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

Αναφορά

7η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ ΓΙΑ ΤΟ ΜΑΘΗΜΑ "Εργαστήριο Μικροϋπολογιστών"

Αισθητήρας Θερμοκρασίας DS1820 στην κάρτα ntuAboard_G1

Ομάδα 1

Ξανθόπουλος Παναγιώτης (03119084)

Παπαναστάσης Αθανάσιος (03113197)

Ζήτημα 7.1

Στην άσκηση αυτή παίρνουμε μετρήσεις από έναν αισθητήρα θερμοκρασίας DS1820 τον οποίο έχουμε συνδέσει στον ακροδέκτη PD4.

Ορίζουμε συναρτήσεις αντίστοιχα με τις ρουτίνες που έχουν δωθεί στην εκφώνηση και τη συνάρτηση read_temp, η οποία αναλαμβάνει να πραγματοποιήσει την μέτρηση. Συγκεκριμένα, καλεί την one_wire_reset για αρχικοποίηση και ταυτόχρονα για έλεγχο αν υπάρχει συνδεδεμένο θερμόμετρο. Αν δεν υπάρχει, θα επιστραφεί έξοδος 0x8000, ενώ αν υπάρχει θα καλέσει την one_wire_transmit αρχικά για αποστολή της εντολής 0xCC και μετά για αποστολή της 0x44 για να ξεκινήσει η μέτρηση της θερμοκρασίας. Χρησιμοποιούμε μία while με έλεγχο αν η one_wire_receive_bit ισούται με 0x00, ώστε μόλις ληφθεί η θερμοκρασία και η επιστροφή της συνάρτησης είναι διαφορετική να συνεχιστεί το πρόγραμμα. Γίνεται ξανά αρχικοποίηση και έλεγχος αν υπάρχει συνδεδεμένο θερμόμετρο, και αυτή τη φορά, αν είναι συνδεδεμένο το θερμόμετρο, αφού καλέσουμε την one_wire_transmit_byte για αποστολή της εντολής 0xCC, καλούμε ξανά την one_wire_transmit_byte για αποστολή της εντολής 0xBE για ανάγνωση των 16 bit της μετρημένης θερμοκρασίας. Στη συνέχεια, καλούμε την one_wire_receive_byte αποθηκεύουμε τα πρώτα 8 bit στη μεταβλητή lsbyte και τα επόμενα 8 στη msbyte. Στη συνέχεια, με την κατάλληλη μετατόπιση 8 θέσεων αριστερά των bit της msbyte αποθηκεύουμε στη μεταβλητή ret το άθροισμα lsbyte με msbyte και το επιστρέφουμε.

Η main καλεί τη read_temp ανά ένα δευτερόλεπτο και αποθηκεύει το αποτέλεσμα στη μεταβλητή measurement.

```
#define F_CPU 16000000UL

#include<avr/io.h>
#include<util/delay.h>

uint8_t one_wire_reset(void)
{
   DDRD |= 0x10;
   PORTD &= 0xEF;
   _delay_us(480);

   DDRD &= 0xEF;
   PORTD &= 0xEF;

   _delay_us(100);

   uint8_t temp = PIND;
   _delay_us(380);

   temp &= 0x10;      //isolate PD4 bit

   if (temp == 0x10)
   {
      return 0;
   }
   return 1;
}
```

```
uint8_t one_wire_receive_bit(void)
{
   DDRD |= 0x10;
   PORTD &= 0xEF;

   _delay_us(2);

   DDRD &= 0xEF;
   PORTD &= 0xEF;

   _delay_us(10);

   uint8_t temp = PIND;

   temp &= 0x10;
   temp = temp >> 4;

   _delay_us(49);

   return temp;
}

void one_wire_transmit_bit(uint8_t val)
{
   DDRD |= 0x10;
   PORTD &= 0xEF;

   _delay_us(2);

   val &= 0x01;

   if(val == 0x01)
   {
      PORTD |= 0x10;
   } else if(val == 0x00)
   {
      PORTD &= 0xEF;
   }

   _delay_us(58);

   DDRD &= 0xEF;
   PORTD &= 0xEF;

   _delay_us(1);

   return;
}

uint8_t one_wire_receive_byte(void)
{
   uint8_t temp, ret = 0x00;

   for(int i=0; i<8; i++)
   {
      temp = one_wire_receive_bit();
```

```
      temp = temp << i;
      ret |= temp;
   }
   return ret;
}

void one_wire_transmit_byte(uint8_t byte)
{
   uint8_t temp;

   for(int i=0; i<8; i++)
   {
      temp = byte >> i;
      temp &= 0x01;
      one_wire_transmit_bit(temp);
   }
   return;
}

static volatile uint16_t measurement;
static volatile uint8_t lsbyte, msbyte;

uint16_t read_temp(void)
{
   uint16_t ret;

   if(one_wire_reset() == 0)
   {
      return 0x8000;
   } else
   {
      one_wire_transmit_byte(0xCC);

      one_wire_transmit_byte(0x44);

      while(one_wire_receive_bit() == 0x00)
      {
         _delay_ms(750);
      }// wait for the measurement to become available

      if(one_wire_reset() == 0)
      {
         return 0x8000;
      } else
      {
         one_wire_transmit_byte(0xCC);

         one_wire_transmit_byte(0xBE);

         lsbyte = one_wire_receive_byte();

         msbyte = one_wire_receive_byte();

         ret = msbyte;
         ret  = ret << 8;
         ret += lsbyte;
         return ret;
      }
   }
}
```

```
int main(void)
{
   while(1)
   {
      measurement = read_temp();
      _delay_ms(1000);
   }
}
```

