8η εργαστηριακή άσκηση

Φωτεινή Κυριακοπούλου 03120182
Ιωάννης Μπουφίδης 03120162

Ζήτημα 8

```
#define F_CPU 16000000UL

#include<avr/io.h>
#include<avr/interrupt.h>
#include<util/delay.h>
#include<math.h>
#include<string.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, read is followed by a stop condition
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;
	TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN); // send START condition
	while(!(TWCR0 & (1<<TWINT)));		// wait until transmission completed
	twi_status = TW_STATUS & 0xF8;		// check value of TWI Status Register
	if ( (twi_status != TW_START) && (twi_status != TW_REP_START)) return 1;
	TWDR0 = address;					// send device address
	TWCR0 = (1<<TWINT) | (1<<TWEN);
	// wail until transmission completed and ACK/NACK has been received
	while(!(TWCR0 & (1<<TWINT)));
	twi_status = TW_STATUS & 0xF8; // check value of TWI Status Register
	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 ) {
		TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN); // send START condition
		while(!(TWCR0 & (1<<TWINT))); // wait until transmission completed

		twi_status = TW_STATUS & 0xF8; // check value of TWI Status Register
		if ( (twi_status != TW_START) && (twi_status != TW_REP_START)) continue;

		TWDR0 = address; // send device address
		TWCR0 = (1<<TWINT) | (1<<TWEN);

		while(!(TWCR0 & (1<<TWINT))); // wail until transmission completed

		twi_status = TW_STATUS & 0xF8; // check value of TWI Status Register
		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;
}

uint8_t scan_row(uint8_t row) {
    PCA9555_0_write(REG_OUTPUT_1, row); // set corresponding row to 0

    uint8_t keys = PCA9555_0_read(REG_INPUT_1);
    return keys; // return state of columns and rows
}

uint16_t scan_keypad(void) {
    uint16_t value = 0; // current state of the 16 buttons
    uint8_t keypad = 0;

    keypad = scan_row(0xFE); // check row 1
    if (keypad == 0xFE) value |= 0xF; // no button is pressed
    // find which button is pressed and move it to the appropriate position
    else value |= ((keypad & 0xF0) >> 4);
    value <<= 4; // leave space for the next check
    // repeat for remaining rows
    keypad = scan_row(0xFD); // check row 2
    if (keypad == 0xFD) value |= 0xF;
    else value |= ((keypad & 0xF0) >> 4);
    value <<= 4;
    keypad = scan_row(0xFB); // check row 3
    if (keypad == 0xFB) value |= 0xF;
    else value |= ((keypad & 0xF0) >> 4);
    value <<= 4;
    keypad = scan_row(0xF7); // check row 4
    if (keypad == 0xF7) value |= 0xF;
    else value |= ((keypad & 0xF0) >> 4);

    return value;
}


uint16_t prev = 0xffff;
uint16_t scan_keypad_rising_edge() {
    // scan keypad twice with a delay of 20 ms
    uint16_t check1 = scan_keypad();
    _delay_ms(20);
    uint16_t check2 = scan_keypad();

    uint16_t final;
    // if second check is different return the most recent state
    if (check1 != check2) final = check2;
    else final = check1; // else the state of buttons hasn't changed

    // do not return the current state more than once
    if (final == prev) return 0xffff;
    prev = final;
```

```
    return final;
}

// convert pressed button to ascii
unsigned char keypad_to_ascii() {
    uint16_t x = scan_keypad_rising_edge(); // state of keypad
    unsigned char c = 0;
    switch (x) {// return corresponding character
        // 1st row
        case(0b1111111111111110): c='1'; break;
        case(0b1111111111111101): c='2'; break;
        case(0b1111111111111011): c='3'; break;
        case(0b1111111111110111): c='A'; break;
        // 2nd row
        case(0b1111111111101111): c='4'; break;
        case(0b1111111111011111): c='5'; break;
        case(0b1111111110111111): c='6'; break;
        case(0b1111111101111111): c='B'; break;
        // 3rd row
        case(0b1111111011111111): c='7'; break;
        case(0b1111110111111111): c='8'; break;
        case(0b1111101111111111): c='9'; break;
        case(0b1111011111111111): c='C'; break;
        // 4th row
        case(0b1110111111111111): c='*'; break;
