# Αισθητήρας θερμοκρασίας DS1820 στην κάρτα ntuAboard_G1

Εργαστήριο Μικροϋπολογιστών

Κριθαρίδης Κωνσταντίνος, *el*21045 Μπαλάτος Δημήτριος, *el*21170

26 Νοεμβρίου 2024

## 1 Ζήτημα 4.1

Αρχικά μετατρέπουμε τον δωσμένο κώδικα από Assembly σε C. Στην συνέχεια ορίζουμε νέα βοηθητική συνάρτηση `read_temp(void)` που διαβάζει την τιμή του DS1820 και επιστρέφει αντίστοιχη `float` τιμή.

Για διευκόλυνση, ορίσαμε τύπο `bool` και μακροεντολές `SET(x, b)` , `CLEAR(x, b)` για την γρήγορη αλλαγή κάποιου bit σε αριθμό.

```c
/*
 * main.c
 *
 * Created: 11/22/2024 11:04:17 AM
 *  Author: User
 */

#include <xc.h>

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

typedef uint8_t bool;
#define true 1
#define false 0

#define SET(x, b)   do { (x) |=   1 << (b);  } while (0)
#define CLEAR(x, b) do { (x) &= ~(1 << (b)); } while (0)

/* TEMPLATE CODE FROM GIVEN ASSEMBLY */
/* {{{ */
bool one_wire_reset (void)
{
        SET(DDRD, 4); // set output

        CLEAR(PORTD, 4); // sent 0
```

```
        _delay_us(480);

        CLEAR(DDRD, 4); // set input
        CLEAR(PORTD, 4); // disable pull-up
        _delay_us(100);

        int tmp = PIND; // read PORTD
        _delay_us(380);
        return (tmp & (1 << 4)) == 0;
}

uint8_t one_wire_receive_bit (void)
{
        SET(DDRD, 4); // Set output
        CLEAR(PORTD, 4);
        _delay_us(2);

        CLEAR(DDRD, 4); // set input
        CLEAR(PORTD, 4); // disable pull-up
        _delay_us(10);

        uint8_t ret = PIND & 1;
        _delay_us(49);
        return ret;
}

void one_wire_transmit_bit (uint8_t in)
{
        SET(DDRD, 4); // Set output
        CLEAR(PORTD, 4);
        _delay_us(2);

        in &= 1; // keep only LSB
        if (in) SET(PORTD, 4);
        else    CLEAR(PORTD, 4);

        _delay_us(58);
        CLEAR(DDRD, 4); // set input
        CLEAR(PORTD, 4); // disable pull-up
        _delay_us(1);
}

uint8_t one_wire_receive_byte (void)
{
        uint8_t ret = 0;
        for (int i=0; i<8; ++i)
                ret |= (one_wire_receive_bit() << i);
        return ret;
}

void one_wire_transmit_byte (uint8_t in)
```

```
{
        for (int i=0; i<8; ++i, in >>= 1)
                one_wire_transmit_bit(in);
}
/* }}} */

float read_temp (void)
{
        int16_t ret = 0;

        if (one_wire_reset() != 0) return 0x8000;
        one_wire_transmit_byte(0xCC);
        one_wire_transmit_byte(0x44);
        while (one_wire_receive_bit() != 1);

        if (one_wire_reset() != 0) return 0x8000;
        one_wire_transmit_byte(0xCC);
        one_wire_transmit_byte(0xBE);

        ret |= one_wire_receive_byte() << 8;
        ret |= one_wire_receive_byte();
        return (float)(ret) / 2;
}

int main(void)
{
        return 0;
}

```

## 2 Ζήτημα 4.2

> Στην πλακέτα μας είχαμε αισθητήρα DS18B20.

Αντιγράφουμε τον παραπάνω κώδικα, με μία αλλαγή στην `read_temp(void)`: η συνάρτηση πλέον επιστρέφει σε `unsigned int` 16 bits την raw τιμή που επιστρέφει ο αισθητήρας, προκειμένου να διευκολυνθεί η εκτύπωση της θερμοκρασίας. Για την χρήση LCD οθόνης χρειάζεται να χρησιμοποιήσουμε port expander, άρα αντιγράφουμε και τον boilerplate κώδικα επικοινωνίας από παλιότερη άσκηση.