Ζήτημα 4.2

Στην άσκηση αυτή, χρησιμοποιούμε την LCD οθόνη για την εμφάνιση των αποτελεσμάτων. Για το σκοπό αυτό, χρησιμοποιούμε τις ρουτίνες για την LCD και το Port Expander, έτοιμες από την προηγούμενη σειρά ασκήσεων. Επίσης, για το διάβασμα των μετρήσεων, χρησιμοποιούμε τις συναρτήσεις από το ζήτημα 7.1.

Επίσης ορίζουμε τη συνάρτηση print_to_lcd, που θα εμφανίζει τη θερμοκρασία στην lcd οθόνη. Αρχικά, αφού αρχικοποιήσει την οθόνη, ελέγχει αν η παράμετρος sign είναι 0x01, οπότε και εμφανίζει « - », καθώς η θερμοκρασία θα είναι αρνητική και αλλάζει την τιμή της val στην αντίθετή της, αλλιώς εμφανίζει « + ». Αποθηκεύουμε την τιμή της val στη float μεταβλητή temp, την οποία διαιρούμε με το 16 ώστε να μετατραπεί στον κατάλληλο δεκαδικό αριθμό. Επίσης, αποθηκεύουμε το ακέραιο μέρος στην int temp1. Στη συνέχεια, με αλλεπάλληλες ακέραιες διαιρέσεις, παίρνουμε τις εκατοντάδες, δεκάδες και μονάδες του ακέραιου μέρους, τις οποίες και τυπώνουμε. Ελέγχουμε αν το δεκαδικό μέρος είναι μεγαλύτερο του μηδενός, και αν είναι, με τις κατάλληλες πράξεις απομονώνουμε και τυπώνουμε 3 δεκαδικά ψηφία. Τέλος, τυπώνουμε και το «°C».