        case(0b1101111111111111): c='0'; break;
        case(0b1011111111111111): c='#'; break;
        case(0b0111111111111111): c='D'; break;
    }
    return c;
}

// swap macro
uint8_t swap(uint8_t x) {
    return (uint8_t)((x >> 4) | (x << 4));
}

void write_2_nibbles(uint8_t command) {
    // write MSB nibble
    PORTD = ((PIND & 0X0f) | (command & 0xf0));

    PORTD |= (1 << PD3);  // Enable pulse
    _delay_us(1);
    PORTD &= ~(1 << PD3); // Disable pulse

    // write LSB nibble
    PORTD = ((PIND & 0X0f) | (swap(command) & 0xf0));

    PORTD |= (1 << PD3);
    _delay_us(1);
    PORTD &= ~(1 << PD3);
}
```

```
// send a command to lcd's controller (4-bit mode)
void lcd_command(uint8_t command) {
    PORTD &= ~(1 << PD2);  // LCD_RS = 0 => instruction
    write_2_nibbles(command);  // send instruction
    _delay_us(250);  // wait 250us
}

// send data to lcd's controller (4-bit mode)
void lcd_data(uint8_t data) {
    PORTD |= (1 << PD2);  // LCD_RS = 1 => data
    write_2_nibbles(data);  // send instruction
    _delay_us(250);  // wait 250us
}

void lcd_clear_display() {
    lcd_command(0x01);
    _delay_ms(5);
}

// lcd monitor set-up
void lcd_init() {
    _delay_ms(200);  // wait 200ms

    // set microcontroler in 4-bit mode

    for (int i=0; i<3; i++) {  // 3 times
        PORTD = 0x30;  // switch to 8-bit mode
        PORTD |= (1 << PD3);  // Enable pulse
        _delay_us(1);
        PORTD &= ~(1 << PD3); // Disable pulse
        _delay_us(250);
    }

    PORTD = 0x20;  // switch to 4-bit mode
    PORTD |= (1 << PD3);  // Enable pulse
    _delay_us(1);
    PORTD &= ~(1 << PD3); // Disable the enable pulse
    _delay_us(250);

    lcd_command(0x28);   // 5x8 dots, 2 lines
    lcd_command(0x0c);   // display on, cursor-blinking off
    lcd_clear_display(); // clear display
    lcd_command(0x06);   // increase address, no display shift
}

int one_wire_reset() {
    DDRD |= (1 << PD4);     // set PD4 as output
    PORTD &= (0 << PD4);
    _delay_us(480);         // 480 usec reset pulse

    DDRD &= (0 << PD4);     // set PD4 as input
    PORTD &= (0 << PD4);    // disable pull-up
    _delay_us(100);         // wait 100 usec for connected devices
```

```
                                // to transmit the presence pulse

    uint8_t portd = PIND;    // read PORTD
    _delay_us(380);          // wait for 380 usec

    // if a connected device is detected (PD4=0) return 1
    if ((portd & (1 << PD4)) == 0) return 1;
    // else return 0
    else return 0;
}

uint8_t one_wire_receive_bit() {
    DDRD |= (1 << PD4);      // set PD4 as output
    PORTD &= (0 << PD4);
    _delay_us(2);            // time slot 2 usec

    DDRD &= (0 << PD4);      // set PD4 as input
    PORTD &= (0 << PD4);     // disable pull-up
    _delay_us(10);           // wait 10 usec

    uint8_t portd = PIND;
    _delay_us(49);           // delay 49 usec to meet the standards
    return ((portd & 0x10) >> 4);  // return PD4
}

void one_wire_transmit_bit(uint8_t tr_bit) {
    DDRD |= (1 << PD4);      // set PD4 as output
    PORTD &= (0 << PD4);
    _delay_us(2);            // time slot 2 usec

    // if transmitted bit is '1', set PD4
    if (tr_bit == 1) PORTD |= (1 << PD4);
    else PORTD &= (0 << PD4);
    _delay_us(58);  // wait 58 usec for connected device to sample the line

    DDRD &= (0 << PD4);      // set PD4 as input
    PORTD &= (0 << PD4);     // disable pull-up
    _delay_us(1);            // recovery time 1 usec
}

uint8_t one_wire_receive_byte() {
    uint8_t byte = 0, b = 0;
    for (int i = 0; i < 8; i++) {
        byte >>= 1;                 // shift received byte right
        b = one_wire_receive_bit();
        if (b == 1) byte |= 0x80;   // if received bit is '1', set byte's MSB
    }
    return byte;
}

void one_wire_transmit_byte(uint8_t byte) {
    uint8_t tr_bit;
    for (int i=0; i<8; i++) {
```

```
        tr_bit = byte & 0x01;
        one_wire_transmit_bit(tr_bit);    // transmit byte's LSB
        byte >>= 1;                       // shift transmitted byte right
