ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
ΑΚΑΔ. ΕΤΟΣ 2022-2023

Αναφορά

4η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ ΓΙΑ ΤΟ ΜΑΘΗΜΑ "Εργαστήριο Μικροϋπολογιστών"

Ανάπτυξη κώδικα για το μικροελεγκτή ATmega328PB και προσομοίωση της εκτέλεσης του στο αναπτυξιακό περιβάλλον MPLAB X

Ομάδα 1

Ξανθόπουλος Παναγιώτης (03119084)

Παπαναστάσης Αθανάσιος (03113197

Ζήτημα 4.1

Στην άσκηση αυτή, έχουμε τον μετρητή και την ADC μετατροπή. Ο μετρητής υλοποιείται μέσω του χρονιστή 1, όπως στην 3η σειρά ασκήσεων. Όταν περάσει το 1 δευτερόλεπτο, καλείται η ρουτίνα εξυπηρέτησης διακοπής της υπερχείλισης του χρονιστή. Εκεί, ο μετρητής αυξάνεται κάθε φορά (αφού ελεγχθεί αν έφτασε στην μέγιστη τιμή) και η τιμή του φαίνεται στα LED της θύρας B. Τέλος, ανανεώνουμε την τιμή του χρονιστή και επιστρέφουμε.

Η ADC μετατροπή λειτουργεί ως εξής: Αρχικοποιούμε τους καταχωρητές λειτουργίας του ADC ώστε να τρέχει σε free running mode. Αυτό σημαίνει ότι αφότου ξεκινήσουμε την πρώτη μετατροπή (θέτοντας σε 1 το Bit ADSC του ADCSRA), οι επόμενες θα ξεκινήσουν αυτόματα και αμέσως.

Όταν ολοκληρωθεί μια μετατροπή, καλείται η ρουτίνα εξυπηρέτησης διακοπής του ADC. Εκεί, πολλαπλασιάζουμε με 5 (Vref) την τιμή του ADC (ADC_H : ADC_L) ώστε να έχουμε την πραγματική τιμή επί 1024. Τα bit 2-4 του ADC_H τώρα περιέχουν το ακέραιο μέρος της πραγματικής τιμής, οπότε τα αποθηκεύουμε. Πολλαπλασιάζουμε επί 10 οπότε στα bit 2-5 έχουμε το πρώτο δεκαδικό ψηφίο και το αποθηκεύουμε. Επαναλαμβάνουμε και αποθηκεύουμε και το δεύτερο δεκαδικό ψηφίο. Στο τέλος, μετατρέπουμε τα ψηφία σε ASCII τιμές και τις τυπώνουμε στην LCD. Έχουμε και μια καθυστέρηση 100ms ώστε να φαίνεται για λίγο το αποτέλεσμα στην οθόνη πριν κληθεί ξανά η lcd_init και σβηστεί.

