Alexander Powers and Benjamin Mitchinson
ECE:3360
Spring 2019

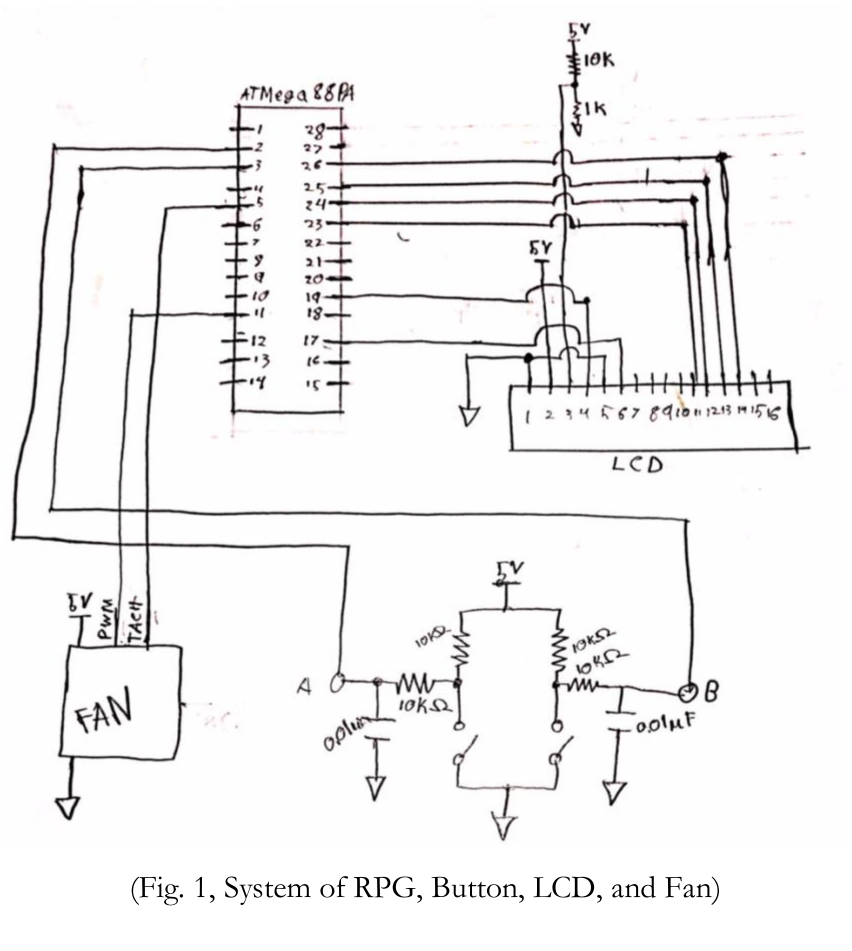
Embedded Systems Post Lab Four Report

1. Introduction

Lab four was a culmination of all of the skills I have developed over the last three labs. Lab four linked together four independent hardware components (LCD, Fan, Button, RPG) to create one functional fan speed monitoring system. This system consisted of two modes (A and B) that were toggled between by the positive edge of a button push. The fan's speed was controlled by pulse width modulation (PWM), which was controlled by the rotary pulse generator (RPG). These components all came together to allow us to display diagnostic information (LOW-RPM, OK, ALARM) and the duty cycle of the PWM on a liquid crystal display (LCD).

2. Schematic

Figure 1 shows the circuit design used to connect all of the hardware sub-component together. These components were all linked together through the ATmega88PA micro-controller.



(Fig. 1, System of RPG, Button, LCD, and Fan)

