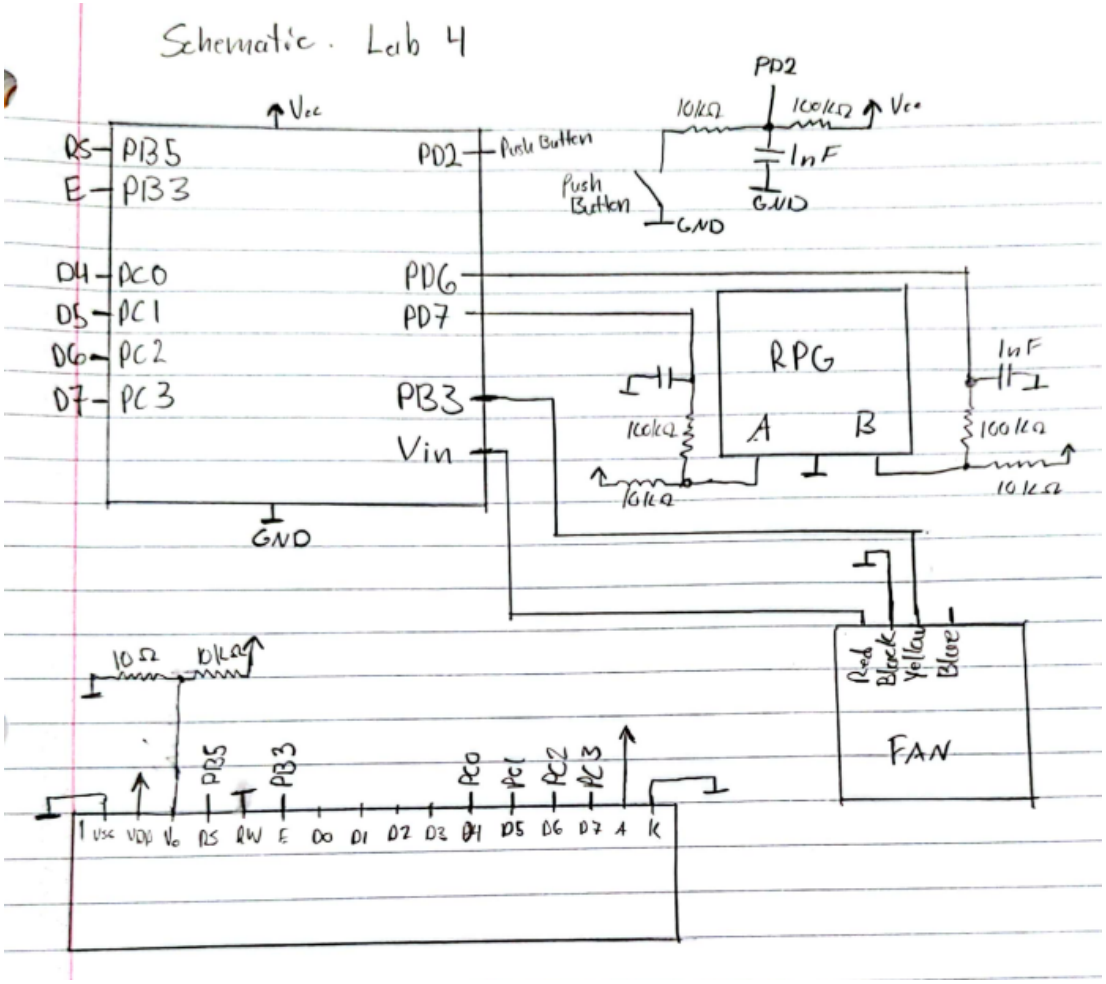
Authors: Brandon Cano, Ian Kuk
Professor Beichel
ECE:3360 Embedded Systems
Post-Lab 4 Report

1. Introduction

The goal of this lab was to create a microcontroller based monitoring system utilizing a LCD display and pulse width modulation (PWM) to determine the speed of the fan. More specifically, a rotating pulse generator (RPG) is used to change the values for the PWM in order to adjust the fan speed and a push button is used to toggle the fan either on or off. All while an LCD display is showing the information related to the fan, such as whether it's on or off, and the current PWM percentage.