Στο κυρίως πρόγραμμα καταχωρούμε τη μέτρηση του θερμομέτρου στη μεταβλητή measurement και αν ισούται με 0x8000 καλείται η write_nodev, η οποία τυπώνει το μήνυμα «NO DEVICE». Αν η τιμή measurement είναι μεγαλύτερη ή ίση με 0xF800 (δηλαδή η θερμοκρασία είναι υπό του μηδενός), καλεί την print_to_lcd με παραμέτρους τη μεταβλητή measurement και το 0x01, αλλιώς καλεί την print_to_lcd με παραμέτρους τη μεταβλητή measurement και το 0x00.

```
#define F_CPU 16000000UL

#include<avr/io.h>
#include<avr/interrupt.h>
#include<util/delay.h>

#define PCA9555_0_ADDRESS 0x40 //A0=A1=A2=0 by hardware
#define TWI_READ 1 // reading from twi device
#define TWI_WRITE 0 // writing to twi device
#define SCL_CLOCK 100000L // twi clock in Hz
//Fscl=Fcpu/(16+2*TWBR0_VALUE*PRESCALER_VALUE)
#define TWBR0_VALUE ((F_CPU/SCL_CLOCK)-16)/2
```

```
// PCA9555 REGISTERS
typedef enum {
   REG_INPUT_0 = 0,
   REG_INPUT_1 = 1,
   REG_OUTPUT_0 = 2,
   REG_OUTPUT_1 = 3,
   REG_POLARITY_INV_0 = 4,
   REG_POLARITY_INV_1 = 5,
   REG_CONFIGURATION_0 = 6,
   REG_CONFIGURATION_1 = 7,
} PCA9555_REGISTERS;

//---------- Master Transmitter/Receiver ----------
#define TW_START 0x08
#define TW_REP_START 0x10

//---------- Master Transmitter ----------
#define TW_MT_SLA_ACK 0x18
#define TW_MT_SLA_NACK 0x20
#define TW_MT_DATA_ACK 0x28

//---------- Master Receiver ----------
#define TW_MR_SLA_ACK 0x40
#define TW_MR_SLA_NACK 0x48
#define TW_MR_DATA_NACK 0x58

#define TW_STATUS_MASK 0b11111000
#define TW_STATUS (TWSR0 & TW_STATUS_MASK)

//initialize TWI clock
void twi_init(void)
{
   TWSR0 = 0; // PRESCALER_VALUE=1
   TWBR0 = TWBR0_VALUE; // SCL_CLOCK 100KHz
}

// Read one byte from the twi device (request more data from device)
unsigned char twi_readAck(void)
{
   TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
   while(!(TWCR0 & (1<<TWINT)));
   return TWDR0;
}

// Read one byte from the twi device (dont request more data from device)
unsigned char twi_readNak(void)
{
   TWCR0 = (1<<TWINT) | (1<<TWEN);
   while(!(TWCR0 & (1<<TWINT)));
   return TWDR0;
}

// Issues a start condition and sends address and transfer direction.
// return 0 = device accessible, 1= failed to access device
unsigned char twi_start(unsigned char address)
{
   uint8_t twi_status;
```

```
// send START condition
   TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
   // wait until transmission completed
   while(!(TWCR0 & (1<<TWINT)));
   // check value of TWI Status Register.
   twi_status = TW_STATUS & 0xF8;
   if ( (twi_status != TW_START) && (twi_status != TW_REP_START)) return 1;
   // send device address
   TWDR0 = address;
   TWCR0 = (1<<TWINT) | (1<<TWEN);
   // wail until transmission completed and ACK/NACK has been received
   while(!(TWCR0 & (1<<TWINT)));
   // check value of TWI Status Register.
   twi_status = TW_STATUS & 0xF8;
   if ( (twi_status != TW_MT_SLA_ACK) && (twi_status != TW_MR_SLA_ACK) )
   {
      return 1;
   }
   return 0;
}

// Send start condition, address, transfer direction.
// Use ack polling to wait until device is ready
void twi_start_wait(unsigned char address)
{
   uint8_t twi_status;
   while ( 1 )
   {
      // send START condition
      TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
      // wait until transmission completed
      while(!(TWCR0 & (1<<TWINT)));
      // check value of TWI Status Register.
      twi_status = TW_STATUS & 0xF8;
      if ( (twi_status != TW_START) && (twi_status != TW_REP_START)) continue;
      // send device address
      TWDR0 = address;