3. Discussion

In order to implement this system, we had to isolate each component and validate/test its functionality in the overall system. The first component we did was the rotary pulse generator. Utilizing the same techniques discussed in lab three, we implemented the input reading and interpretation of the RPG. This naturally led us to the pulse width modulation portion of this lab. We utilized the same code as lab three, but modified it by placing the wave generation into its own ISR. We then connected the fan to the output of our PWM signal, so that the fan speed would vary proportionally to the PWM. Having completed that half of the system, we moved on to the LED and interpreting the tachometer output. In order to interpret the tachometer signal, we create two independent ISRs. One ISR triggered on the positive edge of the tachometer signal, while the other ISR was a 16-bit timer that controlled the period that we were sampling. In combination, these ISRs allowed us to compute the frequency of the fan speed. The final component, and certainly the most complex was the LCD. The process of configuring the LCD, formatting registers for display, and working through the different LCD commands left us overwhelmed at the start. The code provided for lecture allowed us to get a jump start on the LCD configuration and constant string writing, so the only major component left was formatting the duty cycle to display as a percentage. Due to the fact that our max duty cycle register value was 0xC8 or 200, we were able to use the provided division library to simply divide by 2 to get the percentage. With a little more branching logic, we had successfully completed lab four.

Implemented Features:

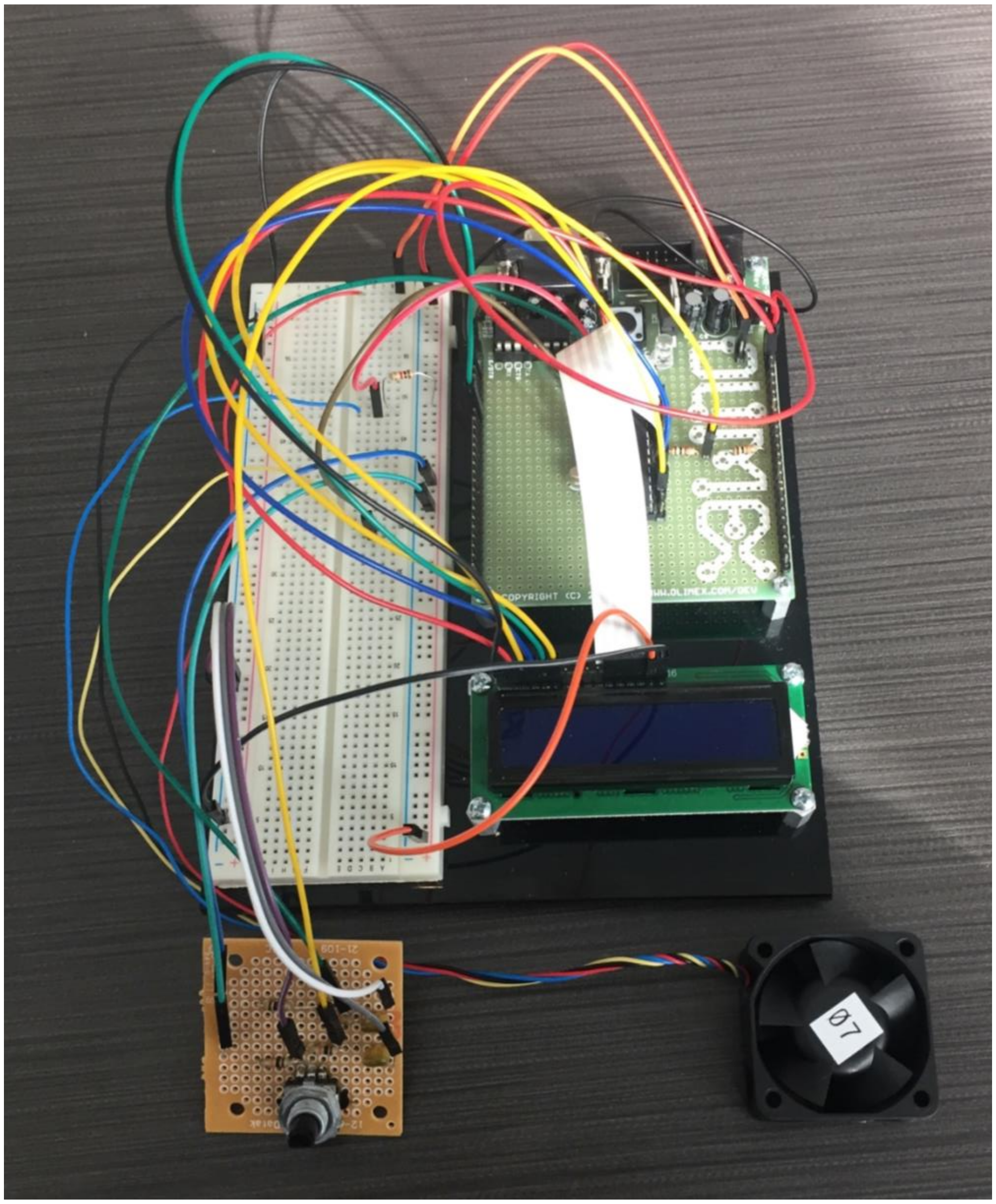
Button Controlled Mode Toggling	X
PWM Controlled Fan	X
Tachometer Controlled Mode Message	X
RPG Controlled PWM	X
Duty Cycle Percentage Display	X
Proper LCD Formatting and Writing	X