2. Schematic

This image shows a simplified view of how we implemented our circuit. The inputs and outputs of the microcontroller are labeled to which pins they go to for all the components. For the LCD display, bottom left, all the pins that go to the microcontroller are labeled. For both the button and the RPG we used a physical debounce which is shown using 100kΩ pull-up resistors and 1nF capacitors.



3. Discussion

When designing the project it was split off into two categories, the hardware and the software. We first started with getting the hardware built.

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.

To debounce the RPG 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 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 PC0 through PC3 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.

For the fan, it was a simple setup to get it working. To start we took the black wire and put it to ground and the red wire went to the Vin input on the microcontroller. From the datasheets the fan needs more than 5V in order to operate at its maximum duty cycle which is why the Vin input is used rather than the power source from the breadboard. The yellow wire then went into port PD3, this is because on the arduino data sheet it shows that this has the PWM associated and it has the OC2B which is what is used in the code.

This lab required that the PWM frequency must be 80kHz. The microcontroller runs at 16mHz, so in order to find out what base value needed to be loaded into OC2B we divided the microcontroller's frequency by the desired frequency. The calculation was 16,000,000 / 80,000 = 200. The resultant value of 200 was low enough that no prescaler was necessary, so the prescaler TCCR2B was set to one. The duty cycle is created by dividing OC2A by OC2B, with OC2B being set at 200 and the fact that we wanted to start at a duty cycle of 50% we initialized OC2A with the value of 100. Our goal was to increment and decrement the duty cycle by 1% with every twist of the RPG, in order to achieve this goal we would either increase or decrease the value of OC2A by a value of 2. By dividing 100 by 200 you get a value of .50, if you decrease the value of 100 by 1 and divided by 200 you get a value of .495 showing a decrease of .5% meaning we had to change by a factor of 2 in order to get the desired 1% change in the duty cycle.

For the software, there were a lot of parts involved with this section so each part will be explained in its own section.

For the button that was needed in the lab, we had to utilize an interrupt to detect when the button was pressed. The wire for the button was placed into pin PD2, this allowed us to use the INT0 interrupt to detect the button press. This was used because that means we were able to have only one input detect and trigger the interrupt, preventing other potential mismatching. Looking at the data sheet, it shows that the address it jumps to is 0x0002 and at this point is where we jump to the 'toggle_fan' subroutine. Inside this routine, we first clear the global interrupt flag so that it can't be triggered by another interrupt. Then we check using the 'sbis PIND, 3' we can determine if the fan is on or off. If on, then we know to turn it off, if it's off, then we turn it on. In order to keep the interrupt subroutines short, we set a flag with either a value of 0 or 1, so we know what to update the display as. Then we use reti, to return back from the interrupt and to enable the global interrupt flag.

Next we also had to use interrupts to detect when a change occurred with the RPG. To do this, the two signals were plugged into PD6 and PD7, which allows us to use PCINT2, which both PCINT22 and PCINT23 can trigger. From the data sheet we could find that 0x000A is where the program will jump to once the interrupt was detected. In the subroutine 'check_rpg' we first clear the global interrupt flag to prevent any other sudden movements from causing issues. Since the RPG can't affect the duty cycle while the fan is off, we check to see if the fan is on or not the same way as the push button. If the fan is off, then we jump to the end of the routine and reti so another call can be made. If the fan is on, then we check to see which pin, either 6 or 7 triggered the interrupt, if it was 7, then we move clockwise, otherwise counter clockwise. In the directional subroutines, we first clear the direction flag for the RPG, that is used in the main loop, then we check to see if counter register R18 has reached one of its bounds, depending on the direction. If the bound is not reached, then the flag is updated to tell the main loop which direction to move, 1 is clockwise, 2 is counter clockwise. If a bound is hit, meaning R18 is 1 when trying to decrease or R18 is 100 when trying to increase, we simply skip over the directional flag and reti to get back to the main loop and nothing will be updated.

