# **Lab 4 Report**

Sage Marks, Matt Krueger

3360:0001 - Embedded Systems

Professor Beichel

University of Iowa, College of Engineering

# 1. Introduction

Embedded Systems Lab 4 introduces a new concept in programming at the hardware level: interrupts. An interrupt is implemented using software to configure a path from an interrupt vector (we used external ATmega328p pins in lab 4) to the processor. This enables the processor to map the incoming signal to its corresponding interrupt service routine in memory. An ISR is simply a subroutine which momentarily stops - or interrupts - the programs current execution to execute the instructions inside of the subroutine jumped to by the interrupt vector.

Lab 4 utilizes this new concept to more easily program the user interface of an embedded system. In previous labs, we have utilized extremely fast polling rates using timers to perceive any change in signals at external pins. The overhead of coordinating timers is drastically reduced using interrupts, not to mention it reduces the latency from a user interaction to an output. We applied interrupts in lab 4 to improve the function of a push button and a rotary pulse generator (RPG) used for a user to control the rpm of a fan using pulse width modulation (PWM).

Listed below is an exhaustive list of hardware required to replicate the lab 4 circuit. Please note that an Arduino Uno contains the microcontroller needed and is sufficient for use in lab. We opted to use the Arduino Uno for simplicity when developing the circuit.

| Hardware | Quantity | Description |
|---|---|---|
| Atmega 328P µC | 1 | Programmable µC |
| Enable Low Push Button | 1 | Control on/off of PWM Fan |
| Rotary Pulse Generator | 1 | Control duty cycle of PWM Fan |
| 16x2 LCD Display | 1 | Display fan status |
| EFB0412VHD-SP05 Fan | 1 | PWM fan |
| 100KΩ Resistor | 1 | Button debouncing |
| 10KΩ Resistor | 6 | RPG, button, and V0 voltage divider |
| 1KΩ Resistor | 1 | V0 voltage divider |
| 0.01µF Capacitor | 4 | Decoupling and filtering |
| Green LED | 1 | Fan on state |
| Red LED | 1 | Fan off state |
| 230Ω Resistor | 2 | Limit LED current |

Figure 1: Materials list

## 2. Schematic

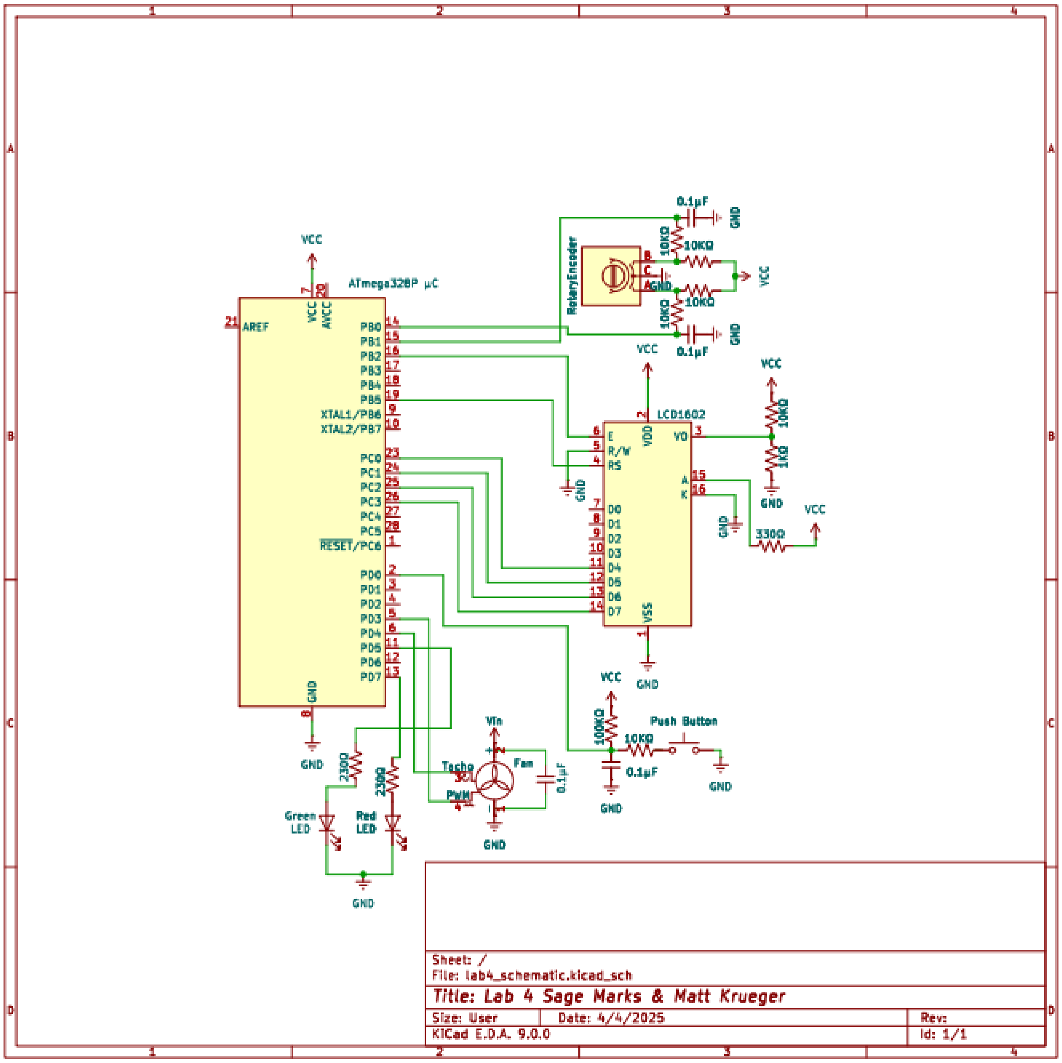


Figure 2: Electrical circuit schematic created using KiCAD

The schematic for our circuit used many components from previous labs, mainly the active low push button and the RPG. We used similar hardware debounce techniques that were implemented in Lab 3. The red and green LEDs were outside the scope of this lab, but they were found to be helpful when debugging to see the state of the fan.

The fan was connected as follows: The voltage wire of the fan was connected to Vin which provides a voltage between 5 to 13 volts as stated in the EFB0412VHD-SP05-1 datasheet. To utilize this voltage, it was necessary to use the wall wart plug that comes with the microcontroller. The fan was also connected to a separate ground pin from the rest of the circuitry. To control the fan via timer counter 2 PWM we connected the signal wire to pin PD5 or OCR2B. The tachometer, an optional portion of this lab was not connected to the microcontroller and no code was written for this feature.