4. Conclusion

This lab was a great capstone to our work in assembly this semester. It touched on all of the concepts from labs one two and three, while still adding interesting and powerful new concepts, such as interrupts. I feel as though this lab has prepared me to develop the software and hardware configurations that are used in industry settings. I think that the power of interrupt driven programming will be my largest take away from this lab. The responsiveness and flexibility that interrupt driven programs allow for is unparalleled by standard sequential execution models.

5. Appendix

Appendix A: Image of circuit implemented in the above schematic



Appendix B: Source Code from main.asm

```
///////////////////////////////////////////////////////////////////////

// Assembly language file for Lab 4 in ECE:3360 (Embedded Systems)

// Spring 2019, The University of Iowa.

//

// Desc: Lab 4 Rotary Pulse Generator Controlled Fan with LCD Display

//

// Authors: B. Mitchinson, A. Powers

///////////////////////////////////////////////////////////////////////


// definitions and preprocessor directives

//===========================================================================
====================


// .inc include files

// //////////////////

.include "m88PAdef.inc"

// //////////////////


// Allocate memory in DSEG for dynamic string

///////////////////////////////////////////////////////////////////////

.dseg

        active: .BYTE 5

///////////////////////////////////////////////////////////////////////


// the rest of the code goes in cseg

///////////////////////////////////////////////////////////////////////

.cseg
```

```
//////////////////////////////////////////////////////////////////////

.org 0x00

// skip interrupt

rjmp skip_interrupt


// interupt methods

// ////////////////////////////////////////////////////////////////////

.org 0x001

        rjmp button_interrupt


.org 0x002

        rjmp tach_interrupt


.org 0x010      ; PC points here on timer 0

        rjmp tim0_ovf ;   over flow interrupt


.org 0x00D

        rjmp tim1_ovf


.org 0x1a

skip_interrupt:


// interrupt config

// ////////////////////////////////////////////////////////////////////

ldi r16, 0x0F

sts EICRA, r16

ldi r16, 0x03

out EIMSK, r16
```

```
lds r16, TIMSK0
ori r16, 0x01       ; Overflow interrupt enable
sts TIMSK0, r16

lds r16, TIMSK1
ori r16, 0x01       ; Overflow interrupt enable
sts TIMSK1, r16

ldi r16, 0x00

// set port numbers
///////////////////////////////////////////////////////////////////////
.equ led_port = 5
.equ pwm_port = 5

// LCD config
///////////////////////////////////////////////////////////////////////
.equ RS = 5
.equ E = 3
.equ D4 = 0
.equ D5 = 1
.equ D6 = 2
.equ D7 = 3
.def data_reg = r19

// set DDRB/C
///////////////////////////////////////////////////////////////////////
```

```
sbi DDRC, led_port

sbi DDRD, pwm_port

sbi DDRC, D4

sbi DDRC, D5

sbi DDRC, D6

sbi DDRC, D7

sbi DDRB, RS

sbi DDRB, E

cbi DDRD, 2


// variables for current and previous state of the rpg

/////////////////////////////////////////////////////////////////////////////////
//

.def current_state = r16

.def previous_state = r17

.def duty_reg = r18

.equ upper_cycle_limit = 198

.equ lower_cycle_limit = 1

.equ half_duty_cycle = 100

ldi duty_reg, half_duty_cycle


.equ BOTH_ON = 0x03

.equ A_ON = 0x01

.equ B_ON = 0x02

.equ BOTH_OFF = 0x00

// 0 - both on   0b00000000

// 1 - clockwise  0b00000001

// 2 - counter    0b00000010
// 3 - both off    0b00000011

// A is bit position 1

// B is bit position 0

//////////////////////////////////////////////////////////////////////////////
//



// configure PWM