In this lab we needed to implement timers of different lengths. In the code we have small durations of <1ms all the way up to 100ms. Like the last lab, the micro-controllers internal timer cannot do durations of 100ms. To solve this problem we used the micro-controllers internal timer to make a 1ms delay and a 50us delay. This was achieved by setting TCCR0B to 0x05, this set the timer's prescaler to 1024. The next step was to find the value to output into TCNT0. The first step was to find the tmax, this was found by taking the microcontrollers frequency multiplying it by the prescaler and 256. The microcontrollers frequency is 16mHz so the equation looked like (1/16,000,000)*1024*256, resulting in a tmax of 16.384 ms. The next step is to find n, to find n we divide the time we want the timer to be by tclk. To find tclk we divide tmax by 256. The equation for tclk is 16.384ms/256 giving 64 µs. With tclk found we divided the desidered time of 10ms by 64µs to give a value of 156.25. This value has to be rounded so we chose to round up, giving n the value of 157. The final step was to load 256 minus n,which is 99, into TCNT0. With TCNT0's value set we simply repeatedly loop a subroutine that checks its value till it is 0, thus creating a 10ms timer.

In order to write to the LCD, there are many parts that go into it, first we have to initialize everything for the LCD. To start, we need to first clear the pins for RS and E so we can write basic commands to the LCD. During this initialization we call a subroutine called 'pulse_lcd' which essentially will enable the high, then delay briefly before clearing it back to low. We then wait about 2ms in order for the pulse to process on the LCD. The last part of the initialization is writing all the commands to the LCD that will likely be used at some point. For each command we set the hex value into a register, then we call another subroutine called 'write_command'. This will set RS to low and then place both the upper and lower nibbles into the LCD. After this initialization, we move onto the next part which is the 'display_lcd' subroutine, which will first clear the interrupt flag so that the screen can always be fully adjusted before

another change happens. Then we clear the LCD and set RS high so that we can rewrite the new strings to it, starting with the top line, which contains the “DC= XY%” which shows some percentage of the calculated duty cycle. In order to show the correct percentage, we first need to check to see if we are at 100%, if we are then we simply load a specific string, msg100, which contains “DC=100%”. If it is not, then we jump down to the normal message which just contains “DC= “ without the number. We then call the divide method, which will take the duty cycle and divide by 10 to find the remainder and result to show our percentage. For us to push these results we call our `display_data` subroutine which will push the remainder and result onto the LCD. For the second line which shows the status of the fan, we do a similar set of logic, we check the flag for the fan, and if it is on, we will display the “ON” string, otherwise if the fan is off we display “OFF”. Then at the end we call the ‘display_message’ subroutine to show this last piece to the LCD.