Οι ρουτίνες για την lcd είναι δοσμένες από την εκφώνηση. Οι ρουτίνες καθυστέρησης είναι από προηγούμενες ασκήσεις.

```
 .include "m328PBdef.inc"

.def temp = r16
.def cnt = r17
.def integer = r18
.def dec1 = r19
.def dec2 = r20
.def ADC_L = r21
.def ADC_H = r22

.org 0x00
   jmp reset

.org 0x1A
   jmp timer1

.org 0x2A
   jmp adc_ready

reset:
   ldi temp, high(RAMEND)
   out SPH, temp
   ldi temp, low(RAMEND)
   out SPL,temp

   ldi temp, 0xFF
   out DDRB, temp
   out DDRD, temp
   ldi temp, 0x00
```

```
    out DDRC, temp

    ; REFSn[1:0]=01 => select Vref=5V, MUXn[4:0]=0010 => select ADC2(pin PC2),
    ; ADLAR=0 => right adjust the ADC result
    ldi temp, (1 << REFS0) | (1 << MUX1)
    sts ADMUX, temp

    ; ADEN=1 => ADC Enable, ADCS=0 => No Conversion yet
    ; ADIE=1 => enable adc interrupt, ADPS[2:0]=111 => fADC=16MHz/128=125KHz
    ldi temp, (1 << ADEN) | (1 << ADIE) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0) | (1 << ADATE)
    sts ADCSRA, temp

    ldi temp, 0x00
    sts ADCSRB, temp

    ldi temp, 0xFF
    ldi cnt, (1 << ADC2D)   ;cnt is used as dummy var
    eor temp, cnt
    sts DIDR0, temp  ;disable all other ADC inputs

    ldi temp,(1<<TOIE1)             ;enable timer1 overflow interrupt
    sts TIMSK1, temp

    ldi r26 ,(1<<CS12) | (0<<CS11) | (1<<CS10) ; CK/1024
    sts TCCR1B, r26

    ldi cnt, 0x00           ;initalize counter at 0
    out PORTB, cnt

    sei                             ;enable all interrupts

    lds temp, ADCSRA                ;start first conversion
    ori temp, (1<<ADSC)
    sts ADCSRA, temp

    ldi temp, HIGH(49910)    ;reset TCNT1
    sts TCNT1H, temp
    ldi temp, LOW(49910)
    sts TCNT1L, temp

main:
    rjmp main

write_2_nibbles:
    push r24                ;r24 contains data to be sent
    in r25, PIND
    andi r25, 0x0f
    andi r24, 0xf0
    add r24, r25
    out PORTD, r24
    sbi PORTD, 3
    nop
    nop
    cbi PORTD, 3
    pop r24
    swap r24
    andi r24, 0xf0
    add r24, r25
```

```
    out PORTD, r24
    sbi PORTD, 3
    nop
    nop
    cbi PORTD, 3
    ret


lcd_data:
    sbi PORTD, 2
    rcall write_2_nibbles
    ldi r24, 100
    ldi r25, 0
    rcall wait_usec
    ret


lcd_command:
    cbi PORTD, 2
    rcall write_2_nibbles
    ldi r24, 100
    ldi r25, 0
    rcall wait_usec
    ret


lcd_init:
    ldi r24, low(16*40)
    ldi r25, high(16*40)
    rcall wait_x_msec
    ldi r24, 0x30
    out PORTD, r24
    sbi PORTD, 3
    nop
    nop
    cbi PORTD, 3
    ldi r24, 100
    ldi r25, 0
    rcall wait_usec
    ldi r24, 0x30
    out PORTD, r24
    sbi PORTD, 3
    nop
    nop
    cbi PORTD, 3
    ldi r24, 100
    ldi r25, 0
    rcall wait_usec
    ldi r24, 0x20
    out PORTD, r24
    sbi PORTD, 3
    nop
    nop
    cbi PORTD, 3
    ldi r24, 100
    ldi r25, 0
    rcall wait_usec
    ldi r24, 0x28
```

```
    rcall lcd_command
    ldi r24, 0x0c
    rcall lcd_command
    ldi r24, 0x01
    rcall lcd_command
    ldi r24, low(5000)
    ldi r25, high(5000)
    rcall wait_usec
    ldi r24, 0x06
    rcall lcd_command
    ret


timer1:
    cpi cnt, 0x3F
    brne incr
    ldi cnt, 0x00
    rjmp outb
incr:
    inc cnt
outb:
    out PORTB, cnt
    ldi temp, HIGH(49910)    ;reset TCNT1
    sts TCNT1H, temp
    ldi temp, LOW(49910)
    sts TCNT1L, temp
    reti


adc_ready:
    lds ADC_L, ADCL
    lds ADC_H, ADCH

    rcall mul5
    mov integer, ADC_H
    lsr integer
    lsr integer                    ;integer contains now the integer part

    andi ADC_H, 0x03               ;delete integer part

    rcall mul5
    lsl ADC_L
    rol ADC_H
    mov dec1, ADC_H
    lsr dec1
    lsr dec1                       ;dec1 contains now the first decimal point

    andi ADC_H, 0x03               ;delete first decimal part

    rcall mul5
    lsl ADC_L
    rol ADC_H
    mov dec2, ADC_H
    lsr dec2
    lsr dec2                       ;dec2 contains now the second decimal point

    ori integer, 0x30  ; to ascii
    ori dec1, 0x30
```

```
    ori dec2, 0x30

    rcall lcd_init
    ldi r24, low(16*2)
    ldi r25, high(16*2)
    rcall wait_x_msec

    mov r24, integer
    call lcd_data

    ldi r24, '.'
    call lcd_data

    mov r24, dec1
    call lcd_data

    mov r24, dec2
    call lcd_data

    ldi r24, low(16*100)
    ldi r25, high(16*100)
    rcall wait_x_msec

    reti


mul5:
    mov r14, ADC_L
    mov r15, ADC_H

    lsl ADC_L
    rol ADC_H

    lsl ADC_L
    rol ADC_H

    add ADC_L, r14
    adc ADC_H, r15

    ret


wait_x_msec:
    rcall delay_986u
    sbiw r24, 1
    breq end

    rjmp help1

help1:
    rjmp help2              ;2 cc

help2:
    rjmp help3              ;2 cc

help3:
    rjmp wait_x_msec        ;2 cc
```

```
end:
   ret

delay_986u:
   ldi r26, 98

loop_986u:
   rcall wait_4u
   dec r26
   brne loop_986u

   nop
   nop
   ret

wait_4u:
   ret


wait_usec:
   sbiw r24 ,1
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   brne wait_usec
   ret
```