// //////////////////////////////////////////////////////////////////////

push r29

ldi r29, 0x23 // 0 0 1 0 0 0 1 1 (Compare to non invert + mode to 7)

out TCCR0A, r29

ldi r29, 0x09 // 0 0 0 0 1 0 0 1 (mode to 7 + prescale of 1)

out TCCR0B, r29


ldi r29, 199 // set TOP of counter -- i.e. where the TOVO bit gets set

out OCR0A, r29


out OCR0B, duty_reg // Set PWM flip point, OCR0B is the pwm active reg

pop r29

// //////////////////////////////////////////////////////////////////////


// Configure Tachometer Sampling

// //////////////////////////////////////////////////////////////////////

push r29

ldi r29, 0x00 // 0 0 0 0 0 0 0 0 (Compare to non invert + mode to 0)

sts TCCR1A, r29

ldi r29, 0x03 // 0 0 0 0 0 0 1 1 (mode to 0 + prescale of 64) # 65535 clock ticks
```

```
sts TCCR1B, r29

pop r29

// ///////////////////////////////////////////////////////////////////


// Mode Configuration Variables

// /////////////////////////////

.def mode_reg = r24

.def pos_tach_reg = r25

.def tach_save_reg = r26

// mode_reg meanings

// 0x00 --> mode a

// 0x01 --> mode b

.equ mode_a_thresh = 1 // threshold for mode a

.equ mode_b_thresh = 21 // threshold for mode b

// /////////////////////////////



// Tables

// ///////////////////////////////////////////////////////////////////

rjmp table_skip // skip table execution

msg_dc: .db "DC =      %", 0x00

msg_0: .db "DC =   0.0%", 0x00

msg_100: .db "DC = 100.0%", 0x00

msg_a: .db "Mode A:", 0x00

msg_b: .db "Mode B:", 0x00

msg_ok:      .db "OK      ", 0x00

msg_low_rpm: .db "LOW RPM", 0x00

msg_alarm:   .db "ALARM  ", 0x00
```

```
table_skip:
// ///////////////////////////////////////////////////////////////////


// initailize the LCD to 8 bit mode
// ////////////////////////////////
push data_reg
rcall lcd_init
pop data_reg
// ////////////////////////////////


// initial top display:
// ///////////////////
ldi R30, LOW(2*msg_dc)
ldi R31, HIGH(2*msg_dc)
sbi PORTB, RS
rcall displayCString
// ///////////////////


// initial display of the mode
rcall disp_mode


sei
// Main loop
// /////////////////////////////////////////////////////////////////////
main:
        // RPG Sub-Methods
        rcall read_rpg // Uses 16, 17
        rcall which_direction // 16, 17, 18
```

```
// push registers
push r14
push r15
push r16
push r17
push r19
push r20
push r21
push r22
push r23
cli
rcall update_duty_display // preps 25 and 26 for display and displays DC = xx.x%
sei
// pop registers
pop r23
pop r22
pop r21
pop r20
pop r19
pop r17
pop r16
pop r15
pop r14
sei
nop
nop
nop
```

```
	cli

	rcall update_mode_display

	sei


	rjmp main

// ////////////////////////////////////////////////////////////////////////


update_mode_display:

	rcall move_to_9_pos_bottom

	cpi mode_reg, 0x00

	breq mode_a_helper

	rjmp mode_b_helper


mode_a_helper:

	cpi tach_save_reg, mode_a_thresh

	brsh disp_ok

	rjmp disp_alarm


mode_b_helper:

	cpi tach_save_reg, mode_b_thresh

	brsh disp_ok

	rjmp disp_low_rmp


disp_ok:

	rcall disp_mode_ok

	rjmp update_mode_display_end


disp_low_rmp:
        rcall disp_mode_low_rpm
        rjmp update_mode_display_end

disp_alarm:
        rcall disp_mode_alarm
        rjmp update_mode_display_end

update_mode_display_end:
        ret

disp_mode_ok:
        ldi R30, LOW(2*msg_ok)
        ldi R31, HIGH(2*msg_ok)
        sbi PORTB, RS
        rcall displayCString
        ret

disp_mode_low_rpm:
        ldi R30, LOW(2*msg_low_rpm)
        ldi R31, HIGH(2*msg_low_rpm)
        sbi PORTB, RS
        rcall displayCString
        ret

disp_mode_alarm:
        ldi R30, LOW(2*msg_alarm)
        ldi R31, HIGH(2*msg_alarm)
        sbi PORTB, RS
```

```
	rcall displayCString

	ret