The connections for the LCD were as follows: The enable line was tied to PB2, this line is important as you have to strobe the enable line anytime you want to pass data to the LCD from the microcontroller. The read and write line was set to ground as for this lab, the LCD is only ever written to. The RS line was connected to PB5, this line is used to tell the LCD when it is receiving commands or displaying characters. The LCD was operated in 4-bit mode as this is much more efficient and requires half the connections of the 8-bit mode. Data lines D4-7 were connected to the microcontroller pins PC0-3 respectively. Anytime data in the form of commands or characters was sent to the LCD, these lines were utilized. To see the display of the LCD the contrast voltage had to be set. To do this we created a simple voltage divider with a 100KΩ resistor to 5V and a 10KΩ resistor to ground on pin 3 of the LCD. In an effort to make viewing the LCD even easier, the backlight was turned on by pulling pin 16 of the LCD to ground and connecting pin 15 to VCC via a 330Ω current limiting resistor.

As we progressed through this lab there was a constant challenge of keeping our circuit clean and managing cables. To combat this a 3D printed case was created to house the LCD, fan, microcontroller, and breadboard. With the use of this case and precise wire placement, it was much easier to keep track of wires and their specific uses. An image of the circuit and the case can be seen below. The 3d models used for the Arduino enclosure can be found in Appendix B.

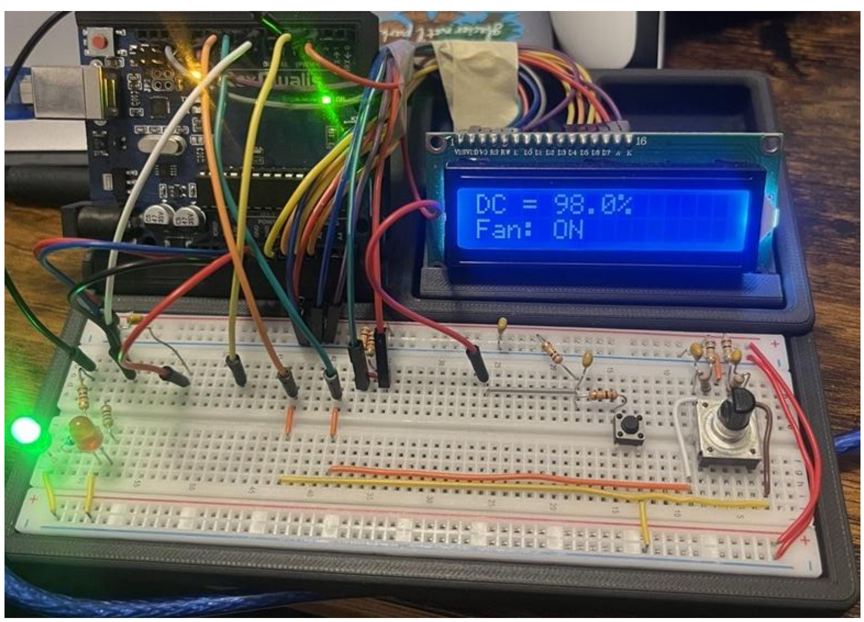


Figure 3: Finished project enclosure

## 3. Discussion

The code for this lab can be split up into three significant sections. The implementation of interrupts, utilizing the PWM functionality of the on-board timer counters, and the use of the LCD. First the use of interrupts will be addressed.

## 3.1 Interrupts

Our program utilized 3 external interrupts, one for the active low push button, and two for the RPG. The active low push button interrupt was used to toggle the fan on and off, subsequently displaying to the LCD the state the fan was in. To make this interrupt we utilized external interrupt INT0. This interrupt was configured as follows:

| | |
|---|---|
| EIRCA | 1<<ISC01 |
| EIMSK | 1<<INT0 |

Figure 4: Interrupt register configurations for external interrupt INT0

To configure the external interrupt as falling edge, a 1 in the ISC01 bit was passed to the external interrupt control register A. Then to enable the interrupt the a 1 in the INT0 bit was passed to the external interrupt mask register. When an interrupt is triggered within the program you are taken to the address of the interrupt from the interrupt vector table. The vector address of INT0 is 0x002. At this address a rjmp command is utilized to send the code to the ISR that handles the logic that is to occur from the interrupt. It is very important to rjmp over this vector table as if the microcontroller were to run through this code issues would occur.

When first trying to implement interrupts in this lab there were various issues. The main one being that one button press was being treated as multiple presses, causing the ISR to be ran multiple times. To combat this a debounce solution was integrating utilizing delays and the interrupt flags. Once this delay, the service routine checks if the fan is in the on or off state. If in the on state the current duty cycle of the fan is saved, and the fan is set to off by sending 0 to OCR2B. After this occurs the LCD cursor is set to the address to write the state (ON or OFF) and OFF is written. When the fan is turned on, the previous duty cycle value is loaded back into OCR2B and the display is updated accordingly. When the fan is on the green LED will be illuminated and when the fan is off the red LED is illuminated. The message displayed was in the second row and can be viewed in the image below.

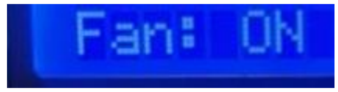


Figure 5: Fan status output to LCD

The two other interrupts utilized in this lab were pin change interrupts, specifically pin change interrupts 0 and 1 (PCINT0 & PCINT1). These interrupts are triggered whenever a change is detected on a pin, namely a switch from 0 to 1 or vice versa. PCINT0 and PCINT1 are located on PB0 and PB1 of the microcontroller. Whenever a logic change was detected on either of these pins, coming from the RPG the interrupt was triggered. The configuration of these two interrupts is shown below.

| | | |
|---|---|---|
| PCICR | (1<<PCIF0) | |
| PCMSK0 | (1<<PCINT1) | (1<<PCINT0) |

Figure 6: Interrupt registers for external Pin Change PCINT0 and PCINT1 interrupts

A 1 bit was passed to the pin change interrupt control register in the location of PCIF0 to enable pin change interrupts 0-7. Then a 1 is passed to the location of PCINT1 and PCINT0 in the pin change mask register 0. This is done to mask out all other interrupts. The vector addresses of these two interrupts are 0x0006 and 0x0008 so the rjmp commands to the ISR were placed there.