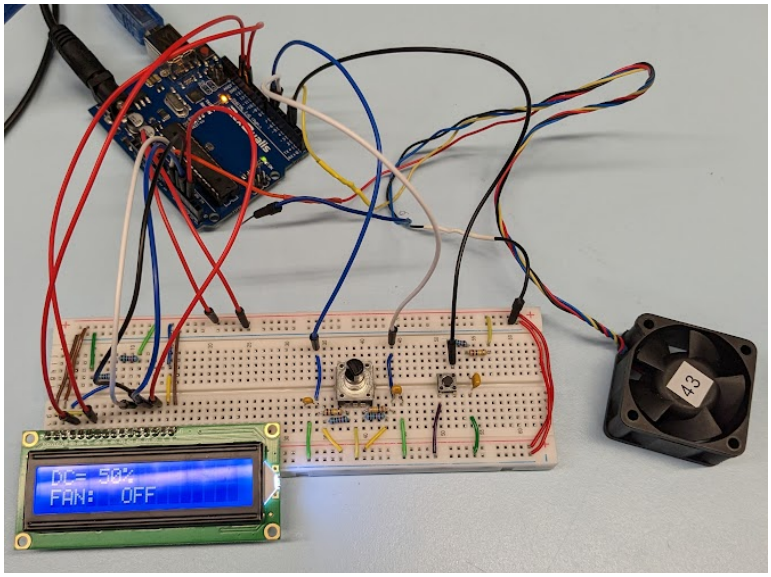
The part that actually pushes the message to the LCD is the ‘display_message’ subroutine and this utilizes the Z register to do so. Essentially how this works is that it will loop through every byte that needs to be displayed on the LCD. Within each iteration, we swap around the R0 register and push it out to PORTC, then pulse and wait for the LCD. We do this a second time before decrementing R24, which was set in the ‘display_lcd’ subroutine saying how many characters there are to display. We continue to loop until R24 is at 0, then we clear the flag to update the display and set the global interrupt flag. Then we also have ‘display_data’ which does a very similar task to ‘display_message’ but instead of a string, we are taking the remainder and result from the division subroutine and pushing those numbers to the LCD, which means we do not loop at all.

In order for us to be able to properly calculate the percentage of the duty cycle, we need to divide the number by then to split the result into individual numbers. Since, there is no built in instruction for this, we had to create it ourselves. In order for us to get both the result and the remainder we had to use a restoring division algorithm to compute both. At the start of the subroutine, we have to load in our divisor, 10, to R19. We also take the value from R18 and push it into the stack, this gives us the initial setup. The next several steps will all loop multiple times depending on the value of R18. First, we shift both A (R16) and Q (R17) to the left with ‘rol’ and decrement the counter for the correct number of cycles. If it’s not at the end of the process, then we branch to another section, ‘divide_two’ this shifts the carry, then computes the new remainder value. If it is negative we don’t want to restore any remainder, so we set the carry and jump back to the beginning of the loop to shift again for the next cycle. If the remainder is not negative then we add R16 and R19 to restore the remainder and clear the carry which will be shifted into the result, then jump back to the start of the loop. This process continues until R29 is 0, once 0 is reached then we have to convert our values into proper ASCII values for the LCD display and then R16 and R17 are now ready to be sent into the LCD display.

After all the initial setup in the code, we then have our main loop which is always looping unless an interrupt is called. In essence, this is where all the flags get checked to see if any action needs to take place, like what direction and RPG was turned and if the LCD display needs to be updated.

For the clockwise or counterclockwise moves that are called from the main loop, they both perform a similar action. For clockwise motion, we first clear the interrupt flag, then we check on of the lower bounds to help prevent the code from breaking. If this condition is not met, then we continue along to increment a counter for the display, R18, and then check for the upper bound of 100 to prevent it from further increasing. If not at 100, then the duty cycle value in OCR2B is loaded in R16, and incremented by a value of 2, which is roughly 1% as determined by the PWM calculations above, and reload the new value back into the OCR2B register. We then reset our RPG flag, since the move is completed. Following this the ‘display_lcd’ is called so that the lcd can now show the updated percentage on screen. For the

counterclockwise direction, it's almost identical except we decrease instead of increase and check the lower bound of 1.



4. Conclusion

From this lab, we both learned how to set up an LCD display along with the setup and different commands that are used in order to send values, or strings, to the display. For the RPG and push button, we had to learn how to utilize interrupts to detect when an event occurs and be able to set flags in order to determine when an action should occur in order to keep the interrupt routines short. Finally, we had to do calculations for PWM and the duty cycle to change the speed of a fan while using the RPG as the components to make the changes occur.