// display the current mode based on mode reg

// ////////////////////////////////////////////////////////////////////////

disp_mode:

	rcall move_to_bottom

	cpi mode_reg, 0x00

	breq disp_a

	cpi mode_reg, 0x01

	breq disp_b

	// mode reg is in and invalid state (reset to mode a)

	ldi mode_reg, 0x00

	rjmp disp_mode_end


disp_a:

	rcall disp_mode_a

	rjmp disp_mode_end


disp_b:

	rcall disp_mode_b

	rjmp disp_mode_end


disp_mode_end:

	ret


// ////////////////////////////////////////////////////////////////////////
```

```
disp_mode_a:
        ldi R30, LOW(2*msg_a)
        ldi R31, HIGH(2*msg_a)
        sbi PORTB, RS
        rcall displayCString
        ret

disp_mode_b:
        ldi R30, LOW(2*msg_b)
        ldi R31, HIGH(2*msg_b)
        sbi PORTB, RS
        rcall displayCString
        ret

// tack increment interrupt
tach_interrupt:
        cpi pos_tach_reg, 0xFE
        brsh tach_int_end
        inc pos_tach_reg
tach_int_end:
        reti

// timer overflow interrupt
tim0_ovf:
        push r21
        push r20
  // Stop timer 0
  in r20, TCCR0B //
```

```
    ldi r21, 0x00
    out TCCR0B, r21

    // Clear over flow flag
    in r21, TIFR0
    sbr r21, 1<<TOV0
    out TIFR0, r21

    // Start timer with new initial count
            push r29
            ldi r29, 0x00
    out TCNT0, r29 // starting point of timer
    out TCCR0B, r20
            pop r29
            pop r20
            pop r21
            reti

tim1_ovf:
            push r21
            push r20
    // Stop timer 0
    lds r20, TCCR1B //
    ldi r21, 0x00
    sts TCCR1B, r21

    // Clear over flow flag
    in r21, TIFR1
```

```
   sbr r21, 1<<TOV1

   sts TIFR1, r21


   // Start timer with new initial count

           push r29

           ldi r29, 0x00

   sts TCNT1H, r29 // starting point of timer

           sts TCNT1L, r29 // starting point of timer

   sts TCCR1B, r20

           pop r29

           pop r20

           pop r21


           mov tach_save_reg, pos_tach_reg

           ldi pos_tach_reg, 0x00

           reti


// btn interrupt

button_interrupt:

           push r16

           push r30

           push r31

           push r0

           push data_reg


           in r16, SREG


           inc mode_reg
```

```
	cpi mode_reg, 0x02
	brlo mode_end
	ldi mode_reg, 0x00
	mode_end:

	rcall disp_mode

	out SREG, r16

	pop data_reg
	pop r0
	pop r31
	pop r30
	pop r16

	reti

// returns the cursor to the first cell of the first row
// ///////////////////////////////////////////////////////////////////////
display_home:
	cbi PORTB, 5
	ldi data_reg, 0x08
	out PORTC, data_reg
	rcall lcd_strobe
	rcall delay_200_us
	ldi data_reg, 0x00
	out PORTC, data_reg
	rcall lcd_strobe
```

```
        rcall delay_200_us
        ret
// ///////////////////////////////////////////////////////////////////////



// move to the 5th position on the first line for variable strings
// ///////////////////////////////////////////////////////////////////////
move_to_5_pos:
        cbi PORTB, 5
        ldi data_reg, 0x08
        out PORTC, data_reg
        rcall lcd_strobe
        rcall delay_200_us
        ldi data_reg, 0x05
        out PORTC, data_reg
        rcall lcd_strobe
        rcall delay_200_us
        ret
// ///////////////////////////////////////////////////////////////////////

// move cursor to the lower line
// ///////////////////////////////////////////////////////////////////////
move_to_bottom:
        cbi PORTB, 5
        ldi data_reg, 0x0C
        out PORTC, data_reg
        rcall lcd_strobe
        rcall delay_200_us
```

```
	ldi data_reg, 0x00
	out PORTC, data_reg
	rcall lcd_strobe
	rcall delay_200_us
	ret

move_to_9_pos_bottom:
	cbi PORTB, 5
	ldi data_reg, 0x0C