      TWCR0 = (1<<TWINT) | (1<<TWEN);
      // wail until transmission completed
      while(!(TWCR0 & (1<<TWINT)));
      // check value of TWI Status Register.
      twi_status = TW_STATUS & 0xF8;
      if ( (twi_status == TW_MT_SLA_NACK )||(twi_status ==TW_MR_DATA_NACK) )
      {
         /* device busy, send stop condition to terminate write operation */
         TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
         // wait until stop condition is executed and bus released
         while(TWCR0 & (1<<TWSTO));
         continue;
      }
      break;
   }
}

// Send one byte to twi device, Return 0 if write successful or 1 if write failed
unsigned char twi_write( unsigned char data )
{
   // send data to the previously addressed device
```

```
   TWDR0 = data;
   TWCR0 = (1<<TWINT) | (1<<TWEN);
   // wait until transmission completed
   while(!(TWCR0 & (1<<TWINT)));
   if( (TW_STATUS & 0xF8) != TW_MT_DATA_ACK) return 1;
   return 0;
}

// Send repeated start condition, address, transfer direction
//Return: 0 device accessible
// 1 failed to access device
unsigned char twi_rep_start(unsigned char address)
{
   return twi_start( address );
}

// Terminates the data transfer and releases the twi bus
void twi_stop(void)
{
   // send stop condition
   TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
   // wait until stop condition is executed and bus released
   while(TWCR0 & (1<<TWSTO));
}

void PCA9555_0_write(PCA9555_REGISTERS reg, uint8_t value)
{
   twi_start_wait(PCA9555_0_ADDRESS + TWI_WRITE);
   twi_write(reg);
   twi_write(value);
   twi_stop();
}

uint8_t PCA9555_0_read(PCA9555_REGISTERS reg)
{
   uint8_t ret_val;
   twi_start_wait(PCA9555_0_ADDRESS + TWI_WRITE);
   twi_write(reg);
   twi_rep_start(PCA9555_0_ADDRESS + TWI_READ);
   ret_val = twi_readNak();
   twi_stop();
   return ret_val;
}

void write_2_nibbles(uint8_t c) {
   uint8_t temp = c;
   uint8_t prev = PCA9555_0_read(REG_INPUT_0);
   prev &= 0x0F;
   c &= 0xF0;
   c |= prev;
   PCA9555_0_write(REG_OUTPUT_0, c);
   c |= 0x08;
   PCA9555_0_write(REG_OUTPUT_0, c);
   c &= 0xF7;
   PCA9555_0_write(REG_OUTPUT_0, c);

   c = temp;
   c &= 0x0F
```

```
    c = c << 4;
    c |= prev;
    PCA9555_0_write(REG_OUTPUT_0, c);
    c |= 0x08;
    PCA9555_0_write(REG_OUTPUT_0, c);
    c &= 0xF7;
    PCA9555_0_write(REG_OUTPUT_0, c);

    return;
}

void lcd_data(uint8_t c) {
    uint8_t temp = PCA9555_0_read(REG_INPUT_0);
    temp |= 0x04;
    PCA9555_0_write(REG_OUTPUT_0, temp);
    write_2_nibbles(c);
    _delay_us(100);
    return;
}

void lcd_command(uint8_t c) {
    uint8_t temp = PCA9555_0_read(REG_INPUT_0);
    temp &= 0xFB;
    PCA9555_0_write(REG_OUTPUT_0, temp);
    write_2_nibbles(c);
    _delay_us(100);
    return;
}

void lcd_init(void) {
    _delay_ms(40);

    PCA9555_0_write(REG_OUTPUT_0, 0x30);
    PCA9555_0_write(REG_OUTPUT_0, 0x38);
    PCA9555_0_write(REG_OUTPUT_0, 0x30);

    _delay_us(100);

    PCA9555_0_write(REG_OUTPUT_0, 0x30);
    PCA9555_0_write(REG_OUTPUT_0, 0x38);
    PCA9555_0_write(REG_OUTPUT_0, 0x30);

    _delay_us(100);

    PCA9555_0_write(REG_OUTPUT_0, 0x20);
    PCA9555_0_write(REG_OUTPUT_0, 0x28);
    PCA9555_0_write(REG_OUTPUT_0, 0x20);

    _delay_us(100);

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

    lcd_command(0x06);
    return;
}
```

```
uint8_t one_wire_reset(void)
{
   DDRD |= 0x10;
   PORTD &= 0xEF;
   _delay_us(480);

   DDRD &= 0xEF;
   PORTD &= 0xEF;

   _delay_us(100);

   uint8_t temp = PIND;
   _delay_us(380);

   temp &= 0x10;      //isolate PD4 bit

