



Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο Μικροϋπολογιστών

7^ο εξάμηνο, Ακαδημαϊκή περίοδος 2025-2026

8^η Εργαστηριακή Άσκηση

Συνδυαστική/Επαναληπτική άσκηση – Εφαρμογή Internet of Things (στο ntuAboard)

Εργαστηριακή Αναφορά

Φοιτητές:	Παπαδάτος Αναστάσιος	->	ΑΜ: 03122847
	Σέρτζιο Γκούρι	->	ΑΜ:03122827

Ομάδα: 30

Ημερομηνία: 12/12/2025

Άσκηση 1:

Παρακάτω φαίνεται ο κώδικας **C** που φτιάξαμε:

```
/*
 * main.c
 *
 * Created: 12/9/2025 11:47:58 AM
 * Author: sergio
 */

#define F_CPU 16000000UL // Set CPU frequency

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

//Fsc1=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 TWRD0;
}
//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 TWRD0;
}
// 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
    TWRD0 = address;
    TWCR0 = (1<<TWINT) | (1<<TWEN);
    // wait 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
        TWRD0 = address;
        TWCR0 = (1<<TWINT) | (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_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 lcd_data) {
    uint8_t temp;
    // Send the high nibble, Keep lower 4 bits of PIND and set high nibble of lcd_data

```

```

temp = (PCA9555_0_read(REG_OUTPUT_0) & 0x0F) | (lcd_data & 0xF0);
PCA9555_0_write(REG_OUTPUT_0, temp);
// Output the high nibble to PORTD, Enable pulse high
PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
_delay_us(1); // Small delay to let the signal settle
// Enable pulse low
PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
// Send the low nibble
lcd_data <<= 4; // Move low nibble to high nibble position
// Keep lower 4 bits of PIND and set high nibble of new lcd_data
temp = (PCA9555_0_read(REG_OUTPUT_0) & 0x0F) | (lcd_data & 0xF0);
PCA9555_0_write(REG_OUTPUT_0, temp);
// Output the low nibble to PORTD, Enable pulse high
PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
_delay_us(1); // Small delay to let the signal settle
// Enable pulse low
PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
}
void lcd_data(uint8_t data)
{
    uint8_t current_state = PCA9555_0_read(REG_OUTPUT_0);
    PCA9555_0_write(REG_OUTPUT_0, current_state | 0x04);
    write_2_nibbles(data);
    _delay_us(100);
}
void lcd_command(uint8_t data)
{
    uint8_t current_state = PCA9555_0_read(REG_OUTPUT_0);
    PCA9555_0_write(REG_OUTPUT_0, current_state & 0xFB);
    write_2_nibbles(data);
    _delay_ms(2);
}
void lcd_clear_display()
{
    uint8_t clear_disp = 0x01; // Clear display command
    lcd_command(clear_disp);
    _delay_ms(5); // Wait 5 msec
    return;
}
void lcd_init() {
    _delay_ms(200);
    // Send 0x30 command to set 8-bit mode (three times)
    PCA9555_0_write(REG_OUTPUT_0, 0x30); // Set command to switch to 8-bit mode
    // Enable pulse
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
    _delay_us(1);
    // Clear enable
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
    PCA9555_0_write(REG_OUTPUT_0, 0x30); // Repeat command to ensure mode set
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
    _delay_us(1);
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
    _delay_us(30);
    PCA9555_0_write(REG_OUTPUT_0, 0x30); // Repeat once more
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
    _delay_us(1);
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
    _delay_us(30);
}

```

```

    // Send 0x20 command to switch to 4-bit mode
    PCA9555_0_write(REG_OUTPUT_0,0x20);
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
    _delay_us(1);
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
    _delay_us(30);
    // Set 4-bit mode, 2 lines, 5x8 dots
    lcd_command(0x28);
    // Display ON, Cursor OFF
    lcd_command(0x0C);
    // Clear display
    lcd_clear_display();
    // Entry mode: Increment cursor, no display shift
    lcd_command(0x06);
}
void lcd_print(const char *str) {
    while (*str) {
        lcd_data(*str++);
    }
}
/* 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){
    UCSRA=0;
    UCSRB=(1<<RXEN0)|(1<<TXEN0);
    UBRR0H=(unsigned char)(ubrr>>8);
    UBRR0L=(unsigned char)ubrr;
    UCSRC=(3 << UCSZ00);
    return;
}

/* Routine: usart_transmit
Description:
This routine sends a byte of data
using usart.
parameters:
data: the byte to be transmitted
return value: None. */
void usart_transmit(uint8_t data){
    while(!(UCSRA&(1<<UDRE0)));
    UDR0=data;
}
void usart_print(const char *str) {
    while (*str) {
        usart_transmit(*str++);
    }
}