Inside the RPG ISR similar logic was implemented as was in lab 3. In the initialization stage of the code, the current RPG state is read and moved to the previous RPG state. When the interrupt is triggered the RPG is at a new state. There is a small debounce delay in the ISR and then we jump into the logic. The entire PINB is read in masking out all other bits besides those connected to the RPG, then by using logical shifts we get the current RPG and previous RPG states in the same register for comparisons to find if we are turning clockwise or counterclockwise. These comparisons can be seen in the table below (in the form prev prev curr curr).

| | |
|---|---|
| 0b0001 | Counter-clockwise |
| 0b0111 | Counter-clockwise |
| 0b1000 | Counter-clockwise |
| 0b1110 | Counter-clockwise |
| 0b0010 | Clockwise |
| 0b0100 | Clockwise |

| | |
|---|---|
| 0b1011 | Clockwise |
| 0b1101 | Clockwise |

Figure 7: rotation encodings checked during PCINT0 and PCINT1 interrupts

Once figuring out what direction we are rotating we utilized an accumulator and threshold to keep RPG turns to only incrementing and decrementing the duty cycle by 0.5%. Each turn without the threshold incremented by around 2%. A threshold of 4 was set so that 4 changes of state had to be detected before the duty cycle was updated. The duty cycle was incremented or decremented by 1 for a turn. This is later divided by 200 to find the DC percentage. After incrementing or decrementing the duty cycle a cursor is set to the location where the percentage is written to the LCD. Then by using some arithmetic logic from the AVR200 library the fraction of OCR2B / OCR2A is performed to get the percent in the form of 0-999 as a percent. Then by dividing by 10 and passing digits to the LCD we can display the current percentage. A limit of 100% was set and a minimum of 1% was set inside the ISR. A separate display routine was created for the 100% display as it is more digits. Everytime the duty cycle of the fan is updated the display percentage was updated in turn. An image of the duty cycle percentage on the LCD is below.


Figure 8: Duty cycle output to LCD

## 3.2 PWM

PWM allows us to simulate analog voltage levels by rapidly toggling a digital output. By adjusting the portion of time the signal remains high in each period, known as the “duty cycle” we can control the current delivered to devices like motors or fans. In this lab, the goal was to use PWM to regulate a cooling fan's speed via Timer/Counter2 on the ATmega328P by creating a pulse width modulated square wave that alters the average voltage, and more importantly current, into the fan.

$$Duty\ Cycle\ (\%) = \frac{Time_{ON}}{Time_{ON} + Time_{OFF}} * 100$$

Figure 9: Duty cycle equation

To utilize and control the fan speed it was essential to understand how PWM works within the microcontroller. There were two main aspects to PWM control in our code, the initialization and then updating of the percentage. In the lab description a frequency of 80Khz was to be used. To calculate this the following formula was utilized.

$$F_{pwm} = \frac{F_{uC}}{(Top + 1)Prescalar}$$

Figure 10: PWM frequency equation

The frequency our microcontroller operates at is 16Mhz and we are solving for a PWM frequency of 80Khz. By doing some simplification we find that using a prescalar of 1 and a top value of 199 will work for our application. Having a top value of 199 also allows us to modify the duty cycle by increments that are less than 1%.

When configuring the PWM we opted to use timer counter 2 as timer counter 1 was already being used to generate our base delay. To utilize timer counter 2, sts commands must be used as timer registers are located in the extended I/O registers. Register 16 is first loaded with the corresponding bits to our configuration and then sts is used to move this registers contents into TCCR2A and TCCR2B. The configuration for the timer counter is shown below.

| | | | |
|---|---|---|---|
| TCCR2A | (1<< COM2B1) | (1<<WGM21) | (1<<WGM20) |
| TCCR2B | (1<<WGM22) | (1<<CS20) | |
| OCR2A | 199 | | |
| OCR2B | 99 | | |

Figure 11: Timer2 configurations for non-inverting, fast-PWM mode

We configured in non-inverting PWM mode by passing a 1 to COM2B1 and have a pre scalar of 1 by passing a 1 to the location of CS20. Fast PWM mode is configured by passing 1's to the locations of WGM22, WGM21, and WGM20. The top value OCR2A is set to 199 to reach the 80Khz frequency and the bottom value of OCR2B is set to 100 to start at a duty cycle of 50%.

When the RPG interrupt is triggered, it is OCR2B that is updated and the remaining configurations stay static. To calculate the duty cycle percentage OCR2B+1 is divided by OCR2A+1 to offset the 0-based indexing of the timer2. It is this percentage that is being displayed and updated on the LCD. An example duty cycle is calculated as shown below:

OCR2A = 199. Let OCR2B = 98

$$Duty\ Cycle\ (\%) = \frac{OCR2B + 1}{OCR2A + 1} * 100 = \frac{98 + 1}{199\ + 1} * 100 = \frac{9900}{200} = 49.5\%$$

Figure 12: Example duty cycle generated by timer2

Because AVR microcontroller does not have native instructions for multiplication and division of numbers, we used subroutines mpy16u and div16u taken from ATMEL AVR200.asm to achieve multiplication of two unsigned integers. Please refer to code for subroutine instructions. Here are the respective register transfer notations (RTN) for each mpy16u and div16u subroutines:

$$mpy16u\text{:}$$
$$r17\text{:}r16 \leftarrow mult(r19\text{:}r18, r17\text{:}r16)$$

$$div16u\text{:}$$
$$r17\text{:}r16 \leftarrow div(r19\text{:}r18, r17\text{:}r16)$$
$$r15\text{:}r14 \leftarrow$$

Figure 12: Register transfer for multiplication and division subroutines

In mpy16u, r19:r18 is the multiplier while r17:r16 is the multiplicand. The product then overwrites r17:r16. In contrast, in div16u, r19:r18 is the divisor while r17:r16 is the dividend. For div16u, there are two outputs: r17:r16 is overwritten with the quotient and r15:r14 stores the remainder. These routines are called after a rpg turn inside of the rpg_change ISR to calculate the duty cycle percentage. The resulting register contents from the mpy16u and div16u are unpacked individually inside of our LCD display functions to write each individual number as a char to the LCD.

## 3.3 LCD

Writing to and configuring the LCD in this lab is very dependent on timing and accurate reading of the datasheet. To configure the LCD to 4-bit mode a specific formula was followed. This formula consisted of setting the LCD to 8-bit mode 3 times and then finally setting the LCD to 4-bit mode. The formula is given below.

| | General Initialization | Example Initialization |
|---|---|---|
| 1 | Wait 100ms for LCD to power up | |
| 2 | Write D7-4 = 3 hex, with RS = 0 | |
| 3 | Wait 5ms | |
| 4 | Write D7-4 = 3 hex, with RS = 0, again | |
| 5 | Wait 200us | |
| 6 | Write D7-4 = 3 hex, with RS = 0, one more time | |
| 7 | Wait 200us | |
| 8 | Write D7-4 = 2 hex, to enable four-bit mode | |
| 9 | Wait 5ms | |
| 10 | Write Command "Set Interface" | Write 28 hex (4-Bits, 2-lines) |
| 11 | Write Command "Enable Display/Cursor" | Write 08 hex (don't shift display, hide cursor) |
| 12 | Write Command "Clear and Home" | Write 01 hex (clear and home display) |
| 13 | Write Command "Set Cursor Move Direction" | Write 06 hex (move cursor right) |
| 14 | -- | Write 0C hex (turn on display) |
| | Display is ready to accept data. | |

Figure 12: Initialization sequence for 4-bit LCD interface

Sending information to the LCD is done by setting the D4-D7 bits and having the RS line set to 0 as we are in command mode. To initialize and send data to the LCD various delay routines were implemented with timer counter 1. To send data with 4 data lines one has to send the upper nibble and then the lower nibble as the LCD can only read data in 8 bit groups. Once the formula was completed we have an LCD that is initialized, on, and ready to receive characters to display.