	out PORTC, data_reg
	rcall lcd_strobe
	rcall delay_200_us
	ldi data_reg, 0x09
	out PORTC, data_reg
	rcall lcd_strobe
	rcall delay_200_us
	ret
// ///////////////////////////////////////////////////////////////////////////


// Subroutine to check for edge cases --> 0% and 100%
// ///////////////////////////////////////////////////////////////////////////
duty_edge_cases:
	cpi duty_reg, 0xC6  // compare 200
	brsh duty_hundo     // jump to 100%
	cpi duty_reg, 0x02  // compare 2
	brlo duty_zero      // jump to 0%
	rjmp no_edge_return
```

```
// //////////////////////////////////////////////////////////////////////////



// subroutine to display the entire top line as the 100.0% string
// //////////////////////////////////////////////////////////////////////////
duty_hundo:
        ldi R30, LOW(2*msg_100)
        ldi R31, HIGH(2*msg_100)
        rcall displayCString
        rjmp duty_display_end
// //////////////////////////////////////////////////////////////////////////



// subroutine to display the entire top line as the 0.0% string
// //////////////////////////////////////////////////////////////////////////
duty_zero:
        ldi R30, LOW(2*msg_0)
        ldi R31, HIGH(2*msg_0)
        rcall displayCString
        rjmp duty_display_end
// //////////////////////////////////////////////////////////////////////////



// main subroutine for updating the display based on the duty cycle
// //////////////////////////////////////////////////////////////////////////
update_duty_display:
        rcall display_home
        rjmp duty_edge_cases    // check for edge cases before doing computations
```

```
no_edge_return:            // tag to return to if neither edge case is met
   rcall move_to_5_pos    // move the cursor to the correct position to write
         push r25
   push r26


// register defenitions for division
/*
.def     drem16uL=r14
.def     drem16uH=r15
.def     dres16uL=r16
.def     dres16uH=r17
.def     dd16uL         =r16
.def     dd16uH         =r17
.def     dv16uL=r23
.def     dv16uH         =r19
.def     dcnt16u        =r20
*/


   // divide duty reg by 2
         mov dd16uL, duty_reg     // LSB of number to display
         ldi r26, 0x00
         mov dd16uH, r26          // MSB of number to display
         ldi dv16uL, low(2)       // divide by 2
         ldi dv16uH, high(2)
         rcall div16u


         // Store terminating for the string.
         ldi r20,0x00             // Terminating NULL
```

```
sts active+4,r20         // Store in RAM


mov r26, r14             // save off r14 into r26
cpi r26, 0x00            // check if it is zero
breq zero_dec            // move to place a zero iff r26 == 0x00
ldi r25, 0x35            // load r25 with a '5' string
rjmp end_dec             // jmp to the end so r25 isn't overwritten
zero_dec:
ldi r25, 0x30            // load r25 with a '0' string
end_dec:
sts active+3,r25         // store the resulting r25 into dseg


// Generate decimal point.
ldi r20,0x2e       // ASCII code for .
sts active+2,r20     // Store in RAM


// get the tens position of the percentage
mov dd16uL, r16
mov dd16uH, r17
ldi dv16uL, low(10)
ldi dv16uH, high(10)
rcall div16u


// offset the resulting tens position by 48 to get it to ASCII
ldi r20,0x30
add r14, r20
sts active+1,r14
```

```
	// offset the resulting 100s position by 48 to get it to ASCII
	ldi r20,0x30
	add r16, r20
	sts active+0,r16

	// display the updated string
	rcall displayDString

	pop r26
	pop r25
duty_display_end: // end tag for edge case methods to jump to
	ret
// ////////////////////////////////////////////////////////////////////////

// subroutine to display a constant string
// ////////////////////////////////////////////////////////////////////////
