrix
```

```
}

void spiTransfer(volatile byte opcode, volatile byte data){
  int offset = 0; //only 1 device
  int maxbytes = 2; //16 bits per SPI command

  for(int i = 0; i < maxbytes; i++) { //zero out spi data
    spidata[i] = (byte)0;
  }
  //load in spi data
  spidata[offset+1] = opcode+1;
  spidata[offset] = data;
  digitalWrite(CS, LOW); //
  for(int i=maxbytes;i>0;i--)
    shiftOut(DIN,CLK,MSBFIRST,spidata[i-1]); //shift out 1 byte of data starting
with leftmost bit
  digitalWrite(CS,HIGH);
}
```

LED Matrix With Tone

```
/*
* University of Washington
* ECE 474,  7/14/23
* Jason Bentley
*
* Lab 2 - Part 4 - LED Matrix & Little Lamb
*
* Acknowledgments: Ishaan Bhimani, ChatGPT code interpreter plug in
*/

#define NOTE_C 261
#define NOTE_D 294
#define NOTE_E 329
#define NOTE_F 349
#define NOTE_G 392
#define NOTE_A 440
#define NOTE_B 493
#define NOTE_C 523
#define NOTE_R 5

#define OP_DECODEMODE  8
#define OP_SCANLIMIT   10
#define OP_SHUTDOWN    11
#define OP_DISPLAYTEST 14
#define OP_INTENSITY   10

#define PIN_OC4A PH3


int melody[] = { NOTE_E, NOTE_R, NOTE_D, NOTE_R, NOTE_C, NOTE_R, NOTE_D, NOTE_R,
NOTE_E,
NOTE_R,NOTE_E, NOTE_R,NOTE_E, NOTE_R,NOTE_D, NOTE_R,NOTE_D, NOTE_R,NOTE_D,
NOTE_R,NOTE_E,
NOTE_R,NOTE_G,NOTE_R,NOTE_G, NOTE_R,NOTE_E, NOTE_R,NOTE_D, NOTE_R,NOTE_C,
NOTE_R,NOTE_D,
NOTE_R,NOTE_E, NOTE_R,NOTE_E, NOTE_R,NOTE_E, NOTE_R,NOTE_E, NOTE_R,NOTE_D,
NOTE_R,NOTE_D,
NOTE_R,NOTE_E, NOTE_R,NOTE_D, NOTE_R,NOTE_C, NOTE_R,NOTE_C };

volatile int uS_counter  = 0;
volatile int ms_counter  = 0;
volatile bool tone_flag = true;
volatile int tone_counter  = 0;
volatile int tone_state    = 0;

int x_axis        = 520;
int y_axis        = 520;

// a global array to store the columns of the LED Matrix
byte column[] = {B10000000, B01000000, B00100000, B00010000, B00001000,
B00000100, B00000010, B00000001};

//Transfers 1 SPI command to LED Matrix for given row
//Input: row - row in LED matrix
//       data - bit representation of LEDs in a given row; 1 indicates ON, 0
indicates OFF
void spiTransfer(volatile byte row, volatile byte data);

// change these pins as necessary
int DIN = 12;
int CS =  11;
int CLK = 10;

byte spidata[2]; //spi shift register uses 16 bits, 8 for ctrl and 8 for data

void setup(){

  //must do this setup
  pinMode(DIN, OUTPUT);
  pinMode(CS, OUTPUT);
  pinMode(CLK, OUTPUT);

  DDRH |= (1 << PIN_OC4A);  //initialize pin 6

  digitalWrite(CS, HIGH);
  spiTransfer(OP_DISPLAYTEST,0);
  spiTransfer(OP_SCANLIMIT,7);
  spiTransfer(OP_DECODEMODE,0);
  spiTransfer(OP_SHUTDOWN,1);

  //clear the matrix
  for (int i = 0; i < 8; i++)
    spiTransfer(i,B00000000);

}
```

```
void loop(){

    // uS counter increments every 100uS
    uS_counter++;

    // delay function is 100 uS, therefore 100uS * 10 = 1ms
    if(uS_counter ==10){
        ms_counter++;
        uS_counter = 0;
    }

    generate_tone();

  // read the joystick ADC and update the LED matrix every 50 ms
  if (ms_counter == 50){

    // clear the LED matrix
    for (int i = 0; i < 8; i++)
        spiTransfer(i,B00000000);


    /*********
    * X AXIS *
    **********/
    int x_axis = analogRead(A0);

    //map 0-1023 to 0-8. 8 is set as max because the joystick looses sensitivity
towards
    //the edges of its travel distances in X and Y directions
    x_axis = map(x_axis, 0, 1023, 0, 8);