Δημιουργούμε νέα συνάρτηση για την εκτύπωση θερμοκρασίας `lcd_temp(val, precision)`: διαχωρίζουμε την raw τιμή σε ακέραιο και δεκαδικό μέρος, εκτυπώνουμε ολόκληρο το ακέραιο μέρος και μετά `precision` ψηφία του δεκαδικού. Στο τέλος εκτυπώνουμε το σύμβολο $^oC$.

Στην συνάρτηση `main`, διαβάζουμε συνέχεια τιμή θερμοκρασίας και, αν αυτή εκφράζει σφάλμα ανάγνωσης εκτυπώνουμε το ζητούμενο error message, αλλιώς εκτυπώνουμε την θερμοκρασία μέσω της συναρτησης.

> Για να επιταχύνουμε τον κώδικα εκτύπωσης, αντί να καθαρίζουμε την LCD μετακινούμε τον κέρσορα στην πρώτη στήλη της πρώτης γραμμής μέσω `lcd_command(instr)`.

```c
/*
 * main.c
 *
 * Created: 11/22/2024 11:05:01 AM
 *  Author: User
 */

#include <xc.h>

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

typedef uint8_t bool;
#define true 1
#define false 0

/* LCD DISPLAY THROUGH PORT EXPANDER */
/* {{{ */
#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

#define NOP() do { __asm__ __volatile__ ( "nop "); } while (0)

// 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