```

```

/* 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;
}

void usart_receive_string(char *buf, uint8_t len) {
    char c;
    while ((c = usart_receive()) != '\n') {
        if (c != '\r' && --len > 0) {
            *buf++ = c;
        }
    }
    *buf = '\0'; //string (Null terminator)
}

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

    usart_init(103);

    char buf[20];
    char buffer[20];

    while(1)
    {
        //char buf[20];
        //char buffer[20];
        // 1) Send connect command
        usart_print("ESP:connect\r\n");
        usart_receive_string(buf,20);

        if (strcmp(buf, "Success") == 0) {

            lcd_print("1.Success");
            _delay_ms(2000);

            lcd_clear_display();
        }
        else {
            lcd_print("1.Fail");
            _delay_ms(2000);
        }
        // 2) Send URL command
        lcd_clear_display();

        usart_print("ESP:url:\n"http://192.168.1.250:5000/data"\r\n");
        usart_receive_string(buffer,20);
    }
}

```

```

        if (strcmp(buffer, "Success") == 0) {
            lcd_print("2.Success");
            _delay_ms(2000);

            lcd_clear_display();
            break;
        }
        else {
            lcd_print("2.Fail");
            _delay_ms(2000);

            lcd_clear_display();
        }
    }
}

```

Σχόλια/Περιγραφή του κώδικα:

Η άσκηση ακολουθεί την εξής πορεία για την υλοποίηση της. Αρχικά απ τις συναρτήσεις που μας δίνονται στον οδηγό για την εργαστηριακή άσκηση 8, παρατηρούμε ότι μπορούμε να επικοινωνήσουμε στέλνοντας 1 byte τη φορά ή λαμβάνοντας ένα byte τη φορά. Για να επιλύσουμε αυτό το ζήτημα, φτιάξαμε 2 νέες συναρτήσεις την `usart_print` και την `usart_receive_string` έτσι ώστε να μπορούμε μέσω 2 διαφορετικών buffers να στέλνουμε και να λαμβάνουμε δεδομένα τα οποία είναι strings. Αυτά τα strings είναι οι εντολές που στέλνουμε και οι απαντήσεις που λαμβάνουμε απ τον σερβερ μας (ή στη προκειμένη περίπτωση απ τον online Arduino μας). Απο εκεί και πέρα στέλνουμε διαδοχικά τις εντολές για να πραγματοποιήσουμε την σύνδεση και απο εκεί και πέρα περιμένουμε να λάβουμε απάντηση απ τον σερβερ μας. Πριν όμως απο κάθε απάντηση που λάμβανουμε, οφείλουμε να καθαρίσουμε τους καταχωρητές που πραγματοποιούν την σύνδεση έτσι ώστε να περιμένει κάθε φορά να λάβει απάντηση και να μην διαβάσει σκουπίδια που μπορεί να υπάρχουν στον buffer απο προηγούμενες επικοινωνίες.

Άσκηση 3:

Παρακάτω φαίνεται ο κώδικας **C** που φτιάξαμε:

```

#define F_CPU 16000000UL // Set CPU frequency

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdbool.h>
#include <stdio.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
#define MAX_CVP 20