Η άσκηση σε C λειτουργεί με ελαφρώς διαφορετικό τρόπο. Αρχικά, δεν έχουμε μετρητή. Επίσης, περιμένουμε με polling να τελειώσει η μετατροπή ADC ελέγχοντας συνεχώς αν έγινε 0 το bit ADSC του καταχωρητή ADCSRA. Επίσης, κάθε φορά ξεκινάμε χειροκίνητα την μετατροπή, αφού διαβάσουμε και επεξεργαστούμε δηλαδή τα δεδομένα της προηγούμενης. Επίσης, χρησιμοποιούμε floating point operations για να βρούμε τα ψηφία. Τα υπόλοιπα κομμάτια είναι αντίστοιχα με την assembly.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

void write_2_nibbles(uint8_t c) {
   uint8_t temp = c;
   uint8_t prev = PIND;
   prev &= 0x0F;
   c &= 0xF0;
   c |= prev;
   PORTD = c;
   PORTD |= 0x08;
```

```
    PORTD &= 0xF7;

    c = temp;
    c &= 0x0F;
    c = c << 4;
    c |= prev;
    PORTD = c;
    PORTD |= 0x08;
    PORTD &= 0xF7;

    return;
}

void lcd_data(uint8_t c) {
    PORTD |= 0x04;
    write_2_nibbles(c);
    _delay_us(100);
    return;
}

void lcd_command(uint8_t c) {
    PORTD &= 0xFB;
    write_2_nibbles(c);
    _delay_us(100);
    return;
}

void lcd_init(void) {
    _delay_ms(40);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    PORTD = 0x20;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    lcd_command(0x28);
    lcd_command(0x0C);
    lcd_command(0x01);
    _delay_us(5000);

    lcd_command(0x06);
    return;
}

static volatile float ADC_val;
static volatile uint8_t integer;
static volatile uint8_t dec1;
static volatile uint8_t dec2;
```

```
int main(void) {
    DDRD = 0xFF;
    DDRC = 0x00;

    ADMUX = (1 << REFS0) | (1 << MUX1);
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
    ADCSRB = 0x00;
    DIDR0 = ~(1 << ADC2D);

    while (1) {
        ADCSRA |= (1 << ADSC);
        while ((ADCSRA & (1 << ADSC)) == (1 << ADSC)) {
            _delay_us(10);
        }
        ADC_val = ADC;
        ADC_val = (ADC_val * 5) / 1024;

        integer = (uint8_t)ADC_val;
        float decimal = ADC_val - integer;
        dec1 = (uint8_t)(decimal * 10);
        dec2 = (uint8_t)(((decimal * 10) - dec1) * 10);

        integer |= 0x30;
        dec1 |= 0x30;
        dec2 |= 0x30;

        lcd_init();
        _delay_ms(2);

        lcd_data(integer);
        lcd_data('.');
        lcd_data(dec1);
        lcd_data(dec2);