Displaying to the LCD can be done with single characters or strings of information. Two different subroutines were utilized to send each of these. We used strings for “DC =”, “Fan: “, “ON”, and “OFF”, while we used the character display for the duty cycle display percentage. To display these strings the address of the strings is stored in registers R30 and R31 and our string display subroutine is called. All commands and characters must be sent with the enable line strobe subroutine. This ensures that the LCD can accept commands and data.

To display the variable duty cycle percentage utilizing characters instead of strings was required. Each time the duty cycle was updated and then the percentage was calculated in the form of 0-999, the display function was called. First the display address cursor must be set to the location to which you want to write to. The addresses of the LCD are pictured below.

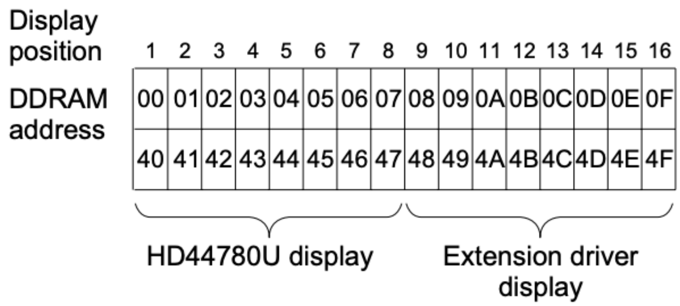


Figure 13: DDRAM addresses of LCD

By correctly setting the cursor address you can avoid having to write the entire display every time there is an update. This looks much better to the user as only certain parts of the display have visual changes. The duty cycle percentage display works by displaying the digits and decimal points in order followed by a percentage sign. When switching between changing the ON and OFF states and the percentage the cursor address has to move between the first and second LCD rows. For the variable duty cycle string, we started writing to address 05. For the variable fan status string, the cursor started at address 45


Figure 14: Superimposed DDRAM addressing on LCD. Highlighted addresses depicting starting addresses of variable strings.

Once configuring the LCD and understanding how the timing and enable lines work, using the peripheral becomes relatively straightforward. This provides a viable alternative to the 7-segment display utilized in previous labs.

## 4. Conclusion

Following the brief interruption of the Embedded Systems midterm, we applied our new-found knowledge of interrupts to build a fan monitoring system. Already having learned the lowest level of program flow of using timers to do routine checks on the status of input pins, interrupts were easily picked up. This change in program flow can be treated conceptually as a higher level: functions responding to events. Though we move towards more abstract logic, the fundamentals of coding remain the same - the processing and transfer of bits. This lab provided more valuable insight into real-world applications such as PWM control and LCDs. The move to C code following this lab is a bittersweet one as many code segments will be much easier to implement and libraries will make our lives much easier. However, over the past weeks, we have grown to love the accurate timing and low-level logic assembly code offers.

# 5. Appendix A: Source Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
;
;                    Lab 4: ECE:3360 Embedded Systems
;
;
;
;                                        Authors:
;
;                                Sage Marks & Matt Krueger
;
;
;
;                                    Project Statement:
;
;                      This AVR program controls a pwm cooling fan, LCD display,
;           active-Low pushbutton, and RPG Encoder to create a fan monitoring system.
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.include "m328pdef.inc"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          Register Aliases                           ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.def dc_high      = r29                                   ; Y reg
.def dc_low       = r28                                   ; Y reg
.def tmp2             = r24             ; temporary register
.def tmp1             = r23             ; temporary register
.def count            = r22             ; stores counter for timer0
.def rpg_current_state   = r21
.def rpg_previous_state  = r20
.def fan_state          = r19             ; boolean flag for fan on/off
.def previous_dc_divisor = r18             ; tracks previous duty cycle divisor
.def current_dc_divisor  = r17            ; tracks current duty cycle divisor
.def rpg_accumulator     = r5                      ; accumulator for our rpg <1% change per turn
.def rpg_threshold       = r4                      ; max for accumulator

.cseg
.org 0x0000
rjmp reset                                                                                   ; jump over interrupts & LUTs

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          Interrupt Vectors                          ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.org 0x0002
rjmp toggle_fan

.org 0x0006
rjmp rpg_change

.org 0x0008
rjmp rpg_change

.org 0x0034                                                                                  ; end of interrupt vector

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          Lookup Tables                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
prefix_string:
        .db "DC = ", 0x00

suffix_string:
        .db "%", 0x00

fan_string:
        .db "Fan: ", 0x00

on_string:
        .db "ON ", 0x00

off_string:
        .db "OFF", 0x00

space_string:
```

```
                .db " ", 0x00

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                              Component Configuration                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
configure_outputs:
   ; port B
                sbi DDRB, 5                                        ; R/S on LCD (Instruction/register
selection) (arduino pin 13)
                sbi DDRB, 2                                        ; E on LCD (arduino
pin ~10)

                ; port C
                sbi DDRC, 3                                        ; D7 on LCD (arduino
pin A3)
                sbi DDRC, 2                                        ; D6 on LCD (arduino
pin A2)
                sbi DDRC, 1                                        ; D5 on LCD (arduino
pin A1)
                sbi DDRC, 0                                        ; D4 on LCD (arduino
pin A0)

                ; port D
                sbi DDRD, 3                                        ; pwm fan signal
(arduino pin ~3)
                sbi DDRD, 5                                        ; green LED indicator
for fan ON (arduino pin ~5)
                sbi DDRD, 7                                        ; red LED indicator for
fan OFF(arduino pin ~7)
                ret

configure_inputs:
                ; port B
                cbi DDRB, 1                                        ; A signal from RPG
(arduino pin 4)
                cbi DDRB, 0                                        ; B signal from RPG
(arduino pin ~5)