```

```

//Fsc1=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 0b1111000
#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 TWDRO;
}
//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 TWDRO;
}
// 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);
    // wait 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);
        // wait 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;
}
// ----- LCD -----
void write_2_nibbles(uint8_t lcd_data) {
    uint8_t temp;
    // Send the high nibble, Keep lower 4 bits of PIND and set high nibble of lcd_data
    temp = (PCA9555_0_read(REG_OUTPUT_0) & 0x0F) | (lcd_data & 0xF0);
    PCA9555_0_write(REG_OUTPUT_0 , temp);
    // Output the high nibble to PORTD, Enable pulse high
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
    _delay_us(1); // Small delay to let the signal settle
    // Enable pulse low
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
    // Send the low nibble
    lcd_data <<= 4; // Move low nibble to high nibble position
    // Keep lower 4 bits of PIND and set high nibble of new lcd_data
    temp = (PCA9555_0_read(REG_OUTPUT_0) & 0x0F) | (lcd_data & 0xF0);
    PCA9555_0_write(REG_OUTPUT_0 , temp);
    // Output the low nibble to PORTD, Enable pulse high
    PCA9555_0_write(REG_OUTPUT_0 , PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
    _delay_us(1); // Small delay to let the signal settle
    // Enable pulse low
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
}
void lcd_data(uint8_t data)
{
    uint8_t current_state = PCA9555_0_read(REG_OUTPUT_0);
    PCA9555_0_write(REG_OUTPUT_0, current_state | 0x04);
    write_2_nibbles(data);
}

```

```

        _delay_us(100);
    }
    void lcd_command(uint8_t data)
    {
        uint8_t current_state = PCA9555_0_read(REG_OUTPUT_0);
        PCA9555_0_write(REG_OUTPUT_0, current_state & 0xFB);
        write_2_nibbles(data);
        _delay_ms(2);
    }
    void lcd_clear_display()
    {
        uint8_t clear_disp = 0x01; // Clear display command
        lcd_command(clear_disp);
        _delay_ms(5); // Wait 5 msec
        return;
    }
    void lcd_init() {
        _delay_ms(200);
        // Send 0x30 command to set 8-bit mode (three times)
        PCA9555_0_write(REG_OUTPUT_0, 0x30); // Set command to switch to 8-bit mode
        // Enable pulse
        PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
        _delay_us(1);
        // Clear enable
        PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
        PCA9555_0_write(REG_OUTPUT_0, 0x30); // Repeat command to ensure mode set
        PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
        _delay_us(1);
        PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
        _delay_us(30);
        PCA9555_0_write(REG_OUTPUT_0, 0x30); // Repeat once more
        PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
        _delay_us(1);
        PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
        _delay_us(30);
        // Send 0x20 command to switch to 4-bit mode
        PCA9555_0_write(REG_OUTPUT_0, 0x20);
        PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));
        _delay_us(1);
        PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));
        _delay_us(30);
        // Set 4-bit mode, 2 lines, 5x8 dots
        lcd_command(0x28);
        // Display ON, Cursor OFF
        lcd_command(0x0C);
        // Clear display
        lcd_clear_display();
        // Entry mode: Increment cursor, no display shift
        lcd_command(0x06);
    }
    // Print a string to LCD
    void lcd_print(const char *str) {
        while (*str) {
            lcd_data(*str++);
        }
    }
}
//----- DS18B20 -----
int one_wire_reset(void) {

```

```

        DDRD |= (1 << PD4); // sbi DDRD, PD4 (Set
output)
        PORTD &= ~(1 << PD4); // cbi PORTD, PD4 (Set low)
        _delay_us(480); // 480us delay
        DDRD &= ~(1 << PD4); // cbi DDRD, PD4 (Set input)
        PORTD &= ~(1 << PD4); // cbi PORTD, PD4 (Disable
pull-up)
        _delay_us(100); // 100us delay
        uint8_t port_sample = PIND; // Save PORTD state
        _delay_us(380); // 380us delay
        if (!(port_sample & (1 << PD4))) return 1; // If PD4 = 0 return
1
        else return 0; // If PD4 = 1 return
0
}
uint8_t one_wire_receive_bit(void) {
    DDRD |= (1 << PD4); // sbi DDRD, PD4 (Set
output)
    PORTD &= ~(1 << PD4); // cbi PORTD, PD4 (Set
low)
    _delay_us(2); // time slot 2 usec
    DDRD &= ~(1 << PD4); // cbi DDRD, PD4 (Set input)
    PORTD &= ~(1 << PD4); // cbi PORTD, PD4 (Disable pull-
up)
    _delay_us(10); // wait 10 usec
    uint8_t r24 = 0; // clr r24
    if (PIND & (1 << PD4)) r24 = 1; // r24 = PD4
    _delay_us(49); // delay 49usec
    return r24;
}
void one_wire_transmit_bit(uint8_t r24) {
    DDRD |= (1 << PD4); // sbi DDRD, PD4 (Set output)
    PORTD &= ~(1 << PD4); // cbi PORTD, PD4 (Set low)
    _delay_us(2); // time slot 2 usec
    if (r24 & 0x01) PORTD |= (1 << PD4);
    else PORTD &= ~(1 << PD4); // PD4 = r24[0]
    _delay_us(58); // wait 58 usec
    DDRD &= ~(1 << PD4); // cbi DDRD, PD4 (Set input)
    PORTD &= ~(1 << PD4); // cbi PORTD, PD4 (Disable pull-up)
    _delay_us(1); // recovery time 1 usec
}
uint8_t one_wire_receive_byte()
{
    uint8_t received_byte = 0x00; // Store the byte (8-bit) we received
    for (uint8_t i = 0; i < 8; i++)
    {
        uint8_t received_bit = one_wire_receive_bit();
        // Logical shift left, because DS18B20 send LSB first
        // Logical OR to insert new bit into byte sequence
        received_byte |= (received_bit << i);
    }
    return received_byte;
}
void one_wire_transmit_byte(uint8_t byte_to_transmit)
{
    for (uint8_t i = 0; i < 8; i++)
    {
        // Bit to transmit now in position bit 0