        _delay_ms(100);    //result must be visible for some time before lcd inits again
    }
}
```

Ζήτημα 4.2

Στην άσκηση αυτή καλούμαστε να προσομοιώσουμε ένα σύστημα ανίχνευσης μονοξειδίου του άνθρακα. Ανάλογα τη συγκέντρωση του αερίου, ανάβουν τα LED PB0-PB5. Ορίζουμε τα εξής επίπεδα (η συγκέντρωση μετριέται σε ppm):

- [0, 70): Ανάβει το 1^{ο} LED μόνο
- [70, 140): Ανάβουν τα 2 πρώτα LED
- [140, 210): Ανάβουν τα 3 πρώτα LED
- [210, 280): Ανάβουν τα 4 πρώτα LED
- [280, 350): Ανάβουν τα 5 πρώτα LED
- [350, …): Ανάβουν και τα 6 LED

Έχουμε τους εξής τύπους:

- $C_x = \frac{1}{M}(V_{gas} - V_{gas0})$, όπου C_x η συγκέντρωση του αερίου σε ppm, M μια ειδική παράμετρος, V_{gas} η τάση που δίνει ο αισθητήρας (προσομοιώνεται από ένα ποτενσιόμετρο, από 0 μέχρι 5V, όπως στην άσκηση 1) και $V_{gas0} = 0.1V$.
- $M = Sensitivity\ Code(^{nA}/_{ppm}) \cdot TIAGain(^{kV}/_{A}) \cdot 10^{-6}$

Μας δίνεται ότι το Sensitivity Code είναι 129nA/ppm ότι το TIA Gain είναι 100kV/A.

Επίσης, για το V_{gas}, εφόσον θα προέρχεται από μετατροπή ADC 10 bits, γνωρίζουμε ότι $V_{gas} = \frac{5 \cdot ADC}{1024}$.

Οπότε, υπολογίζουμε εκ των προτέρων τις τιμές ADC για τα παραπάνω επίπεδα καπνού. Προκύπτουν οι εξής τιμές:

- Επίπεδο 1: ADC < 205
- Επίπεδο 2: 205 <= ADC < 391
- Επίπεδο 3: 391 <= ADC < 576
- Επίπεδο 4: 576 <= ADC < 761
- Επίπεδο 5: 761 <= ADC < 947
- Επίπεδο 6: ADC >= 947

Ο κώδικας λειτουργεί ως εξής: Αρχικά θέτουμε μεταξύ άλλων τον ADC να τρέχει σε free running mode. Επίσης, όταν το αποτέλεσμα της μετατροπής είναι διαθέσιμο, καλείται η ρουτίνα εξυπηρέτησης διακοπής του ADC. Τέλος, ξεκινάμε την πρώτη μετατροπή και τυπώνουμε “CLEAR” στην lcd.

Όταν είναι έτοιμη η μετατροπή, με μια σειρά συγκρίσεων βρίσκουμε σε ποιο επίπεδο βρισκόμαστε και κάνουμε Jump σε αυτό. Για παράδειγμα, το επίπεδο 1 (που αντιστοιχεί σε μήνυμα clear), ανάβει μόνο το πρώτο LED. Στη συνέχεια, ελέγχει τον καταχωρητή prev_state.

Ο καταχωρητής αυτός αναλαμβάνει να αποθηκεύει την προηγούμενη κατάσταση που βρισκόμασταν (2 καταστάσεις, clear ή gas detected), ώστε αν η κατάσταση δεν αλλάξει, να μην ξανατυπωθεί το ίδιο μήνυμα. Αν ο prev_state είναι 0, τότε η προηγούμενη κατάσταση ήταν η clear και αν ο prev_state είναι 1, η προηγούμενη κατάσταση ήταν η gas detected. Ο καταχωρητής αυτός αρχικοποιείται στο 0, δηλαδή στην κατάσταση clear (το πρώτο print_clear πριν την main υπάρχει γιατί με την αρχικοποίηση του prev_state σε 0, αν δεν αλλάξει η κατάσταση, δεν θα τυπωθεί τίποτα στην οθόνη).

Στη συνέχεια, καλείται η συνάρτηση print_clear η οποία αρχικοποιεί την οθόνη και τυπώνει το μήνυμα “CLEAR”. Τέλος, τίθεται ο prev_state κατάλληλα, καλείται μια καθυστέρηση 100ms και επιστρέφουμε.

Τα υπόλοιπα επίπεδα λειτουργούν ομοίως (ανάβουν τα κατάλληλα LED, καλούν την print_gas_detected που τυπώνει “GAS DETECTED”, θέτουν τον prev_state, καθυστερούν 100ms και επιστρέφουν).

Όμως, εφόσον θέλουμε να αναβοσβήνουν τα LED σε περίπτωση ανίχνευσης αερίου, στα επίπεδα 2-6, κάνουμε το εξής: Κρατάμε έναν μετρητή με αρχική τιμή το 0. Κάθε φορά που μπαίνουμε σε ένα εκ των επιπέδων 2-6, καλούμε την συνάρτηση blink. Αυτή, αν ο μετρητής είναι <4, ανάβει στην PORTB τα κατάλληλα LED. Αν ο μετρητής είναι >=4, σβήνει όλα τα LED. Όταν ο μετρητής φτάσει στο 7, ξαναγίνεται 0. Έτσι, ανά 4*100ms = 400ms, αναβοσβήνουν τα LED.

```
.include "m328PBdef.inc"

.def temp = r16
.def prev_state = r17          ;prev_state is 0x00 if clear, prev_state is 0x01 if gas detected
                               ;useful for skipping lcd_init and output
.def cnt = r18
.def ADC_L = r21
.def ADC_H = r22

.org 0x00
   jmp reset

.org 0x2A
   jmp adc_ready

reset:
   ldi temp, high(RAMEND)
   out SPH, temp
   ldi temp, low(RAMEND)
   out SPL,temp

   ldi temp, 0xFF
   out DDRB, temp
   out DDRD, temp
   ldi temp, 0x00
   out DDRC, temp  ;Set PORTC as input

   ; REFSn[1:0]=01 => select Vref=5V, MUXn[4:0]=0011 => select ADC3(pin PC3),
   ; ADLAR=0 => right adjust the ADC result
   ldi temp, (1 << REFS0) | (1 << MUX1) | (1 << MUX0)
   sts ADMUX, temp