                ; port d
                cbi DDRD, 2                                        ; Pushbutton input
signal
                ret

configure_timer0:
                ; TCCR0A:
                ;     ------------------------------------------------------------        ---------------------------------
                ;     | COM0A1 |  COM0A0 | COM0B1 | COM0B0 | - | - | WGM01 | WGM00 |  --->  | 0 | 0 | 0 | 0 | - | - | 0 | 0 |
                ;     ------------------------------------------------------------        ---------------------------------
                ; TCCR0B:
                ;           ------------------------------------------------------        ---------------------------------
                ;           | FOC0A | FOC0B | - | - | WGM02 | CS02 | CS01 | CS00 |  --->  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
                ;           ------------------------------------------------------        ---------------------------------
                ;
                ; Configuration:
                ; - clock source: 16MHz prescaled by 8 to yield a 2MHz tick
                ;                                                          1/2MHz = 0.5us per tick
                ;
                ; - counter operation: normal with top of 0xFF, counting up from `count` (56 decimal)
                ;                                              256 - 56 = 200 ticks before overflow,
                ;                                              200 ticks * 0.5us = 100us
                ;                                  this means that timer overflows every 100us, yeilding a delay of 100us
   ;
                ; - *important note* because we are passing a count into the tcnt0 register, tcnt0 resets at turnover.
                ;    Therefore, we must reload at every overflow (see delay section for details)
                ldi count, 0x38
                ldi tmp1, (1 << CS01)
                out TCNT0, count
                out TCCR0B, tmp1
                ret

configure_timer2:
                ; TCCR2A:
                ;     ------------------------------------------------------------        ---------------------------------
                ;     | COM2A1 |  COM2A0 | COM2B1 | COM2B0 | - | - | WGM21 | WGM20 |  --->  | 0 | 0 | 1 | 0 | - | - | 1 | 1 |
                ;     ------------------------------------------------------------        ---------------------------------
                ; TCCR2B:
                ;           ----------------------------------------------------        ---------------------------------
                ;           | FOC2A | FOC2B | - | - | WGM22 | CS22 | CS21 | CS20|  --->  | 0 | 0 | - | - | 1 | 0 | 0 | 1 |
                ;           ----------------------------------------------------        ---------------------------------
                ;
```

```
		; Configuration:
		; - counter operation: Clear OC2B on compare match, set OC2B at BOTTOM (0). All wgm bits set, so top is fast-pwm with a top of OCR2A
		;									With these selections, we can simlpy increment/decrement OCR2B to 
achieve the duty cycle.
		;								Setting OCR2A to 199, we can increment/decrement OCR2B from 0 to 199 to achieve 
a duty cycle of 0-99%
		;			Thus, an increment/decrement of 1 in OCR2B will change the duty cycle by +/-0.5%
		;
		; - clock source: no prescaling. 16MHz clock, so 1 tick = 0.0625us
		ldi r16, (1 << COM2B1) | (1 << WGM21) | (1 << WGM20)	; Fast pwm, non-inverting (COM0B1=1), TOP=OCR0A (Mode 7)
		sts TCCR2A, r16
		ldi r16, (1 << WGM22) | ( 1<< CS20)							; Prescaler=1 (CS20=1), Fast pwm 
with TOP=OCR0A (WGM02=1)
		sts TCCR2B, r16
		ldi r16, 199
		sts OCR2A, r16
		ldi current_dc_divisor, 195									; initial duty 
cycle is 195/200 = 97.5%
		sts OCR2B, current_dc_divisor
		ret

configure_pushbutton_interrupt:
		; configure INT0 interrupt to trigger on falling edge
		; this is used to control on/off of fan
		ldi r16, (1 << ISC01)			; falling edge
		sts EICRA, r16
		ldi r16, (1 << INT0)			; enable int0
		out EIMSK, r16
		ret

configure_rpg_interrupt:
		; configure PCINT0 and PCINT1
		; this is used to control the duty cycle of the fan
		ldi r16, (1 << PCIE0)						; enable PCINT 7..0
		sts PCICR, r16
		ldi r16, (1 << PCINT1) | (1 << PCINT0)	; PCINT1..0 mask
		sts PCMSK0, r16
		ret

configure_lcd:
	; LCD power-up sequence
		rcall delay_100ms						; wait >40ms
		cbi PORTB, 5							; set R/S to low (data transferred is 
treated as commands)

	; set 8-bit mode by sending 0011 0000 3 times
		rcall set_8_bit_mode
		rcall lcd_strobe
		rcall delay_10ms						; wait for >4.1ms after setting 8-bit (via datasheet pg 45)

		rcall set_8_bit_mode
		rcall lcd_strobe
		rcall delay_1ms							; subsequent delays >100us.
		rcall set_8_bit_mode
		rcall lcd_strobe
		rcall delay_1ms						; delay between commands >100us

		; set 4-bit mode
		set_4_bit_mode:
			ldi r17, 0x02
			out PORTC, r17
			rcall lcd_strobe
		rcall delay_10ms

		; finilize 4-bit mode
		set_interface:
			ldi r17, 0x02
			out PORTC, r17
			rcall lcd_strobe
			rcall delay_100us
			ldi r17, 0x08
			out PORTC, r17
			rcall lcd_strobe
			rcall delay_1ms
		; now 4 bit mode is set

		rcall delay_10ms

		; enable_display_cursor:
		;		ldi r17, 0x00;
```

```
;		out PORTC, r17;
;		rcall lcd_strobe;	; additional strobe included inside of slides:
;		rcall delay_100us;		; not needed as it is overwritten by turn on display
;		ldi r17, 0x08;			; delaying before next command to ensure not busy is sufficient
;		out PORTC, r17;
;		rcall lcd_strobe;
;		rcall delay_10ms

	; reset cursor to home
	clear_home:
		ldi r17, 0x00
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_100us
		ldi r17, 0x01				; 0000 0001 -> return cursor to home
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_10ms

; set cursor move direction to right
	set_cursor_move_direction:
		ldi r17, 0x00
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_100us
		ldi r17, 0x06				; 0000 0110 -> cursor move direction to right
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_1ms

	; turn on display... overwrites display off command enable_display_cursor
	turn_on_display:
		ldi r17, 0x00
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_100us
		ldi r17, 0x0C				; 0000 1100 -> display on, cursor off, blink off
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_1ms

; The following code is for displaying static strings. these are overwritten by the dynamic strings in isrs
	;  ---------------------------------------------------
	;  | D | C | _ | = | _ | | | | | | | | | | |
	;  | F | a | n | : | _ | | | | | | | | | | |
	;  ---------------------------------------------------
	; display prefix on first row
	display_dc_prefix:
		sbi PORTB, 5
		ldi r30, LOW(2 * prefix_string)			; "DC = "
		ldi r31, HIGH(2 * prefix_string)		; obtain address of prefix string
		rcall write_string_to_lcd

	;move the cursor to the second row
	move_cursor_to_second_row:
		cbi PORTB, 5
		ldi r17, 0x0C
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_100us
		ldi r17, 0x00						; move cursor to beginning of
second row (ddram 40)
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_1ms

	; display fan status on second row
	display_fan_status:
		sbi PORTB, 5
		ldi r30, LOW(2 * fan_string)				; "Fan: "
		ldi r31, HIGH(2 * fan_string)			; obtain address of fan string
		rcall write_string_to_lcd

	rcall delay_1ms
	ret

	set_rpg_accumulator:						;Sets the value of the rpg accumulator
to 0
		push r16							;sets the
threshold to 4
```