```

```

        uint8_t send_bit = (byte_to_transmit >> i) & 0x01;
        one_wire_transmit_bit(send_bit);
    }
}
//----- Keypad -----
uint8_t scan_row(int row)
{
    uint8_t keypad_row = 0;
    if ((row >= 0) && (row < 4))
    {
        PCA9555_0_write(REG_OUTPUT_1, ~(0x01 << row)); // Check row
        _delay_ms(1);
        keypad_row = PCA9555_0_read(REG_INPUT_1);
        keypad_row = (~keypad_row >> 4) & 0x0F; // Shift, mask and invert input
from keypad
        PCA9555_0_write(REG_OUTPUT_1, 0x0F); // Reset row to High
        _delay_ms(1);
    }
    return keypad_row;
}
uint16_t scan_keypad()
{
    uint16_t keypad=0;
    for(int i=0; i<4; i++) // Check all 4 rows
    {
        keypad = keypad << 4; // Shift and save pressed keys to a 16bit
        keypad |= scan_row(i);
    }
    return keypad;
}
uint16_t pressed_keys = 0x0000; // Save key pressed
uint16_t scan_keypad_rising_edge()
{
    uint16_t pressed_keys_tempo = scan_keypad();
    _delay_ms(15);
    pressed_keys_tempo &= scan_keypad(); // Check again (de-bouncing)
    uint16_t new_pressed_keys = pressed_keys_tempo & ~pressed_keys; // Check new
    pressed_keys = pressed_keys_tempo; // Save the current pressed keys
    return new_pressed_keys; // Return new pressed keys
}
// Table to store ASCII characters
uint8_t key_table[] = { '1', '2', '3', 'A',
    '4', '5', '6', 'B',
    '7', '8', '9', 'C',
    '*', '0', '#', 'D' };
uint8_t keypad_to_ascii(uint16_t keypad)
{
    for(int i = 0; i<16; i++)
    {
        // Return the value of the first 1 in the 16bit variable through the table
        if (keypad & (1 << i)) return key_table[i];
    }
    return 0;
}
//-----GET_TEMP_FUNCTIONS-----
int16_t get_temp()
{
    int connected_device = one_wire_reset(); // Check for connected device