   ; ADEN=1 => ADC Enable, ADCS=0 => No Conversion yet
   ; ADIE=1 => enable adc interrupt, ADPS[2:0]=111 => fADC=16MHz/128=125KHz
   ; ADATE = 1, for free running
   ldi temp, (1 << ADEN) | (1 << ADIE) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0) | (1 << ADATE)
   sts ADCSRA, temp

   ldi temp, 0x00
   sts ADCSRB, temp

   ldi temp, 0xFF
   ldi r18, (1 << ADC3D)   ;r18 is used as dummy var
   eor temp, r18
   sts DIDR0, temp      ;disable all other ADC inputs

   sei                           ;enable all interrupts

   lds temp, ADCSRA              ;start first conversion
   ori temp, (1<<ADSC)
```

```
    sts ADCSRA, temp

    ldi prev_state, 0x00            ;original state is clear
    rcall print_clear

main:
    rjmp main

adc_ready:
    lds ADC_L, ADCL
    lds ADC_H, ADCH

    cpi ADC_H, 0x00
    breq lo1
    rjmp next1
lo1:                                ;ADC_H = 0x00
    cpi ADC_L, 0xCD
    brlo level1                     ;ADC_H = 0x00 and ADC_L < 0xCD
    rjmp level2                     ;ADC_H = 0x00 and ADC_L >= 0xCD
next1:
    cpi ADC_H, 0x01
    breq lo2
    rjmp next2
lo2:                                ;ADC_H = 0x01
    cpi ADC_L, 0x87
    brlo level2                     ;ADC_H = 0x01 and ADC_L < 0x87
    rjmp level3                     ;ADC_H = 0x01 and ADC_L >= 0x87
next2:
    cpi ADC_H, 0x02
    breq lo3
    rjmp next3
lo3:                                ;ADC_H = 0x02
    cpi ADC_L, 0x40
    brlo level3                     ;ADC_H = 0x02 and ADC_L < 0x40
    cpi ADC_L, 0xF9
    brlo level4                     ;ADC_H = 0x02 and ADC_L < 0xF9 (also ADC_L >= 0x40
    rjmp level5                     ;ADC_H = 0x02 and ADC_L >= 0xF9
next3:                              ;ADC_H must be 0x03
    cpi ADC_L, 0xB3
    brlo level5                     ;ADC_H = 0x03 and ADC_L < 0xB3
    rjmp level6                     ;ADC_H = 0x03 and ADC_L >= 0xB3

level1:
    ldi temp, 0x01
    out PORTB, temp
    sbrc prev_state, 0              ;if previous state was clear, dont output
    rcall print_clear
    cbr prev_state, 1 ;set previous state
    rcall delay

level2:
    ldi temp, 0x03
    rcall blink
    sbrs prev_state, 0              ;if previous state was gas detected, dont output
    rcall print_gas_detected
    sbr prev_state, 1
    rcall delay
```

```
level3:
   ldi temp, 0x07
   rcall blink
   sbrs prev_state, 0
   rcall print_gas_detected
   sbr prev_state, 1
   rcall delay

level4:
   ldi temp, 0x0F
   rcall blink
   sbrs prev_state, 0
   rcall print_gas_detected
   sbr prev_state, 1
   rcall delay

level5:
   ldi temp, 0x1F
   rcall blink
   sbrs prev_state, 0
   rcall print_gas_detected
   sbr prev_state, 1
   rcall delay

level6:
   ldi temp, 0x3F
   rcall blink
   sbrs prev_state, 0
   rcall print_gas_detected
   sbr prev_state, 1
   rcall delay

blink:
   cpi cnt, 0x04
   brlo lower
   rjmp higher
lower:
   out PORTB, temp
   inc cnt
   rjmp end1
higher:
   ldi temp, 0x00
   out PORTB, temp
   inc cnt
end1:
   sbrc cnt, 3
   ldi cnt, 0x00
   ret
```

```
delay:
    ldi r24, low(16*100)
    ldi r25, high(16*100)
    rcall wait_x_msec
    reti

print_clear:
    rcall lcd_init
    ldi r24, low(16*2)
    ldi r25, high(16*2)
    rcall wait_x_msec

    ldi r24, 'C'
    rcall lcd_data

    ldi r24, 'L'
    rcall lcd_data

    ldi r24, 'E'
    rcall lcd_data

    ldi r24, 'A'
    rcall lcd_data

    ldi r24, 'R'
    rcall lcd_data

    ret

print_gas_detected:
    rcall lcd_init
    ldi r24, low(16*2)
    ldi r25, high(16*2)
    rcall wait_x_msec