```
            ldi r16, 0;                                                          ;This way
each rpg turn only increments OCR2B one time (checking for intermediate states)
            mov rpg_accumulator, r16;
            ldi r16, 4;
            mov rpg_threshold, r16
            pop r16
        ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                               MAIN CODE                                   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
reset:
        ; configure components
        rcall configure_outputs
        rcall configure_inputs
        rcall configure_timer0
        rcall configure_timer2
        rcall configure_pushbutton_interrupt
        rcall configure_rpg_interrupt
        rcall configure_lcd
        rcall set_rpg_accumulator

        ; read initial rpg state
        in rpg_previous_state, PINB
        andi rpg_previous_state, 0x03                   ; mask to get pins 5 (A) and 4 (B)

        ; initialie fan to on with current duty cycle quotient set in configuration subroutine
        mov previous_dc_divisor, current_dc_divisor
        ldi fan_state, 0xff                                         ; set fan state to on (1)
        rcall fan_on

        ; initial LED indicators used on circuit
        sbi PORTD, 5
        cbi PORTD, 7
        ; enable global interrupts
        sei

        ; display initial pwm value
        rcall move_cursor_to_dc_addr_lcd
        rcall convert_dc_to_percentage
        rcall write_dc_to_lcd

; program loop. because this is an interrupt-driven program, nothing is in main loop
main:
        rjmp main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                       Pushbutton Interrupt Service Routine
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
toggle_fan:
        push r17
  in r17, SREG
  push r17

        ;debouncing (issue with manufacturing uc, fixed by prof Beichel)
        rcall delay_100ms
        ldi r20, (1<< INTF0)
        out EIFR, r20
        sbic PIND,2
        rjmp exit_toggle

        toggle_code:
            lds r17, OCR2B                      ; get current pwm value
            tst fan_state                                       ; if fan state is 0 (off)
            brne turn_off                       ; if currently ON (0xFF), turn OFF (0x00)

        turn_on:
            ; change indicator LEDs (simply to let user know if button has worked correctly)
            sbi PORTD, 5                                            ; turn green led on
            cbi PORTD, 7                                            ; turn red led off

            ; set fan state to on and restore saved duty cycle
            ldi fan_state, 0xFF
            mov r17, previous_dc_divisor
            in rpg_previous_state, PINB
            andi rpg_previous_state, 0x03
            rjmp update_pwm

        turn_off:
```

```
			; change indicator LEDs (simply to let user know if button has worked correctly)
			cbi PORTD, 5								; turn green led off
			sbi PORTD, 7								; turn red led on

			; set fan state to off and save current duty cycle
			clr fan_state					; set state to OFF
			mov previous_dc_divisor, r17				; save current duty cycle
			ldi r17, 0					; set duty to 0

		update_pwm:
			sts OCR2B, r17					; update pwm register with the stored value

		update_fan_display:
			tst fan_state
			brne display_on
			rcall move_cursor_to_onoff_addr_lcd
			rcall fan_off
			rjmp exit_toggle

			display_on:
			rcall move_cursor_to_onoff_addr_lcd
			rcall fan_on

		exit_toggle:
			pop r17
			out SREG, r17
			pop r17
			reti

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;				RPG Interrupt Service Routine
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
rpg_change:
	; save registers that will be operated on
	push r16
	in r16, SREG
	push r16
	push r17
	push r30

		; exit if fan is off
	rcall delay_100us
		tst fan_state
	breq exit_rpg_isr

		; detect state of RPG pins
		;		-----------------------------------------
		;		| 0 |  0 | 0 | 0 | 0 | 0 | currA | currB |
		;		-----------------------------------------
		in r17, PINB
		andi r17, 0x03

	; build sequence
		; previous state bits are shifted twice, and then combined.
		; because rpg A and B are two bits, shifting twice will result:
		;		-----------------------------------------
		;		| 0 |  0 | 0 | 0 | prevA | prevB | 0 | 0 |
		;		-----------------------------------------
		; then applying bitwise or with current A and current B (without shifting), will result:
		;	-------------------------------------------------
		;	| 0 |  0 | 0 | 0 | prevA | prevB | currA | currB |
		;	-------------------------------------------------
		; now, register 16 is in the form of a unique gray code encoding a cw or ccw turn.
	mov r16, rpg_previous_state
	lsl r16
	lsl r16
	or r16, r17

		; update previous state
	mov rpg_previous_state, r17

		; cases:
	;	- counter-clockwise: 0b0001, 0b0111, 0b1000, 0b1110
	;	- clockwise: 0b0010, 0b0100, 0b1011, 0b1101
		; if none of these cases, jumps to exit
	; if ccw
	cpi r16, 0b0001
	breq counter_clockwise
	cpi r16, 0b0111
```

```
    breq counter_clockwise
    cpi r16, 0b1000
    breq counter_clockwise
    cpi r16, 0b1110
    breq counter_clockwise

    ; if cw
    cpi r16, 0b0010
    breq clockwise
    cpi r16, 0b0100
    breq clockwise
    cpi r16, 0b1011
    breq clockwise
    cpi r16, 0b1101
    breq clockwise

    ; if other
    rjmp exit_rpg_isr

clockwise:
    inc rpg_accumulator                  ; increment accumulator and compare with threshold
    mov r30, rpg_accumulator
    cp r30, rpg_threshold
    brne exit_rpg_isr                    ; if threshold not reached, skip OCR2B update

    clr rpg_accumulator                  ; reset accumulator
    lds r30, OCR2B                       ; get current duty cycle
    cpi r30, 198                         ; check if newest turn reaches max dc
    breq full_speed_call                 ; if at max, don't increment and exit
    cpi r30, 199
    breq exit_rpg_isr
    inc r30                              ; increment
    sts OCR2B, r30                       ; update ocr2b
    rjmp exit_rpg_update

counter_clockwise:
    inc rpg_accumulator                  ; increment the accumulator
    mov r30, rpg_accumulator             ; use r30 as temporary register
    cp r30, rpg_threshold                ; compare with threshold
    brne exit_rpg_isr                    ; if not reached threshold, skip OCR2B update

    clr rpg_accumulator                  ; reset accumulator
    lds r30, OCR2B                       ; get current duty cycle
    cpi r30, 2                           ; check if at min (2 or 1%)
    breq exit_rpg_isr                    ; if at min, don't decrement
    dec r30                              ; decrease duty cycle
    sts OCR2B, r30                       ; update pwm register
    rjmp exit_rpg_update

exit_rpg_update:
    rcall move_cursor_to_dc_addr_lcd     ; move cursor to dc address
    rcall convert_dc_to_percentage       ; convert pwm to percent
    rcall write_dc_to_lcd                ; display pwm
    rjmp exit_rpg_isr

full_speed_call:
        rcall pwm_full_speed

exit_rpg_isr:
    pop r30
    pop r17
    pop r16
    out SREG, r16
    pop r16
    reti