5. Appendix A: Source Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Lab4.asm
;
; Author : Ian Kuk, Brandon Cano
;
; Notes: I/0 pins and registers
;        PD2   -> Push Button
;        PD3   -> Yellow Fan Wire
;        PD6   -> RPG Right
;        PD7   -> RPG Left
;        PB3   -> LCD Enable
;        PB5   -> LCD RS
;        PC0-3 -> LCD D3-7
;        ...
;   VIN -> Fan Power
;   A0  -> LCD
;        A1  -> LCD
;        A2  -> LCD
;        A3  -> LCD
;
;   R0
;   R16 -> Temp value holder
;   R17 -> Used in divide subroutine
```

```
;   R18 -> Hold duty cycle percentage for LCD display
;   R19 -> Divisor in division
;   R20 -> Flag for RPG direction - 0x00 -> no turn, 0x01 -> CW, 0x02 -> CCW
;   R21 -> Flag to see if display needs to be updated - 0x00 -> no, 0x01 -> yes
;   R22 -> Flag to see if the fan is on or off - 0x00 -> OFF, 0x01 -> ON
;   R23 -> Register that is used to copy and push to stack and temporarily hold values
;   R24 -> Used as the character counter for displaying strings to the LCD
;   R25 -> Used for timer
;   R26 -> Used for timer
;   R27 -> Holds values read in from PIND to see what's on or off
;   R28 -> UNUSED
;   R29 -> LCD Display commands and division use
;   R30 -> Message Use
;   R31 -> Message Use
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.cseg
.org 0x0000
rjmp main                           ; skip over addresses for interrupts

.org 0x0002                         ; address for INT0 - for button detection
rjmp toggle_fan

.org 0x000A                         ; address for PCINT2 - for RPG detection
rjmp check_rpg

; strings
msg:    .db "DC= "
msg100: .db "DC=100"
msg2:   .db "%                                FAN:  "
off:    .db "OFF       "
on:     .db "ON        "
        .dw 0

; start of main program
.org 0x0034
main:
         cli                        ; clear the global interrupt

         ldi R16, 0                 ; set timer counter to 0
         sts TCNT2, R16

         sbi DDRD, 3                ; set the port that has the fan wire as an output
         ldi R16, 0x0F              ; set pins 0-3 on PORTC as outputs
         out DDRC, R16
         ldi R16, 0x38              ; set pins 11-13 on PORTB as outputs
         out DDRB, R16

         ldi R16, 200               ; set top value 200, with prescaler 1 it will make a 80kHz signal
         sts OCR2A, R16
         ldi R16, 100               ; half the value will set the duty cycle to 50%
         sts OCR2B, R16

         ldi R16, 0x23              ; enables fast PMW and non-inverting output
         sts TCCR2A, R16
         ldi R16, 0x01              ; set prescaler 1
         sts TCCR2B, R16

         ldi R16, 0x03              ; rising edge will generate a request to the interrupt
         sts EICRA, R16
         ldi R16, 0x01              ; enable INT0
         out EIMSK, R16
         ldi R16, 0b01000000        ; RPG - PCINT22 interrupt
         sts PCMSK1, R16
         ldi R16, 0b10000000        ; RPG - PCINT23 interrupt
         sts PCMSK2, R16
         ldi R16, 0x06              ; enable pin change interrupt for the above PCINTs
```

```
        sts PCICR, R16

        ldi R16, 0x00              ; initialize registers with default values
        ldi R17, 0x00
        ldi R18, 50
        ldi R20, 0x00
        ldi R21, 0x00
        ldi R22, 0x00

        ; initialize LCD
        cbi PORTB, 3
        cbi PORTB, 5

        ldi R29, 0x03
        out PORTC, R29
        rcall pulse_lcd

        ldi R29, 0x03
        out PORTC, R29
        rcall pulse_lcd

        ldi R29, 0x03
        out PORTC, R29
        rcall pulse_lcd

        ldi R29, 0x02
        out PORTC, R29
        rcall pulse_lcd

        ; writes all the commands to the LCD
        ldi R29, 0x01           ; clear and home
        rcall write_command
        ldi R29, 0x06           ; set cursor move direction
        rcall write_command
        ldi R29, 0x08           ; enable display/cursor
        rcall write_command
        ldi R29, 0x0C           ; turn on display
        rcall write_command
        ldi R29, 0x28           ; set interface
        rcall write_command
        sei                                             ; enable global interrupt flag