    ldi r24, 'G'
    rcall lcd_data

    ldi r24, 'A'
    rcall lcd_data

    ldi r24, 'S'
    rcall lcd_data

    ldi r24, ' '
    rcall lcd_data

    ldi r24, 'D'
    rcall lcd_data

    ldi r24, 'E'
    rcall lcd_data

    ldi r24, 'T'
    rcall lcd_data

    ldi r24, 'E'
```

```
    rcall lcd_data

    ldi r24, 'C'
    rcall lcd_data

    ldi r24, 'T'
    rcall lcd_data

    ldi r24, 'E'
    rcall lcd_data

    ldi r24, 'D'
    rcall lcd_data

    ret

write_2_nibbles:
    push r24                ;r24 contains data to be sent
    in r25, PIND
    andi r25, 0x0f
    andi r24, 0xf0
    add r24, r25
    out PORTD, r24
    sbi PORTD, 3
    nop
    nop
    cbi PORTD, 3
    pop r24
    swap r24
    andi r24, 0xf0
    add r24, r25
    out PORTD, r24
    sbi PORTD, 3
    nop
    nop
    cbi PORTD, 3
    ret


lcd_data:
    sbi PORTD, 2
    rcall write_2_nibbles
    ldi r24, 100
    ldi r25, 0
    rcall wait_usec
    ret


lcd_command:
    cbi PORTD, 2
    rcall write_2_nibbles
    ldi r24, 100
    ldi r25, 0
    rcall wait_usec
    ret
```

```
lcd_init:
   ldi r24, low(16*40)
   ldi r25, high(16*40)
   rcall wait_x_msec
   ldi r24, 0x30
   out PORTD, r24
   sbi PORTD, 3
   nop
   nop
   cbi PORTD, 3
   ldi r24, 100
   ldi r25, 0
   rcall wait_usec
   ldi r24, 0x30
   out PORTD, r24
   sbi PORTD, 3
   nop
   nop
   cbi PORTD, 3
   ldi r24, 100
   ldi r25, 0
   rcall wait_usec
   ldi r24, 0x20
   out PORTD, r24
   sbi PORTD, 3
   nop
   nop
   cbi PORTD, 3
   ldi r24, 100
   ldi r25, 0
   rcall wait_usec
   ldi r24, 0x28
   rcall lcd_command
   ldi r24, 0x0c
   rcall lcd_command
   ldi r24, 0x01
   rcall lcd_command
   ldi r24, low(5000)
   ldi r25, high(5000)
   rcall wait_usec
   ldi r24, 0x06
   rcall lcd_command
   ret