```

```
52 //---------------- Master Receiver ----------------
53 #define TW_MR_SLA_ACK 0x40
54 #define TW_MR_SLA_NACK 0x48
55 #define TW_MR_DATA_NACK 0x58
56
57 #define TW_STATUS_MASK 0b11111000
58 #define TW_STATUS (TWSR0 & TW_STATUS_MASK)
59
60 //initialize TWI clock
61 void twi_init(void)
62 {
63         TWSR0 = 0; // PRESCALER_VALUE=1
64         TWBR0 = TWBR0_VALUE; // SCL_CLOCK 100KHz
65 }
66
67 // Read one byte from the twi device (request more data from device)
68 unsigned char twi_readAck(void)
69 {
70         TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
71         while(!(TWCR0 & (1<<TWINT)));
72         return TWDR0;
73 }
74
75 //Read one byte from the twi device, read is followed by a stop condition
76 unsigned char twi_readNak(void)
77 {
78         TWCR0 = (1<<TWINT) | (1<<TWEN);
79         while(!(TWCR0 & (1<<TWINT)));
80         return TWDR0;
81 }
82
83 // Issues a start condition and sends address and transfer direction.
84 // return 0 = device accessible, 1= failed to access device
85 unsigned char twi_start(unsigned char address)
86 {
87         uint8_t twi_status;
88
89         // send START condition
90         TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
91
92         // wait until transmission completed
93         while(!(TWCR0 & (1<<TWINT)));
94
95         // check value of TWI Status Register.
96         twi_status = TW_STATUS & 0xF8;
97         if ( (twi_status != TW_START) && (twi_status != TW_REP_START)) return 1;
98
99         // send device address
100        TWDR0 = address;
101        TWCR0 = (1<<TWINT) | (1<<TWEN);
102
```

```
        // 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;
        }
```

```
151 }
152
153 // Send one byte to twi device, Return 0 if write successful or 1 if write failed
154 unsigned char twi_write( unsigned char data )
155 {
156         // send data to the previously addressed device
157         TWDR0 = data;
158         TWCR0 = (1<<TWINT) | (1<<TWEN);
159
160         // wait until transmission completed
161         while(!(TWCR0 & (1<<TWINT)));
162         if( (TW_STATUS & 0xF8) != TW_MT_DATA_ACK) return 1;
163         return 0;
164 }
165
166 // Send repeated start condition, address, transfer direction
167 //Return: 0 device accessible
168 // 1 failed to access device
169 unsigned char twi_rep_start(unsigned char address)
170 {
171         return twi_start( address );
172 }
173
174 // Terminates the data transfer and releases the twi bus
175 void twi_stop(void)
176 {
177         // send stop condition
178         TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
179
180         // wait until stop condition is executed and bus released
181         while(TWCR0 & (1<<TWSTO));
182 }
183
184 uint8_t LAST;
185
186 void PCA9555_0_write(PCA9555_REGISTERS reg, uint8_t value)
187 {
188         twi_start_wait(PCA9555_0_ADDRESS + TWI_WRITE);
189         twi_write(reg);
190         twi_write(value);
191         twi_stop();
192         LAST = value;
193         //if (reg != REG_CONFIGURATION_0) exit(0);
194 }
195
196 uint8_t PCA9555_0_read(PCA9555_REGISTERS reg)
197 {
198         uint8_t ret_val;
199         twi_start_wait(PCA9555_0_ADDRESS + TWI_WRITE);
200         twi_write(reg);
201         twi_rep_start(PCA9555_0_ADDRESS + TWI_READ);
```

```
202         ret_val = twi_readNak();
203         twi_stop();
204         return ret_val;
205 }
206 
207 void flash ()
208 {
209         _delay_us(50);
210         uint8_t tmp = PCA9555_0_read(REG_INPUT_0);
211         PCA9555_0_write(REG_OUTPUT_0, tmp | (1 << 3));
212         _delay_us(50);
213         PCA9555_0_write(REG_OUTPUT_0, tmp & ~(1 << 3));
214 }
215 
216 void write_2_nibbles(uint8_t data){
217         uint8_t temp = LAST & 0x0f;
218         uint8_t out = data & 0xf0 | temp;
219         PCA9555_0_write(REG_OUTPUT_0, out);
220         flash();
221 
222         out = (data << 4) & 0xf0 | temp;
223         PCA9555_0_write(REG_OUTPUT_0, out);
224         flash();
225 }
226 
227 void lcd_data (uint8_t data)
228 {
229         uint8_t tmp = LAST;
230         PCA9555_0_write(REG_OUTPUT_0, tmp | (1 << 2));
231         write_2_nibbles(data);
232         _delay_us(500);
233 }
234 
235 void lcd_command (uint8_t instr)
236 {
237         uint8_t tmp = LAST;
238         PCA9555_0_write(REG_OUTPUT_0, tmp & ~(1 << 2));
239         write_2_nibbles(instr);
240         _delay_us(500);
241 }
242 
243 void lcd_clear_display(){
244         lcd_command(0x01);
245         _delay_ms(200);
246 }
247 
248 void lcd_init ()
249 {
250         _delay_ms(200);
251 
252         uint8_t out = 0x30;
```

```