/*
 * this will load the Z register with the message to be displayed to the LCD
 */
display_lcd:
        cli
        ldi R29, 0x01           ; clear the LCD
        rcall write_command
        sbi PORTB, 5            ; set R/S to write characters

        ldi R24, 4              ; set number of characters for the base percentage message
        cpi R18, 100            ; if R18 is 100 then we want to branch to show "DC=100%"
        brge set5
        rjmp line_one           ; otherwise, we jump down to the regular 2 digit percentage
        set5:
                ldi R24, 6         ; set the number of characters needed to display 100%
                ldi R30, LOW(2*msg100)   ; load lower byte into the low Z register
                ldi R31, HIGH(2*msg100)  ; load higher byte into the high Z register

                rcall display_message    ; send data to LCD
                rjmp line_two           ; jump to the end to display line 2
        line_one:
                ldi R30, LOW(2*msg)      ; load lower byte into the low Z register
                ldi R31, HIGH(2*msg)     ; load higher byte into the high Z register

                rcall display_message    ; send data to LCD
```

```
                ldi R16, 0x00           ; resets remainder to 0
                ldi R17, 0x00           ; resets division result to 0
                rcall divide          ; calls the divide subroutine to seperate the two characters of the PWM counter

                mov R31, R17            ; moves tens place into R31 to be displayed
                rcall display_data      ; displays the character in R31 to the LCD
                mov R31, R16            ; moves ones place into R31 to be displayed
                rcall display_data      ; displays the character in R31 to the LCD
        line_two:
                ldi R24, 40             ; sets the loop counter to loop over every character in the string
                ldi R30, LOW(2*msg2)    ; splits the message into 8 bit parts
                ldi R31, HIGH(2*msg2)
                rcall display_message   ; loops over all the characters to display them to the LCD

        ldi R20, 0x00
        cpi R22, 0x01                   ; if R22 is 0x01, we want to show the fan is on, otherwise off
        breq display_on
        display_off:
                ldi R30, LOW(2*off)     ; display -> "Fan: OFF"
                ldi R31, HIGH(2*off)
                ldi R16, 0x00
                sts TCCR2A, R16
                rjmp display_end
        display_on:
                ldi R30, LOW(2*on)       ; display -> "Fan: ON"
                ldi R31, HIGH(2*on)
                ldi R16, 0x23            ; enable fast PWM and non-inverting
                sts TCCR2A, R16
        display_end:
                ldi R24, 10              ; load R24 with the number of characters in the 'on' or 'off' strings
                rcall display_message
        sei


/*
 * the main_loop will be what checks flags, while being able to have interrupts occur
 */
main_loop:
        cpi R21, 0x01                       ; R21 determines if the display needs to be updated
        breq display_lcd

        cpi R20, 0x01                       ; if R20 is 1, then we want the clockwise motion
        breq clockwise_move

        cpi R20, 0x02                       ; if R20 is 2, then we want the counter-clockwise motion
        breq counter_clockwise_move

        rjmp main_loop                      ; continue looping
clockwise_move:
        cli
        push R23

        cpi R18, 0x04                       ; if R18 is less than 4, we need to prevent the program from breaking
        brlt increment_fix

        inc R18                 ; 1% duty cycle increase
        cpi R18, 0x64
        brge stop_incrementing  ; if at 100, we want to stop increasing the duty cycle

        lds R16, OCR2B          ; load value of OCR2B into R16
        inc R16                 ; increase by 2
        inc R16
        sts OCR2B, R16          ; set new value in R16 back out to OCR2B, this increase the duty cycle of the PWM

        rcall delay_100ms
        ldi R20, 0x00           ; resets the RPG rotation indicator flag
        pop R23                ; restore R23
        rjmp display_lcd        ; jumps to display_lcd to update the LCD with the new numbers
```