displayCString:		// Prints whatever is in Z
	sbi PORTB, RS
	lpm r0,Z+		// <-- first byte
	tst r0			// Reached end of message ?
	breq doneC		// Yes => quit
	swap  r0		// Upper nibble in place
	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 displayCstring
```

```
doneC:

        sei

        nop

        nop

        nop

        nop

        nop

        cli

        ret

// ///////////////////////////////////////////////////////////////////



// subroutine to display dynamic string --> effectively the same as displayCString

// but it load from dseg and not from a constant string

// ///////////////////////////////////////////////////////////////////

displayDString:         // Prints whatever is in Z

        sbi PORTB, RS

        ldi r30, LOW(active)

        ldi r31, HIGH(active)

progressDString:

        ld r0,Z+              // <-- first byte

        tst r0               // Reached end of message ?

        breq doneD              // Yes => quit

        swap  r0              // Upper nibble in place

        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 progressDString
doneD:
	ret

// toggle enable off - on - off
/// ////////////////////////////////////////////////////////////////////
lcd_strobe:
	cbi PORTB, E
	rcall delay_200_us
	sbi PORTB, E
	rcall delay_200_us
	cbi PORTB, E
	ret
/// ////////////////////////////////////////////////////////////////////


// initialize the lcd to 4-bit mode and send configuration commands
// ////////////////////////////////////////////////////////////////////
lcd_init:
	cbi PORTB, RS	// set to command mode
	rcall delay_10_ms
	rcall set_to_8_bit_mode
	rcall set_to_8_bit_mode
	rcall set_to_8_bit_mode
	rcall set_to_4_bit_mode
	rcall set_to_4_bit_mode
```

```
ldi data_reg, 0x08 // Two rows 5x7 characters
out PORTC, data_reg
rcall lcd_strobe
rcall delay_200_us

ldi data_reg, 0x00 // clear display
out PORTC, data_reg
rcall lcd_strobe
rcall delay_200_us

ldi data_reg, 0x01 // also clear display
out PORTC, data_reg
rcall lcd_strobe
rcall delay_10_ms

ldi data_reg, 0x00 // display on, underline + blink off
out PORTC, data_reg
rcall lcd_strobe
rcall delay_200_us

ldi data_reg, 0x0c // also display on, underline + blink off
out PORTC, data_reg
rcall lcd_strobe
rcall delay_200_us

ldi data_reg, 0x00 // display shift off, address increment mode
out PORTC, data_reg
rcall lcd_strobe
```

```
	rcall delay_200_us


	ldi data_reg, 0x06 // also display shift off, address increment mode
	out PORTC, data_reg
	rcall lcd_strobe
	rcall delay_200_us


	ret
// ///////////////////////////////////////////////////////////////////////



// sends the bit 0x03 as a command nibble
// ///////////////////////////////////////////////////////////////////////
set_to_8_bit_mode:
	ldi data_reg, 0x03
	nop
	out PORTC, data_reg
	rcall lcd_strobe
	rcall delay_200_us
	nop
	ret
// ///////////////////////////////////////////////////////////////////////



// sends the bit 0x02 as a command nibble
// ///////////////////////////////////////////////////////////////////////
set_to_4_bit_mode:
	ldi data_reg, 0x02
```

```
        nop
        out PORTC, data_reg
        rcall lcd_strobe
        rcall delay_200_us
        nop
        ret
// //////////////////////////////////////////////////////////////////////


// 10ms, and 200us delays for LCD initialization
////////////////////////////////////////////////////////////////////////
delay_10_ms:
         push r23
         push r24
         push r25
         nop
         ldi r23, 2       // r23 <-- Counter for outer loop
 d21: ldi r24, 44     // r24 <-- Counter for level 2 loop
 d22: ldi r25, 227    // r25 <-- Counter for inner loop
 d23: dec r25