    }
}

//display and transmit byte
void lcd_usart(uint8_t data) {
    lcd_data(data);
    usart_transmit(data);
}

int temperature() {
    if (one_wire_reset() == 0) return 0x8000; //no device connected
    one_wire_transmit_byte(0xCC); //skip device selection
    one_wire_transmit_byte(0x44); //start measurement

    uint8_t finished = 0;
    while (!finished) { //wait for measurement to be completed
        finished = one_wire_receive_bit();
    }
    if (one_wire_reset() == 0) return 0x8000;
    one_wire_transmit_byte(0xCC);

    //read 16-bit value of measurement
    one_wire_transmit_byte(0xBE);
    uint8_t temp_low = one_wire_receive_byte(); //low-byte
    uint8_t temp_high = one_wire_receive_byte(); //high-byte

    uint16_t value = 0;
    //if sign bits are set (negative temperature), set the 12th bit of the 16-bit
    //value
    if ((temp_high & 0xF8) == 0xF8) value = 1;
    //combine the low and high bytes
    value <<=3;
    value |= (temp_high & 0x07);
    value <<= 8;
    value |= temp_low;

    return value;
}

void lcd_usart_temp(uint16_t temp, uint16_t offset) {
    //initialize LCD screen
    DDRD |= 0xFF;
    lcd_init();
    _delay_ms(100);

    //if 12th bit not set - positive temperature
    if((temp & 0x0800) != 0x0100) lcd_data('+');
    //else negative temperature
    else {
        lcd_data('-');
```

```
        temp = ~(temp) + 1; //2's complement
        temp &= 0x7F;
        //2's complement -> sign bit set to '0'
    }

    temp += (offset << 4);

    //compute the fractional part (bits 0-3)
    float dec = 0;
    for (int i=4; i>0; i--) {
        if ((temp & 0x01) == 0x01) dec += 1/pow(2,i);
        temp >>= 1;
    }


    int h = temp/100;
    if (h != 0) lcd_usart(h + '0');

    int t = (temp%100)/10;
    if (t!=0 || h!=0) lcd_usart(t + '0');

    int o = temp%10;
    lcd_usart(o + '0');

    lcd_usart('.');
    //if there's a fractional part
    if (dec != 0) {
        int digit = 0;
        for (int i = 0; i < 3; i++) {
            digit = dec*10;
            dec = dec*10 - (float)digit;
            if (dec >= 0.5 && i == 2) digit++; //round last digit
            lcd_usart(digit + '0'); //display each digit
        }
    }
    else {
        for (int i=0; i<3; i++) lcd_usart('0'); //else display zeros
    }
    lcd_data(' ');
}

void lcd_usart_pressure(int press) {
    //get digits of pressure value
    if (press > 10) lcd_usart((press/10) + '0');
    lcd_usart((press%10) + '0');
    lcd_command (0xC0); //new line command
}

char nurse[] = {'N', 'U', 'R', 'S', 'E', 'C', 'A', 'L', 'L'};
char temp[] =  {'C', 'H', 'E', 'C', 'K', 'T', 'E', 'M', 'P'};
char press[] =  {'C', 'H', 'E', 'C', 'K',
                 'P', 'R', 'E', 'S', 'S', 'U', 'R', 'E'};
char ok[] = {'O', 'K'};
```

```
enum states {NURSECALL, CHECKPRESSURE, CHECKTEMP, OK};

//display and transmit status
void lcd_usart_status(enum states status) {
    switch (status) {
        case NURSECALL: {
            for(int i = 0; i < 9; i++) lcd_usart(nurse[i]);
            break;
        }
        case CHECKTEMP: {
            for(int i = 0; i < 9; i++) lcd_usart(temp[i]);
            break;
        }
        case CHECKPRESSURE: {
            for(int i = 0; i < 13; i++) lcd_usart(press[i]);
            break;
        }
        case OK: {
            for(int i = 0; i < 2; i++) lcd_usart(ok[i]);
            break;
        }
    }
}

/* Routine: usart_init
Description:
This routine initializes the
usart as shown below.
------- INITIALIZATIONS -------
Baud rate: 9600 (Fck= 8MH)
Asynchronous mode
Transmitter on
Reciever on
Communication parameters: 8 Data ,1 Stop, no Parity
--------------------------------
parameters: ubrr to control the BAUD.
return value: None.*/
void usart_init(unsigned int ubrr){
    UCSR0A=0;
    UCSR0B=(1<<RXEN0)|(1<<TXEN0);
    UBRR0H=(unsigned char)(ubrr>>8);
    UBRR0L=(unsigned char)ubrr;
    UCSR0C=(3 << UCSZ00);
    return;
}

/* Routine: usart_transmit
Description:
This routine sends a byte of data
5
using usart.