wait_x_msec:
   rcall delay_986u
   sbiw r24, 1
   breq end

   rjmp help1

help1:
   rjmp help2              ;2 cc

help2:
   rjmp help3              ;2 cc

help3:
   rjmp wait_x_msec        ;2 cc
```

```
end:
   ret

delay_986u:
   ldi r26, 98

loop_986u:
   rcall wait_4u
   dec r26
   brne loop_986u

   nop
   nop
   ret

wait_4u:
   ret


wait_usec:
   sbiw r24 ,1
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   brne wait_usec
   ret
```

Ο κώδικας σε C λειτουργεί με ακριβώς αντίστοιχο τρόπο. Η μόνη διαφορά είναι ότι χρησιμοποιούμε τη μέθοδο polling για να περιμένουμε το τέλος της μετατροπής ADC.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

static volatile uint8_t prev_state = 0;
static volatile uint16_t ADC_val;
static volatile uint8_t cnt;

void write_2_nibbles(uint8_t c) {
   uint8_t temp = c;
   uint8_t prev = PIND;
   prev &= 0x0F;
   c &= 0xF0;
   c |= prev;
   PORTD = c;
   PORTD |= 0x08;
   PORTD &= 0xF7;
```

```
    c = temp;
    c &= 0x0F;
    c = c << 4;
    c |= prev;
    PORTD = c;
    PORTD |= 0x08;
    PORTD &= 0xF7;

    return;
}

void lcd_data(uint8_t c) {
    PORTD |= 0x04;
    write_2_nibbles(c);
    _delay_us(100);
    return;
}

void lcd_command(uint8_t c) {
    PORTD &= 0xFB;
    write_2_nibbles(c);
    _delay_us(100);
    return;
}

void lcd_init(void) {
    _delay_ms(40);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    PORTD = 0x20;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    lcd_command(0x28);
    lcd_command(0x0C);
    lcd_command(0x01);
    _delay_us(5000);

    lcd_command(0x06);
    return;
}

void print_clear(void) {
    lcd_init();
    _delay_ms(2);
    lcd_data('C');
    lcd_data('L');
    lcd_data('E');
    lcd_data('A');
    lcd_data('R');
    return;
}
```

```
void print_gas_detected(void) {
   lcd_init();
   _delay_ms(2);
   lcd_data('G');
   lcd_data('A');
   lcd_data('S');
   lcd_data(' ');
   lcd_data('D');
   lcd_data('E');
   lcd_data('T');
   lcd_data('E');
   lcd_data('C');
   lcd_data('T');
   lcd_data('E');
   lcd_data('D');
   return;
}

void blink(uint8_t temp) {
   if (cnt < 4) {
      PORTB = temp;
      cnt++;
   }else {
      PORTB = 0x00;
      cnt++;
   }
   if (cnt == 8) {
      cnt = 0;
   }
   return;
}

void level1(void) {
   PORTB = 0x01;
   if (prev_state != 0) {
      print_clear();
      prev_state = 0;
   }
   return;
}

void level2(void) {
   blink(0x03);
   if (prev_state != 1) {
      print_gas_detected();
      prev_state = 1;
   }
   return;
}

void level3(void) {
   blink(0x07);
   if (prev_state != 1) {
      print_gas_detected();
      prev_state = 1;
   }
   return;
}
```

```
void level4(void) {
   blink(0x0F);
   if (prev_state != 1) {
      print_gas_detected();
      prev_state = 1;
   }
   return;
}

void level5(void) {
   blink(0x1F);
   if (prev_state != 1) {
      print_gas_detected();
      prev_state = 1;
   }
   return;
}

void level6(void) {
   blink(0x3F);
   if (prev_state != 1) {
      print_gas_detected();
      prev_state = 1;
   }
   return;
}

int main(void) {
   DDRD = 0xFF;
   DDRB = 0xFF;
   DDRC = 0x00;

   ADMUX = (1 << REFS0) | (1 << MUX1) | (1 << MUX0);
   ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
   ADCSRB = 0x00;
   DIDR0 = ~(1 << ADC3D);

   cnt = 0;
   prev_state = 9;
   print_clear();

   while (1) {
      ADCSRA |= (1 << ADSC);
      while ((ADCSRA & (1 << ADSC)) == (1 << ADSC)) {
         _delay_us(10);
      }
      ADC_val = ADC;

      if (ADC_val < 205) {
         level1();
      }else if (ADC_val >= 205 && ADC_val < 391) {
         level2();
      }else if (ADC_val >= 391 && ADC_val < 576) {
         level3();
      }else if (ADC_val >= 576 && ADC_val < 761) {
         level4();
      }else if (ADC_val >= 761 && ADC_val < 947) {
         level5();
      }else {
         level6();
      }
      _delay_ms(100);
   }
}
```

Ζήτημα 4.3

Στη main αρχικοποιούμε τον χρονιστή ώστε να παράγει fast PWM 8 bit με BOTTOM=0 και TOP = ICR1 = 399. Με prescaler στα 8, προκύπτει η επιθυμητή συχνότητα των 5Khz (η κυματομορφή δεν παράγεται ακόμα, πρέπει να θέσουμε το bit COM1A1 του TCCR1A, πράγμα που γίνεται μόνο όταν πατηθεί κάποιο κουμπί). Προετοιμάζουμε και τους καταχωρητές λειτουργίας ADC και ξεκινάμε να διαβάζουμε την PINB. Όταν πατηθεί κάποιο κουμπί, καλείται η αντίστοιχη συνάρτηση, κατ' αντιστοιχία με την άσκηση 3 της 3^{ης} σειράς ασκήσεων. Όταν αφεθεί το κουμπί αυτό και επιστρέψουμε, ξανακαλούμε την lcd_init ώστε να καθαριστεί η οθόνη και σταματάμε την παραγωγή PWM.

Κάθε μια από τις 4 συναρτήσεις αρχικά θέτει τον OCR1A στην κατάλληλη τιμή ώστε να έχουμε το επιθυμητό DC. Η τιμή υπολογίζεται ως το DC επί το TOP. Στη συνέχεια, αρχικοποιεί την οθόνη και τυπώνει στην πρώτη σειρά το αντίστοιχο ποσοστό. Μετά, μπαίνει σε μια άπειρη επανάληψη. Εκεί, καλείται η lcd command η οποία τοποθετεί τον cursor στην πρώτη θέση της δεύτερης γραμμής της lcd. Στη συνέχεια καλείται η read_print_ADCval. Η συνάρτηση αυτή, απλά ξεκινάει μια μέτρηση, διαβάζει το αποτέλεσμα και το τυπώνει στην lcd με ακρίβεια 2 δεκαδικών (έχει και καθυστέρηση 1s ώστε να μην αλλάζει πολύ γρήγορα το αποτέλεσμα). Όταν επιστρέψει η read_print_ADCval, ελέγχουμε αν αφέθηκε το κουμπί που ήταν πατημένο. Αν ναι, βγαίνουμε από την άπειρη επανάληψη και επιστρέφουμε, αν όχι, επαναλαμβάνουμε την διαδικασία.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