   if (temp == 0x10)
   {
      return 0;
   }
   return 1;
}

uint8_t one_wire_receive_bit(void)
{
   DDRD |= 0x10;
   PORTD &= 0xEF;

   _delay_us(2);

   DDRD &= 0xEF;
   PORTD &= 0xEF;

   _delay_us(10);

   uint8_t temp = PIND;

   temp &= 0x10;
   temp = temp >> 4;

   _delay_us(49);

   return temp;
}

void one_wire_transmit_bit(uint8_t val)
{
   DDRD |= 0x10;
   PORTD &= 0xEF;

   _delay_us(2);

   val &= 0x01;

   if(val == 0x01)
   {
      PORTD |= 0x10;
   } else if(val == 0x00)
```

```
    {
        PORTD &= 0xEF;
    }

    _delay_us(58);

    DDRD &= 0xEF;
    PORTD &= 0xEF;

    _delay_us(1);

    return;
}

uint8_t one_wire_receive_byte(void)
{
    uint8_t temp, ret = 0x00;

    for(int i=0; i<8; i++)
    {
        temp = one_wire_receive_bit();
        temp = temp << i;
        ret |= temp;
    }
    return ret;
}

void one_wire_transmit_byte(uint8_t byte)
{
    uint8_t temp;

    for(int i=0; i<8; i++)
    {
        temp = byte >> i;
        temp &= 0x01;
        one_wire_transmit_bit(temp);
    }
    return;
}

static volatile uint16_t measurement;
static volatile uint8_t lsbyte, msbyte;

uint16_t read_temp(void)
{
    uint16_t ret;

    if(one_wire_reset() == 0)
    {
        return 0x8000;
    } else
    {
        one_wire_transmit_byte(0xCC);

        one_wire_transmit_byte(0x44);

        while(one_wire_receive_bit() == 0x00)
        {
```

```
        _delay_ms(750);
      }  // wait for the measurement to become available

      if(one_wire_reset() == 0)
      {
        return 0x8000;
      } else
      {
        one_wire_transmit_byte(0xCC);

        one_wire_transmit_byte(0xBE);

        lsbyte = one_wire_receive_byte();

        msbyte = one_wire_receive_byte();

        ret = msbyte;
        ret  = ret << 8;
        ret += lsbyte;
        return ret;
      }
   }
}

void write_nodev(void)
{
   lcd_init();
   _delay_ms(2);
   lcd_data('N');
   lcd_data('O');
   lcd_data(' ');
   lcd_data('D');
   lcd_data('E');
   lcd_data('V');
   lcd_data('I');
   lcd_data('C');
   lcd_data('E');
   return;
}

//for DS18B20

void print_to_lcd(uint16_t val, uint8_t sign)
{
   lcd_init();
   _delay_ms(2);
   if(sign == 0x01)
   {
      lcd_data('-');
      val = ~val;
      val = val + 1;
   } else if(sign == 0x00)
   {
      lcd_data('+');
   }

   float temp = val;
   temp = temp / 16;
```

```
    int temp1 = (int)temp;      // get integer part

    uint8_t hun = temp1 / 100;
    lcd_data(hun | 0x30);

    uint8_t ten = (temp1 - (hun * 100)) / 10;
    lcd_data(ten | 0x30);

    uint8_t one = (temp1 - (hun * 100) - (ten * 10));
    lcd_data(one | 0x30);

    temp = temp - temp1;        // temp now contains only the decimal part, which can be 0.5 or 0

    if(temp > 0)
    {
        lcd_data('.');
        int decimal1 = (int)(temp * 10);
        lcd_data(decimal1 | 0x30);
        int decimal2 = (int)(temp * 100) % 10;
        lcd_data(decimal2 | 0x30);
        int decimal3 = (int)(temp * 1000) % 10;
        lcd_data(decimal3 | 0x30);
    }

    lcd_data(223);
    lcd_data('C');
    return;
}

int main(void)
{
    twi_init();
    PCA9555_0_write(REG_CONFIGURATION_0, 0x00);

    while(1)
    {
        measurement = read_temp();
        if(measurement == 0x8000)
        {
            write_nodev();
        } else
        {
            if(measurement >= 0xF800)
            {
                print_to_lcd(measurement, 0x01);
            } else
            {
                print_to_lcd(measurement, 0x00);
            }
        }
    }
}
```