```

```

    if (!connected_device) return 0x8000; // Error, return 0x8000
    one_wire_transmit_byte(0xCC); // Only one device
    one_wire_transmit_byte(0x44); // Read temperature
    while (!one_wire_receive_bit()); // Wait until the above counting
    one_wire_reset(); // Re-initialize
    one_wire_transmit_byte(0xCC);
    one_wire_transmit_byte(0xBE); // Read 16-bit result of temperature value
    uint16_t temp = 0;
    temp |= one_wire_receive_byte(); // 8-bit LSB of the total 16-bit value
    temp |= ((uint16_t)one_wire_receive_byte() << 8); // Get the other 8 bits shifted
8
    return temp;
}
int get_temp_decoded(uint16_t raw_temp)
{
    // The DS18B20 gives a 16-bit signed integer.
    // The last 4 bits are the decimal part (x 0.0625 = 1/16)
    // The first 5 bits (S) are the sign.
    // Simple way to get 1 decimal digit of precision:
    // Multiply by 10 and divide by 16 (>>4)
    // Use cast to (int16_t) so the compiler handles the sign automatically
    int16_t signed_raw = (int16_t)raw_temp;
    // Calculation: (Raw * 10) / 16
    // Use long to avoid overflow during multiplication before division
    return ((long)signed_raw * 10) / 16;
}
// buf:
int read_human_temp() {

    uint16_t temp;
    int temp_decoded; // Temperature in tenths (e.g. 25625 = 25.625 C)

    temp=get_temp();
    if (temp == 0x8000){
        temp_decoded = -999;
    } // Connection error

    else{
        temp_decoded = get_temp_decoded(temp); //get decoded temperature
        if (temp_decoded < 0) temp_decoded = -temp_decoded; // Positive for division
and display
        temp_decoded += 130;

    }
    return temp_decoded;
}
//----- POT0||Human pressure-----
void ADC_init()
{
    //set ADC with Vref = AVcc, ADC0 channel
    ADMUX = (1 << REFS0);
    // enable ADC, prescaler 128
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
}

// Read the DC voltage from POT1 analog filter, and take the digital result
uint16_t get_CVP_pressure()

```

```

{
    ADCSRA |= (1 << ADSC);
    while(ADCSRA & (1 << ADSC));           // Wait for conversion to end
    uint16_t raw_adc = ADC;
    uint16_t pressure = (raw_adc * MAX_CVP) / 1023;
    return pressure;                         // Return the value pressure
}

//----- USART -----
/* Routine: usart_init
Description:
This routine initializes the
usart as shown below.
----- INITIALIZATIONS -----
Baud rate: 9600 (Fck= 8MH)
Asynchronous mode
Transmitter on
Receiver on
Communication parameters: 8 Data ,1 Stop, no Parity
-----
parameters: ubrr to control the BAUD.
return value: None.*/
void usart_init(unsigned int ubrr){
    UCSRA=0;
    UCSRB=(1<<RXEN)|(1<<TXEN);
    UBRRH=(unsigned char)(ubrr>>8);
    UBRRL=(unsigned char)ubrr;
    UCSRC=(3 << UCSZ0);
    return;
}

/* Routine: usart_transmit
Description:
This routine sends a byte of data
using usart.
parameters:
data: the byte to be transmitted
return value: None. */
void usart_transmit(uint8_t data){
    while(!(UCSRA&(1<<UDRE)));
    UDR0=data;
}

// 1. Transmitt string
void usart_print(const char *str) {
    while (*str) {
        usart_transmit(*str++);
    }
}

/* 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(!(UCSRA&(1<<RXC)));

```

```

        return UDR0;
    }

void usart_receive_string(char *buf, uint8_t len) {
    char c;

    while ((c = usart_receive()) != '\n') {

        if (c != '\r' && --len > 0) {
            *buf++ = c;
        }
    }
    *buf = '\0';
}

void uart_flush(void)
{
    uint8_t dummy;

    while (UCSR0A & (1 << RXC0)) {
        dummy = UDR0;
    }
}

int check_nurse_call(int nurse_call_active)
{
    uint16_t key_pressed = scan_keypad_rising_edge();
    char key_char = keypad_to_ascii(key_pressed);

    if (key_char == '0') {
        return 1;
    }
    else if (key_char == '#') {
        return 0;
    }
    return nurse_call_active;
}

int main(void)
{
    // Getting the ADC
    ADC_init();

    //Initialize lcd and twi for keypad and lcd communication
    twi_init();
    lcd_init();
    lcd_clear_display();
    PCA9555_0_write(REG_CONFIGURATION_0, 0x00);    // Set configuration and control
LCD    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0);    // Set IO1[0:3] as output and
IO1[4:7] as input

    //Initialize the communication with esp
    usart_init(103);

    uint16_t pressure= 0;
    int human_temp;    // Temperature as a whole number (e.g. 25625 = 25.625 C)

    int nurse_call_active = 0; // Flag for Nurse Call