parameters:
data: the byte to be transmitted
```

```
return value: None. */
void usart_transmit(uint8_t data){
    while(!(UCSR0A&(1<<UDRE0)));
    UDR0=data;
}

//transmit string byte-by-byte
void usart_transmit_message(const char *command){
    int i = 0;
    while(command[i] != '\0'){
        usart_transmit(command[i]);
        ++i;
    }
}

/* Routine: usart_receive
Description:
This routine receives a byte of data
from usart.
parameters: None.
return value: the received byte */
uint8_t usart_receive(){
    while(!(UCSR0A&(1<<RXC0)));
    return UDR0;
}

//receive string byte-by-byte and store it in array received[]
void usart_receive_message(char received[]) {
    uint8_t received_byte;
    int i = 0;
    while((received_byte=usart_receive())!='\n'){
                received[i]=received_byte;
                i++;
            }
}

char success_msg[] = {'S', 'u', 'c', 'c', 'e', 's', 's'};
char fail_msg[] = {'F', 'a', 'i', 'l'};
char ok_msg[] = {'2', '0', '0', ' ', 'O', 'K'};

//display server's response on a request
void esp_messages(char received[], int request){
    DDRD |= 0b11111111;
    lcd_init();
    lcd_data(request + '0'); //display number of request
    lcd_data('.');

    if(received[1]=='S') {
        for (int i=0; i<7; i++)
            lcd_data(success_msg[i]);
    }
    else if(received[1]=='F') {
           for (int i=0; i<4; i++)
```

```
            lcd_data(fail_msg[i]);
    }
    else if(received[2]=='0') {
        for (int i=0; i<6; i++)
            lcd_data(ok_msg[i]);
    }
    else lcd_clear_display();
}

int main() {
    int n = 7;
    char rec[n]; //receiver buffer
    int press;
    char key;
    int check_rest = 1;
    enum states status = OK;
    uint16_t temp;

    memset(rec,'\0',n); //initialize buffer with null bytes

    usart_init(103);
    usart_transmit_message("ESP:restart\n");

    //initialize LCD
    DDRD |= 0b11111111;
    lcd_init();
    _delay_ms(100);

    // set voltage reference to AVCC and select ADC0 channel
    ADMUX = (1 << REFS0);
    // enable ADC and set prescaler to 128 for maximum resolution
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

    twi_init();
    PCA9555_0_write(REG_CONFIGURATION_1, 0xf0);

    while(1){ //to repeat entire procedure
        usart_transmit_message("ESP:connect\n");
        usart_receive_message(rec);
        esp_messages(rec, 1);
        _delay_ms(2000);

        usart_transmit_message("ESP:url:\"http://192.168.1.250:5000/data\"\n");
        usart_receive_message(rec);
        esp_messages(rec, 2);
        _delay_ms(2000);
        lcd_clear_display();

        //wait a couple seconds to check if patient pressed a key
        key = 0;
        for (int i=0; i<50; i++) {
            key = keypad_to_ascii();
            if (key != 0) break;
```

```
        }


        temp = temperature(); //get patient's temperature
        usart_transmit_message("ESP:payload:[{\"name\": \"temperature\",\"value\":
\"");
        lcd_usart_temp(temp, 11); //add 11 to measured temperature to get a valid
number

        // start an ADC conversion and wait for it to finish
        // if ADSC is set, a conversion is in progress
        ADCSRA |= (1 << ADSC);
        while (ADCSRA & (1 << ADSC));
        // scale the digital ADC to a blood pressure level
        press = (ADC*20)/1023; //get patient's pressure
        usart_transmit_message("\"},{\"name\": \"pressure\",\"value\": \"");
        lcd_usart_pressure(press);

        //patient needs help
        if (key == '1') {
            status = NURSECALL;
            check_rest = 0; //skip remaining checks
        }
        //nurse has arrived
        else if (key == '#' && status == NURSECALL) check_rest = 1;

        //set status according to patient's pressure and temperature levels
        if (check_rest == 1) {
            if (press > 12 || press < 4) status = CHECKPRESSURE;
            else if (temp > ((37-11) << 4) || temp < ((34-11) << 4)) status =
CHECKTEMP;
            else status = OK;
        }
        usart_transmit_message("\"},{\"name\": \"team\", \"value\": \"11\"}, "
                               "{\"name\": \"status\",\"value\": \"");
        lcd_usart_status(status);
        usart_transmit_message("\"}]\n");
        _delay_ms(2000);

        usart_receive_message(rec);
        esp_messages(rec, 3);
        _delay_ms(2000);

        usart_transmit_message("ESP:transmit\n");
        usart_receive_message(rec);
        esp_messages(rec, 4);
        _delay_ms(2000);

    }
    return 0;
}
```