253         for (int i=0; i<3; ++i) {
254                 PCA9555_0_write(REG_OUTPUT_0, out);
255                 flash();
256                 _delay_us(250);
257         }
258         PCA9555_0_write(REG_OUTPUT_0, 0x20);
259         flash();
260         _delay_us(250);
261 
262         lcd_command(0x28);
263         lcd_command(0x0c);
264         lcd_clear_display();
265         lcd_command(0x06);
266 }
267 
268 void lcd_number(uint16_t number){
269         uint8_t digits[18];
270         int i = 0;
271         for (; number; number >>= 1)
272                 digits[i++] = number & 1;
273         for (i-=1; i>=0; --i)
274                 lcd_data(digits[i] + '0');
275 }
276 
277 void lcd_string (const char* str)
278 {
279         lcd_command(0x02);
280         for (; *str; str++)
281                 lcd_data(*str);
282 }
283 
284 void lcd_temp (int16_t val, int precision)
285 {
286         char tmp[17];
287         int idx = 0;
288         lcd_command(0x02);
289 
290         if (val < 0) {
291                 lcd_data('-');
292                 val = -val;
293         }
294 
295         int   int_part  = val >> precision;
296         float frac_part =  ((float)(val) / (1 << precision)) - int_part;
297 
298         if (int_part) {
299                 for (; int_part; int_part /= 10)
300                         tmp[idx++] = (int_part % 10) + '0';
301                 for (idx -= 1; idx >= 0; --idx)
302                         lcd_data(tmp[idx]);
303         } else {
```

```
                lcd_data('0');
        }

        lcd_data('.');
        for (int i=0; i<precision; ++i) {
                frac_part *= 10;
                lcd_data((int)(frac_part) + '0');
                frac_part -= (int)(frac_part);
        }

        lcd_data(0b11011111); lcd_data('C');
}
/* }}} */

#define SET(x, b)   do { (x) |=   1 << (b);  } while (0)
#define CLEAR(x, b) do { (x) &= ~(1 << (b)); } while (0)

/* TEMPLATE CODE FROM GIVEN ASSEMBLY */
/* {{{ */
bool one_wire_reset (void)
{
        SET(DDRD, 4); // set output

        CLEAR(PORTD, 4); // sent 0
        _delay_us(480);

        CLEAR(DDRD, 4); // set input
        CLEAR(PORTD, 4); // disable pull-up
        _delay_us(100);

        int tmp = PIND; // read PORTD
        _delay_us(380);
        return (tmp & (1 << 4)) == 0;
}

uint8_t one_wire_receive_bit (void)
{
        SET(DDRD, 4); // Set output
        CLEAR(PORTD, 4);
        _delay_us(2);

        CLEAR(DDRD, 4); // set input
        CLEAR(PORTD, 4); // disable pull-up
        _delay_us(10);

        uint8_t ret = PIND & (1 << 4);
        _delay_us(49);
        return ret >> 4;
}

void one_wire_transmit_bit (uint8_t in)
```

```
{
        SET(DDRD, 4); // Set output
        CLEAR(PORTD, 4);
        _delay_us(2);

        in &= 1; // keep only LSB
        if (in) SET(PORTD, 4);
        else    CLEAR(PORTD, 4);

        _delay_us(58);
        CLEAR(DDRD, 4); // set input
        CLEAR(PORTD, 4); // disable pull-up
        _delay_us(1);
}

uint8_t one_wire_receive_byte (void)
{
        uint8_t ret = 0;
        for (int i=0; i<8; ++i)
                ret |= (one_wire_receive_bit() << i);
        return ret;
}

void one_wire_transmit_byte (uint8_t in)
{
        for (int i=0; i<8; ++i, in >>= 1)
                one_wire_transmit_bit(in);
}
/* }}} */

#define TEMP_ERR 0x8000

int16_t read_temp (void)
{
        int16_t ret = 0;

        if (one_wire_reset() != 1) return TEMP_ERR;
        one_wire_transmit_byte(0xCC);
        one_wire_transmit_byte(0x44);
        // lcd_string("Waiting...");
        while (one_wire_receive_bit() != 1);
        // lcd_string("Finished!");

        if (one_wire_reset() != 1) return TEMP_ERR;
        one_wire_transmit_byte(0xCC);
        one_wire_transmit_byte(0xBE);

        ret |= one_wire_receive_byte();
        ret |= one_wire_receive_byte() << 8;
        return ret;
}
```

```
406
407 const char err_msg[] = "NO Device";
408
409
410 int main(void) {
411         twi_init();
412         PCA9555_0_write(REG_CONFIGURATION_0, 0x00); //Set EXT_PORT0 as output
413         lcd_init();
414         lcd_clear_display();
415
416         bool is_prev_err = false;
417         while (1) {
418                 int16_t temp = read_temp();
419                 if (temp != TEMP_ERR) {
420                         is_prev_err = false;
421                         lcd_temp(temp, 4);
422                 } else if (!is_prev_err) {
423                         is_prev_err = true;
424                         lcd_string(err_msg);
425                 }
426                 _delay_ms(50);
427         }
428
429         return 0;
430 }
431
```