```

```

char receive_buffer[20];          //Buffer to receive answer
char send_connect[20];
char send_URL[20];
char payload_cmd[200];           // Buffer for payload
char lcd_buffer[20];             // Buffer for 1st line of LCD
char status_str[20];             // Buffer for status

while(1)
{
    //----- 1) Send connect command-----
    uart_flush();
    lcd_clear_display();
    usart_print("ESP:connect\r\n");
    usart_receive_string(send_connect,20);

    if (strcmp(send_connect, "\"Success\"") == 0) {
        lcd_print("1.Success");
    }
    else {
        lcd_print("1.Fail");
        _delay_ms(1500);
        continue;
    }

    _delay_ms(1500);

    //----- 2) Send URL command-----
    uart_flush();
    usart_print("ESP:url:\r\n\"http://192.168.1.250:5000/data\"\r\n");
    usart_receive_string(send_URL,20);

    lcd_clear_display();
    if (strcmp(send_URL, "\"Success\"") == 0) {
        lcd_print("2.Success");
    }
    else {
        lcd_print("2.Fail");
        _delay_ms(1500);
        continue;
    }

    _delay_ms(1500);

    //-----3)Read human temp and pressure -----
    human_temp =read_human_temp();                                //Calculate
    pressure= get_CVP_pressure();                                  //Calculate

    //-----4)Check conditions for NURSE CALL-----
    nurse_call_active = check_nurse_call(nurse_call_active);

    // -----5)Status configuration-----
    if (nurse_call_active) {
        strcpy(status_str, "NURSE CALL");
    }
    else if (pressure > 12 || pressure < 4) {
        strcpy(status_str, "CHECK PRESSURE");
    }
    else if (human_temp > 370 || human_temp < 340) {

```

```

        strcpy(status_str, "CHECK TEMP");
    }
    else {
        strcpy(status_str, "OK");
    }
//-----6)Output to LCD-----
    lcd_clear_display();
    //1st line is for temp and pressure
    sprintf(lcd_buffer, "T:%d.%d P:%d", human_temp/10, human_temp%10, (int)
pressure);
    lcd_print(lcd_buffer);

    //2nd line is for status
    lcd_command(0xC0); // send to second line
    lcd_print(status_str);

    _delay_ms(1500); // Delay
//-----7)Create Payload-----
    sprintf(payload_cmd, "ESP:payload:[{\"name\": \"temperature\", \"value\":
\"%d.%d\"},{\"name\": \"pressure\", \"value\": \"%d\"},{\"name\": \"team\", \"value\":
\"30\"},{\"name\": \"status\", \"value\": \"%s\"}]\r\n", human_temp/10, human_temp%10,
(int) pressure, status_str);
//-----8)Send to ESP-----
    uart_flush();
    usart_print(payload_cmd);
//-----9) Check Payload Response-----
    usart_receive_string(receive_buffer, 20);

    lcd_clear_display();
    if (strcmp(receive_buffer, "\"Success\"") == 0) {
        lcd_print("3.Success");
    } else {
        lcd_print("3.Fail");
        _delay_ms(1500);
        continue;
    }
    _delay_ms(1500);
//----- 10) Transmit to Server-----
    uart_flush();
    usart_print("ESP:transmit\r\n");

    usart_receive_string(receive_buffer, 20);

    lcd_clear_display();
    lcd_print("4."); // 4.Answer
    lcd_print(receive_buffer); // It should show us 200 OK
    _delay_ms(1500);
}

}

```

Σχόλια:

Η λογική στο κώδικα εδώ πέρα είναι παρόμοια με του προηγούμενου. Στη πραγματικότητα είναι συνέχεια του προηγούμενου και στα **σχόλια** δείχνουμε αναλυτικά τον συλλογισμό μας και τα βήματα μας. Αφότου λοιπόν εχούμε κάνει οτι

και στην άσκηση 1 μετράμε τη θερμοκρασία του ανθρώπου και τη πίεση με συναρτήσεις που έχουμε φτιάξει και στη συνέχεια ελέγχουμε τις συνθήκες τις οποίες μας ζητάει η άσκηση και τις μεταδίδουμε στον σέρβερ μέχρι να λάβουμε την τελική απάντηση. Σε περίπτωση αποτυχίας κάποιου βήματος όσο αναφορά την επικοινωνία του σέρβερ με εμάς, ο χρήστης ξαναπροσπαθεί να μεταδώσει απ την αρχή (με την χρήση της [continue](#);).