    //Increase the sensitivity of the edges of the joysticks travel
    if (x_axis == 8)
        x_axis = 7;


    /*********
    * Y AXIS *
    **********/
    int y_axis = analogRead(A1);

    //map 0-1023 to 0-8. 8 is set as max because the joystick looses sensitivity
towards
    //the edges of its travel distances in X and Y directions
    y_axis = map(y_axis, 0, 1023, 0, 8);

    //Increase the sensitivity of the edges of the joysticks travel
    if (y_axis == 8)
        y_axis = 7;

    spiTransfer(y_axis, column[x_axis]);
    ms_counter = 0;
  }
  delayMicroseconds(100); //sync delay function acts as the global clock
}


void spiTransfer(volatile byte opcode, volatile byte data){
  int offset = 0; //only 1 device
  int maxbytes = 2; //16 bits per SPI command

  for(int i = 0; i < maxbytes; i++) { //zero out spi data
    spidata[i] = (byte)0;
  }
  //load in spi data
  spidata[offset+1] = opcode+1;
  spidata[offset] = data;
  digitalWrite(CS, LOW); //
  for(int i=maxbytes;i>0;i--)
    shiftOut(DIN,CLK,MSBFIRST,spidata[i-1]); //shift out 1 byte of data starting
with leftmost bit
  digitalWrite(CS,HIGH);
}

/**************************************************
 * void generate_tone()
 *    Argument 1 - No arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function watches the tone_flag global variable and controls the
 *    sequence of speaker tones. The tones are hard coded in a global
 *    array to play Marry Had a Little Lamb. Additionally, tones are hard
 *    coded to play for 1/5 second each.
 *
 *    Acknowledgments: ChatGPT code interpreter plug-in
*/
void generate_tone(){
    static int time;
    static int half_period;
    static int melody_length = sizeof(melody) / sizeof(melody[0]);

    if(tone_flag){
        time++;

        if (tone_state < melody_length) // Check to see position in melody array
            half_period = round((5000 / melody[tone_state])); //calculate half
period of tone
        else
            half_period = 5000;

        // Toggle the state of the OC4A pin on each half period
        if (time == half_period){
            PORTH ^= (1 << PIN_OC4A);
            tone_counter++;
            time = 0;
        }

        // Check to see if we have reached the end of the melody array. And
        // calcualte how long the tone has been playing and change the tone state
        // when after the tone has been playing for set period of time
        if (tone_counter == melody[tone_state]/5 && tone_state < melody_length){
            tone_counter = 0;
            tone_state++;
        } else if (tone_state == melody_length){
            tone_counter = 0;
            tone_state = 0;
        }
    }

    // Reset tone_state and tone_counter to zero so that
    // tones restart at beginning of sequence.
    if(!tone_flag){
        tone_state = 0;
        tone_counter = 0;
    }
}
```

OVERALL PERFORMANCE SUMMARY

The demonstration proceeded smoothly. I successfully showcased all the code functioning correctly on the Arduino. However, I initially overlooked a requirement in the lab document to write code for the final demonstration, which involved simultaneous control of the LED dot matrix and playing "Mary Had a Little Lamb". Despite this oversight, I quickly rectified the situation by reusing the code from Part 3.3 and the LED Matrix section. As a result, I was able to effectively demonstrate each part of the lab.

In terms of the learning objectives, I believe I successfully manipulated the hardware registers/bits without the use of existing libraries to perform low-level hardware functions for simpler tasks. However, I encountered challenges when trying to generate two PWM waveforms from a single timer. This led me to implement a pseudo-timer, as demonstrated in class. Regarding the coordination of multiple concurrent tasks with round-robin scheduling, I believe I effectively demonstrated my understanding and implementation of this concept. Overall, despite some initial challenges, I was able to meet the learning objectives and successfully complete the lab tasks.

TEAM BREAKDOWN

I am the only member on my team and did all the work.

DICUSSION AND CONCLUSION

The most challenging aspect of this lab for me was the generation of two Pulse Width Modulation (PWM) waveforms from a single timer. This task required a deep understanding of the hardware registers and bits, and the complexity increased when I had to perform low-level hardware functions without the aid of existing libraries. However, this challenge provided an opportunity for me to think creatively and implement a pseudo-timer, as demonstrated in class. This solution not only addressed the issue at hand but also enhanced my understanding of the Arduino's timing mechanisms. I am particularly proud of my ability to coordinate multiple concurrent tasks using round-robin scheduling. This was a complex task that required careful planning and precise execution.

In conclusion, this lab served as an effective introduction to direct register access and bit manipulation, providing a solid foundation for hardware functions, task coordination, and interactive display creation using Arduino. Despite some initial challenges, I was able to meet the learning objectives and successfully complete the lab tasks.