     nop            // no operation
     brne d23
     dec r24
     brne d22
     dec r23
     brne d21
         nop
         pop r25
         pop r24
```

```
            pop r23
    ret


delay_200_us:
            push r23
            push r24
            push r25
            nop
            ldi r23, 1        // r23 <-- Counter for outer loop
  d31: ldi r24, 5       // r24 <-- Counter for level 2 loop
  d32: ldi r25, 100    // r25 <-- Counter for inner loop
  d33: dec r25
    nop              // no operation
    brne d33
    dec r24
    brne d32
    dec r23
    brne d31
            nop
            pop r25
            pop r24
            pop r23
    ret


// RPG sub-routines
//////////////////////////////////////////////////////////////////////
read_rpg:
   nop
```

```
    push r28
    push r29
    ldi r28, 0x01
    ldi r29, 0x02

    mov previous_state, current_state
    ldi current_state, 0x00
    sbis PIND, 0
    add current_state, r29 // run if a is high
    sbis PIND, 1
    add current_state, r28 // run if b is high
    pop r29
    pop r28
    ret

which_direction:
    nop
    cpi previous_state, BOTH_ON
    breq which_end

    // if current state low
    cpi current_state, BOTH_OFF
    breq current_low
    rjmp which_end

    current_low:
    cpi previous_state, A_ON
    breq counter_clockwise
    cpi previous_state, B_ON
    breq clockwise
    rjmp which_end


    which_end:
    ret


clockwise:
    inc duty_reg
    cpi duty_reg, upper_cycle_limit
    brsh recover_upper
    rjmp end_cwise
    recover_upper:
    ldi duty_reg, upper_cycle_limit
    end_cwise:
            out OCR0B, duty_reg
    ret


counter_clockwise:
    dec duty_reg
    cpi duty_reg, lower_cycle_limit
    brsh end_ccwise
    ldi duty_reg, lower_cycle_limit
    end_ccwise:
            out OCR0B, duty_reg
    ret
//////////////////////////////////////////////////////////////////////////
```

```
// method taken frome external library
//***************************************************************************
//*
//* "div16u" - 16/16 Bit Unsigned Division
//*
//* This subroutine divides the two 16-bit numbers
//* "dd8uH:dd8uL" (dividend) and "dv16uH:dv16uL" (divisor).
//* The result is placed in "dres16uH:dres16uL" and the remainder in
//* "drem16uH:drem16uL".
//*
//* Number of words :19
//* Number of cycles :235/251 (Min/Max)
//* Low registers used:2 (drem16uL,drem16uH)
//* High registers used :5 (dres16uL/dd16uL,dres16uH/dd16uH,dv16uL,dv16uH,
//*                         dcnt16u)
//*
//***************************************************************************

//***** Subroutine Register Variables

.def    drem16uL=r14
.def    drem16uH=r15
.def    dres16uL=r16
.def    dres16uH=r17
.def    dd16uL      =r16
.def    dd16uH      =r17
```

```
.def    dv16uL=r23
.def    dv16uH          =r19
.def    dcnt16u         =r20


//***** Code


div16u: clr     drem16uL        //clear remainder Low byte
        sub     drem16uH,drem16uH  //clear remainder High byte and carry
        ldi     dcnt16u,17              //init loop counter
d16u_1: rol dd16uL      //shift left dividend
        rol     dd16uH
        dec     dcnt16u                 //decrement counter
        brne    d16u_2          //if done
        ret                             //return
d16u_2:         rol     drem16uL        //shift dividend into remainder
        rol     drem16uH
        sub     drem16uL,dv16uL         //remainder = remainder - divisor
        sbc     drem16uH,dv16uH         //
        brcc    d16u_3          //if result negative
        add     drem16uL,dv16uL         //restore remainder
        adc     drem16uH,dv16uH
        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
.exit
```