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                     LCD Display
                     ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
set_8_bit_mode:
        ; set 8-bit mode by sending 0011 0000 (this is a subroutine called 3 times to set 8-bit mode)
        ldi r17, 0x03
        out PORTC, r17
        ret

lcd_strobe:
        sbi PORTB, 2                     ; set E to high (initiate data transfer)
        ldi r27, 0x00
        ldi r26, 0x05                    ; delay 500us
```

```
        strobe_loop:
                rcall delay_100us
                sbiw r27:r26, 1
                brne strobe_loop
                cbi PORTB, 2            ; set E to low (end of data transfer)
                ret

write_string_to_lcd:
        lpm r0,Z+                       ; start of loaded memory address
        tst r0                          ; check for terminating character
        breq done                       ; if done, exit
        swap r0                         ; else, swap nibbles
        out PORTC, r0                   ; send upper nibble out
        rcall lcd_strobe                ; latch nibble
        swap r0                         ; lower nibble in place
        out PORTC,r0                    ; send lower nibble out
        rcall lcd_strobe                ; latch nibble
        rjmp write_string_to_lcd        ; continue until done
done:
        ret

write_char1:
        push r16

        ; add 0x30 to r2 and move to r16
        ldi r25, 0x30
        add r25, r2
        mov r16, r25

        ; mask upper nibble, swap, and send
        andi r25, 0xf0
        swap r25
        out PORTC, r25
        rcall lcd_strobe
        rcall delay_100us

        ; mask lower nibble, send, and strobe
        andi r16, 0x0f
        out PORTC, r16
        rcall lcd_strobe
        rcall delay_100us
        pop r16
        ret

write_char2:
        push r16

        ; add 0x30 to r1 and move to r16
        ldi r25, 0x30
        add r25, r1
        mov r16, r25

        ; mask upper nibble, swap, and send
        andi r25, 0xf0
        swap r25
        out PORTC, r25
        rcall lcd_strobe
        rcall delay_100us

        ; mask lower nibble, send, and strobe
        andi r16, 0x0f
        out PORTC, r16
        rcall lcd_strobe
        rcall delay_100us
        pop r16
        ret

write_char3:
        push r16

        ; load 0x30 into r25 and add r0
        ldi r25, 0x30
        add r25, r0
        mov r16, r25

        ; mask upper nibble, swap, and send
        andi r25, 0xf0
        swap r25
        out PORTC, r25
        rcall lcd_strobe
```

```
		rcall delay_100us

		; mask lower nibble, send, and strobe
		andi r16, 0x0f
		out PORTC, r16
		rcall lcd_strobe
		rcall delay_100us
		pop r16
		ret

write_decimal:
		; load 0x02 into r25 and send
		ldi r25, 0x02
		out PORTC,r25
		rcall lcd_strobe
		rcall delay_100us

		; load 0x0e into r25 and send
		ldi r25,0x0e
		out PORTC,r25
		rcall lcd_strobe
		rcall delay_100us
		ret

; move cursor to ddram 05 of lcd
move_cursor_to_dc_addr_lcd:
		cbi PORTB, 5
		ldi r17, 0x08
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_100us
		ldi r17, 0x05						; move cursor to position 5
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_1ms
		sbi PORTB, 5
		ret

; move cursor to ddram 05 of lcd and display longer string (100.0%)
pwm_full_speed:
		inc r30
		sts OCR2B, r30
		rcall move_cursor_to_dc_addr_lcd
		ldi r16, 1
		mov r2, r16
		rcall write_char1
		ldi r16, 0
		mov r2, r16
		rcall write_char1
		rcall write_char1
		rcall write_decimal
		rcall write_char1
		ldi r30,LOW(2 * suffix_string)				; "%"
		ldi r31,HIGH(2 * suffix_string)			; obtain address of suffix string
		rcall write_string_to_lcd

		exit_full_speed:
			ldi r16, 3
			mov rpg_accumulator, r16
			ret

; move cursor to ddram 45 of lcd
move_cursor_to_onoff_addr_lcd:
		cbi PORTB, 5
		ldi r17, 0x0C
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_100us
		ldi r17, 0x05							; move cursor to position 45
		out PORTC, r17
		rcall lcd_strobe
		rcall delay_1ms
		sbi PORTB, 5
		ret

; load address of "ON" in mem for display
fan_on:
		ldi r30,LOW(2 * on_string)
		ldi r31,HIGH(2 * on_string)
		rcall write_string_to_lcd
```

```
            ret

; load address of "OFF" in mem for display
fan_off:
            ldi r30,LOW(2 * off_string)
            ldi r31,HIGH(2 * off_string)
            rcall write_string_to_lcd
            ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          Duty Cycle to LCD Subroutine                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; because dc is displayed as a percentage, we need to multiply by 100.
; even though registers are 8 bits and range from 0-255, multiplying 2 8-bit values results in a 16-bit value.
;                                                       generally, b * b = 2b
;
; this is the highest resolution that we need for our application. Because we have valid dc divisor
; values from 0-200 our range of values will not exceed 65535 which is represented in 16-bits (2^16)
;
; Application Note AN_0936 "AVR200: Multiply and Divide Routines":
; - mpy16u: r17:r16 <- r17:r16 * r19:r18
; - div16u: r17:r16 <- r17:r16 / r19:r18
convert_dc_to_percentage:
  ; operation:
  ;          ocr2b / ocr2a * 100 = percentage
  push r1
  push r14
  push r15
  push r16
  push r17
  push r18
  push r19
  push r20
  push r21
  push r22

            lds r16, low(OCR2B)
  lds r17, high(OCR2B)
            inc r16

  ; multiply by 100
  ldi r18, low(100)
  ldi r19, high(100)
  rcall mpy16u              ; r17:r16 = (OCR2B+1) * 100

  ; divide by ocr2a + 1 (max pwm value plus 1, divisor)
  ldi r18, low(200)
  ldi r19, high(200)
  rcall div16u            ; r17:r16 = quotient, r15:r14 = remainder

  ; save quotient for display
            ldi r18, low(10)
            ldi r19, high(10)
            rcall mpy16u

            mov dc_low, r16
            mov dc_high, r17

  ; multiply remainder by 10 and divide again for decimal place
  mov r16, r14
  mov r17, r15
  ldi r18, low(10)
  ldi r19, high(10)
  rcall mpy16u              ; r17:r16 = remainder * 10