```
        sei
        ret
counter_clockwise_move:
        cli
        push R23                        ; save the value of R23 on the stack

        cpi R18, 0x01                   ; if R18 hits one, we need to stop decreasing on the display
        breq decrement_fix

        dec R18                         ; 1% duty cycle decrease
        cpi R18, 0x05     ; once we get below 5, we need to stop decreasing the PWM, but keep decreasing the display
        brlt stop_decrementing

        lds R16, OCR2B                  ; load value of OCR2B into R16
        dec R16
        dec R16
        sts OCR2B, R16              ; set new value in R16 back out to OCR2B, this decreases the duty cycle of the PWM

        rcall delay_100ms
        ldi R20, 0x00               ; resets the RPG rotation indicator flag
        pop R23                     ; restore R23
        rjmp display_lcd            ; jumps to display_lcd to update the LCD with the new numbers

        sei
        ret
stop_incrementing:
        ldi R20, 0x00                   ; reset RPG indicator
        ldi R16, 200                    ; max the PWM for the fan
        ldi R18, 100                    ; max the display value for LCD
        sts OCR2B, R16                  ; load value back into OCR2B
        pop R23                         ; remove from stack
        rjmp display_lcd                ; update display
increment_fix:
        inc R18                         ; increases only the display value
        pop R23
        rjmp display_lcd                ; update display
stop_decrementing:
        ldi R20, 0x00                   ; reset RPG indicator
        ldi R16, 10                     ; min value we can set for the PWM
        sts OCR2B, R16
        pop R23
        rjmp display_lcd            ; update display
decrement_fix:
        ldi R18, 0x01               ; stop counter at 1
        pop R23
        rjmp display_lcd            ; update display

/*
 * this subroutine is responsible for sending the contents of the Z register to the LCD
 */
display_message:
        cli
        dm_inner_loop:
                lpm                 ; loads one byte from the Z register into a destination register
                swap R0             ; swap the nibbles in R0 and outputs them
                out PORTC, R0
                rcall pulse_lcd

                rcall delay_1ms             ; delay

                swap R0                     ; swap the nibbles in r0 and outputs them
                out PORTC, R0
                rcall pulse_lcd

                rcall delay_1ms             ; delay

                adiw zh:zl, 1               ; adds 1 to the Z-pointer register pair
```

```
                dec R24
                brne dm_inner_loop  ; branches back to the start if R24 is 0

        ldi R21, 0x00                       ; set update display flag to 0
        sei
        ret

display_data:
        cli

        swap R31                            ; swaps the nibbles of R31
        out PORTC, R31                      ; send upper nibble
        rcall pulse_lcd
        rcall delay_1ms                     ; delay

        swap R31                            ; swaps the nibbles of R31
        out PORTC, R31                      ; send lower nibble
        rcall pulse_lcd
        rcall delay_1ms                     ; delay

        sei
        ret

pulse_lcd:
        sbi PORTB, 3                        ; enable high
        rcall delay_200us           ; delay for a small amount
        cbi PORTB, 3                        ; enable low
        ; we need to delay ~2ms in order for the LCD to properly send and display data correctly
        rcall delay_1ms
        rcall delay_1ms
        ret

write_command:
        cli
        cbi PORTB, 5          ; sets register select low

        swap R29           ; upper nibble -> LCD
        out PORTC, R29
        rcall pulse_lcd

        swap R29           ; lower nibble -> LCD
        out PORTC, R29
        rcall pulse_lcd

        sei
        ret
/*
 * the divide subroutine uses the restoring divison algorithm in order to compute
 * the result and remainder values separately.
 */
divide:
        cli
        ldi R19, 10           ; 10 is the divisor
        mov R23, R18          ; copy to R23 to be pushed on the stack
        push R23
        sub     R16, R16            ; clear remainder and carry
        ldi     R29, 9                       ; loop counter

        divide_one:
                rol R17
                rol     R18
                dec     R29               ; decrement counter
                brne divide_two

                ldi R23, 0x30             ; will be added to the result and remainder to convert to ASCII value
                add R16, R23
                add R17, R23
                pop R23                   ; take from stack and move new value into R18
```