static volatile float ADC_val;
static volatile uint8_t integer;
static volatile uint8_t dec1;
static volatile uint8_t dec2;

void write_2_nibbles(uint8_t c) {
   uint8_t temp = c;
   uint8_t prev = PIND;
   prev &= 0x0F;
   c &= 0xF0;
   c |= prev;
   PORTD = c;
   PORTD |= 0x08;
   PORTD &= 0xF7;

   c = temp;
   c &= 0x0F;
   c = c << 4;
   c |= prev;
   PORTD = c;
   PORTD |= 0x08;
   PORTD &= 0xF7;

   return;
}

void lcd_data(uint8_t c) {
   PORTD |= 0x04;
   write_2_nibbles(c);
```

```
    _delay_us(100);
    return;
}

void lcd_command(uint8_t c) {
    PORTD &= 0xFB;
    write_2_nibbles(c);
    _delay_us(100);
    return;
}

void lcd_init(void) {
    _delay_ms(40);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    PORTD = 0x20;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(100);

    lcd_command(0x28);
    lcd_command(0x0C);
    lcd_command(0x01);
    _delay_us(5000);

    lcd_command(0x06);
    return;
}

void read_print_ADCval(void) {
    ADCSRA |= (1 << ADSC);
    while ((ADCSRA & (1 << ADSC)) == (1 << ADSC)) {
        _delay_us(10);
    }
    ADC_val = ADC;
    ADC_val = (ADC_val * 5) / 1024;

    integer = (uint8_t)ADC_val;
    float decimal = ADC_val - integer;
    dec1 = (uint8_t)(decimal * 10);
    dec2 = (uint8_t)(((decimal * 10) - dec1) * 10);

    integer |= 0x30;
    dec1 |= 0x30;
    dec2 |= 0x30;

    lcd_data(integer);
    lcd_data('.');
    lcd_data(dec1);
    lcd_data(dec2);
```

```
    _delay_ms(1000);    //result must be visible for some time before lcd inits again
    return;
}

void pb2(void) {
    OCR1A = 80;
    TCCR1A |= (1 << COM1A1);

    lcd_init();
    _delay_ms(2);

    lcd_data('2');
    lcd_data('0');
    lcd_data('%');

    while(1) {
        lcd_command(0xC0);
        read_print_ADCval();
        if ((PINB & 0x04) == 0x04) {
            break;
        }
    }

    return;
}

void pb3(void) {
    TCCR1A |= (1 << COM1A1);
    OCR1A = 160;

    lcd_init();
    _delay_ms(2);

    lcd_data('4');
    lcd_data('0');
    lcd_data('%');

    while(1) {
        lcd_command(0xC0);
        read_print_ADCval();
        if ((PINB & 0x08) == 0x08) {
            break;
        }
    }

    return;
}

void pb4(void) {
    TCCR1A |= (1 << COM1A1);
    OCR1A = 240;

    lcd_init();
    _delay_ms(2);

    lcd_data('6');
    lcd_data('0');
    lcd_data('%');
```

```
    while(1) {
      lcd_command(0xC0);
      read_print_ADCval();
      if ((PINB & 0x10) == 0x10) {
        break;
      }
    }

    return;
}

void pb5(void) {
    TCCR1A |= (1 << COM1A1);
    OCR1A = 320;

    lcd_init();
    _delay_ms(2);

    lcd_data('8');
    lcd_data('0');
    lcd_data('%');

    while(1) {
      lcd_command(0xC0);
      read_print_ADCval();
      if ((PINB & 0x20) == 0x20) {
        break;
      }
    }

    return;
}

int main(void) {
    DDRD = 0xFF;
    DDRB = 0x02;
    DDRC = 0x00;

    static unsigned char x;

    ICR1 = 399;
    TCNT1 = 0x0000;
    TCCR1A = (1 << WGM11);
    TCCR1B = (1 << WGM12) | (1 << WGM13 ) | (1 << CS11);

    ADMUX = (1 << REFS0) | (1 << MUX0);
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
    ADCSRB = 0x00;
    DIDR0 = ~(1 << ADC1D);

    while (1) {
      x = PINB;
      if ((x & 0x04) != 0x04) {   //check if PB2 was pressed
        pb2();
      }
      if ((x & 0x08) != 0x08) {   //check if PB3 was pressed
        pb3();
      }
      if ((x & 0x10) != 0x10) {   //check if PB4 was pressed
```

```
          pb5();
        }
        lcd_init();
        _delay_ms(2);
        TCCR1A = (1 << WGM11);
    }
}
```