  ; divide by 200 again to get decimal place
  ldi r18, low(200)
  ldi r19, high(200)
  rcall div16u            ; r17:r16 = decimal place

            add dc_low, r16
            adc dc_high, r1

  ; r16 now has our decimal digit
  mov dc_low, r28           ; store integer part
  mov dc_high, r29

  pop r22
  pop r21
  pop r20
```

```
    pop r19
    pop r18
    pop r17
    pop r16
    pop r15
    pop r14
    pop r1
    ret

write_dc_to_lcd:
        push r0
        push r1
        push r2
        push r14
        push r15
        push r16
        push r17
        push r18
        push r19
        push r20

        ; get duty cycle
        mov r16, dc_low
        mov r17, dc_high

        ; divide by 10
        ldi r18, low(10)
        ldi r19, high(10)

        rcall div16u
        mov r0, r14
        rcall div16u
        mov r1, r14
        rcall div16u
        mov r2, r14

        rcall write_char1
        rcall write_char2
        rcall write_decimal
        rcall write_char3

        ldi r30,LOW(2 * suffix_string)                          ; "%"
        ldi r31,HIGH(2 * suffix_string)         ; obtain address of suffix string
        rcall write_string_to_lcd;

    ; this handles case where 100.0% and then decremented to some other value in form xx.x%
        ; this overwrites the "%" with a space
        ldi r30,LOW(2 * space_string)                   ; " "
        ldi r31,HIGH(2 * space_string)                          ; obtain address of space string
        rcall write_string_to_lcd;

        pop r20
        pop r19
        pop r18
        pop r17
        pop r16
        pop r15
        pop r14
        pop r2
        pop r1
        pop r0
        ret

; division code taken from ATMEL AVR200.asm
div16u:
        clr     r14     ;clear remainder Low byte
        sub     r15, r15;clear remainder High byte and carry
        ldi     r20, 17 ;init loop counter
        d16u_1:
                rol     r16             ;shift left dividend
                rol     r17
                dec     r20             ;decrement counter
                brne    d16u_2          ;if done
                ret                     ;    return
        d16u_2:
                rol     r14     ;shift dividend into remainder
                rol     r15
                sub     r14, r18        ;remainder = remainder - divisor
                sbc     r15, r19        ;
                brcc    d16u_3          ;if result negative
```

```
			add		r14, r18	;	restore remainder
			adc		r15, r19
			clc				;	clear carry to be shifted into result
			rjmp		d16u_1		;else
		d16u_3:
			sec				;	set carry to be shifted into result
			rjmp		d16u_1


; Multiplication for getting value of 0-999, taken from ATMEL AVR200.asm
; mpy16u: r17:r16 <- r17:r16 * r19:r18
mpy16u:
		clr		r21			;clear 2 highest bytes of result
		clr		r20
		ldi		r22,16		;init loop counter
		lsr		r19
		ror		r18
		m16u_1:
			brcc		noad8			;if bit 0 of multiplier set
			add		r20,r16		;add multiplicand Low to byte 2 of res
			adc		r21,r17		;add multiplicand high to byte 3 of res
		noad8:
			ror		r21			;shift right result byte 3
			ror		r20			;rotate right result byte 2
			ror		r19			;rotate result byte 1 and multiplier High
			ror		r18			;rotate result byte 0 and multiplier Low
			dec		r22			;decrement loop counter
			brne		m16u_1		;if not done, loop more
			mov r16, r18
			mov r17, r19
			ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;				Timer0 Delays				;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
delay_100ms:
		ldi r27, 0x03
		ldi r26, 0xE8
		loop_100ms:
			rcall delay_100us
			sbiw r27:r26, 1
			brne loop_100ms
			ret

delay_10ms:
		ldi r27, 0x00
		ldi r26, 0x64
		loop_10ms:
			rcall delay_100us
			sbiw r27:r26, 1
			brne loop_10ms
			ret

delay_1ms:
		ldi r27, 0x00
		ldi r26, 0x0a
		loop_1ms:
			rcall delay_100us
			sbiw r27:r26, 1
			brne loop_1ms
			ret

delay_100us:
		in tmp1, TCCR0B
		ldi tmp2, 0x00
		out TCCR0B, tmp2
		in tmp2, TIFR0
		sbr tmp2, (1 << TOV0)
		out TIFR0, tmp2
		out TCNT0, count						; reload tcnt0 with count (56)
		out TCCR0B, tmp1
		wait_for_overflow:
			in tmp2, TIFR0
			sbrs tmp2, TOV0
			rjmp wait_for_overflow
			ret
```

## 6. Appendix B: References


Atmel Corporation. *AVR200: Multiply and Divide Routines. September 2009.* <https://ww1.microchip.com/downloads/en/Appnotes/doc0936.pdf >

Beichel, Reinard. *Embedded Systems, Rotary Pulse Generators and Lab 3. The University of Iowa, 2025* <https://uiowa.instructure.com/courses/248357/files/29778930?module_item_id=8162158 >

Beichel, Reinard. *Embedded Systems, Lecture 8: Timers. The University of Iowa, 2025* <https://uiowa.instructure.com/courses/248357/files/29853139?module_item_id=8165716 >

Beichel, Reinard. *Embedded Systems, Lab 4 Considerations. The University of Iowa, 2025* <https://uiowa.instructure.com/courses/248357/files/29941936?module_item_id=818337 >

Beichel, Reinard. *Embedded Systems, Lecture 10: Liquid Crystal Displays. The University of Iowa, 2025* <https://uiowa.instructure.com/courses/248357/files/29883422?module_item_id=8183401 >

Beichel, Reinard. *Embedded Systems, Lecture 11: Interrupts. The University of Iowa, 2025* <https://uiowa.instructure.com/courses/248357/files/30040783?module_item_id=8183403 >

Delta Electronics. *EFB0412VHD-SP05 DC Fan Specification*. January 4, 2008. <https://www.delta-fan.com/Download/Spec/EFB0412VHD-SP05.pdf>

Hitachi. *HD44780U (LCD-II) Dot Matrix Liquid Crystal Display Controller/Driver*. September 1999. <https://cdn.sparkfun.com/assets/9/5/f/7/b/HD44780.pdf>

MakerWorld. Adjustable Arduino Uno Breadboard Workstation. Sept. 12, 2024. https://makerworld.com/en/models/864695-adjustable-arduino-uno-breadboard-workstation?from=search#profileId-815661

MakerWorld. PCB Stand for Arduino, Raspberry Pi, LCD Display. Dec. 26, 2024 https://makerworld.com/en/models/916884-pcb-stand-for-arduino-raspberry-pi-lcd-display?from=search#profileId-878377

Microchip Technology. *ATmega328P – 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. January 2015. <https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf>

Microchip Technology. AVR200: Multiply and Divide Routines. September 2009. https://ww1.microchip.com/downloads/en/Appnotes/doc0936.pdf

Pighixxx. *The Definitive Arduino Uno Pinout Diagram*. *May 5, 2013.* <https://uiowa.instructure.com/courses/248357/files/29320694?module_item_id=8042318>