```
                mov R18, R23
                ret
        divide_two:
                rol     R16             ; put dividend into remainder
                sub     R16, R19        ; remainder = remainder - divisor
                brcc divide_three       ; if negative
                add     R16, R19        ; restore remainder
                clc                     ; clear carry to be shifted into result
                rjmp divide_one
        divide_three:
                sec                     ; set carry
                rjmp divide_one
        sei
        ret

/*
 * this subroutine is called from the button interrupt, will turn fan either on or off
 */
toggle_fan:
        cli                     ; clear interrupt enable flag

        sbis PIND, 3              ; check if the fan is on
        rjmp fan_on               ; if off, turn it on
        rjmp fan_off              ; if on, turn it off

        fan_on:
                sbi PORTD, 3
                ldi R22, 0x01     ; set display flag so that the LCD will update in main_loop
                rjmp toggle_end   ; jump to end
        fan_off:                  ; set R22 indicating the fan needs to be turned off in the main_loop
                cbi PORTD, 3
                ldi R22, 0x00     ; set display flag so that the LCD will update in main_loop
        toggle_end:
                ldi R21, 0x01
        reti

/*
 * The following subroutines determine which actions to take depending on
 *  the interrupt call for the RPG and checks the direction
 */
check_rpg:
        cli
        in R27, PIND                    ; prevents the RPG from changing the PWM if the fan is off
        sbis PIND, 3
        rjmp end_rpg

        sbis PIND, 7                    ; determines if the direction is CW
        rcall clockwise

        sbis PIND, 6                    ; determines if the direction is CCW
        rcall counterclockwise

        end_rpg: reti
clockwise:
        ldi R20, 0x00                   ; resets direction flag
        cpi R18, 0x64                   ; if R18 is 100 go to the end since it's at the upper bound
        breq end_cw

        ldi R20, 0x01                   ; set direction flag for RPG (CW)
        end_cw: reti

counterclockwise:
        ldi R20, 0x00                   ; resets direction flag
        cpi R18, 0x01                   ; if R18 is 1 go to the end since it's at the lower bound
        breq end_ccw

        ldi R20, 0x02                   ; set direction flag for RPG (CCW)
        end_ccw: reti
```

```
/*
 * delay subroutines
 *  - the 100ms and 5ms loops are made from the 1ms timer
 *  - the 200us is made fromt he 50us timer
 */
delay_1ms:
          ldi R25, 0x03
          out TCCR0B, R25
          ldi R25, 0x06
          out TCNT0, R25
loop_1ms:
          in R25, TCNT0
          cpi R25, 0x00
          brne loop_1ms
          ret

delay_5ms:
          ldi R26, 0x05
loop_5ms:
          rcall delay_1ms
          dec R26
          cpi R26, 0x00
          brne loop_5ms
          ret

delay_100ms:
          ldi R26, 0x64
loop_100ms:
          rcall delay_1ms
          dec R26
          cpi R26, 0x00
          brne loop_100ms
          ret

delay_50us:
          ldi R25, 0x02
          out TCCR0B, R25
          ldi R25, 0x9C
          out TCNT0, R25
loop_50us:
          in R25, TCNT0
          cpi R25, 0x00
          brne loop_50us
          ret

delay_200us:
          ldi R26, 0x04
loop_200us:
          rcall delay_50us
          dec R26
          cpi R26, 0x00
          brne loop_200us
          ret
.exit
```