Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο Μικροϋπολογιστών

7ο εξάμηνο, Ακαδημαϊκή περίοδος 2025-2026

7η Εργαστηριακή Άσκηση

Χρήση Αισθητήρα Θερμοκρασίας DS1820 και Σειριακή Επικοινωνία 1-wire

Εργαστηριακή Αναφορά

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

Ομάδα: 30

Ημερομηνία: 2/12/2025

Απάντηση:

Η απάντηση στο **Ζήτημα 7.1** (υλοποίηση της ρουτίνας επικοινωνίας get_temp() και των βασικών λειτουργιών One-Wire) ενσωματώνεται πλήρως στον κώδικα που ακολουθεί. Αυτή η ρουτίνα αποτελεί τη βάση για την επίλυση του **Ζητήματος 7.2**, το οποίο αφορά την επεξεργασία της λαμβανόμενης τιμής της θερμοκρασίας και την απεικόνισή της στην οθόνη LCD σε δεκαδική μορφή °C (3 δεκαδικά ψηφία), καθώς και την εμφάνιση μηνύματος σφάλματος σε περίπτωση αποτυχίας επικοινωνίας.

Ο παρακάτω κώδικας **C** περιλαμβάνει τόσο τις στοιχειώδεις ρουτίνες One-Wire όσο και την κύρια λογική για τη μέτρηση και την εμφάνιση της θερμοκρασίας.

```
#define F_CPU 16000000UL          // Set CPU frequency

#include <avr/io.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, 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;

	// 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 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);
}

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);
	}
}

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
terminates

	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
times left
	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 3 decimal digit of precision:
	// Multiply by 1000 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 * 1000) / 16
	// Use long to avoid overflow during multiplication before division
	return ((long)signed_raw * 1000) / 16;
}

int main()
{
	twi_init();
	lcd_init();
	char no_device[] = "No Device";
	uint16_t temp;
	int temp_decoded;		// Temperature in tenths (e.g. 25625 = 25.625 C)
	int integer_part, decimal_part;

	while(1)
	{
		temp = get_temp();

		if (temp == 0x8000)  // Connection error
		{
			lcd_clear_display();		// Clear Display
			for (uint8_t i = 0; no_device[i] != '\0'; i++)
			{
				lcd_data(no_device[i]);		// Display “NO Device”
			}
			while ((temp = get_temp()) == 0x8000);	// Wait for device connection
		}
		else
		{
			lcd_clear_display();		// Clear Display

			// Convert data (DS18B20 format)
			temp_decoded = get_temp_decoded(temp);

			// Display sign
			if (temp_decoded < 0) {
				lcd_data('-');
				temp_decoded = -temp_decoded;// Positive for division and display
				} else {
				lcd_data('+');
			}

			integer_part = temp_decoded / 1000;
```

```
                decimal_part = temp_decoded % 1000;

                // Print hundreds (if any)
                if (integer_part >= 100) lcd_data('0' + (integer_part / 100));

                // Print tens (if any)
                if (integer_part >= 10) lcd_data('0' + ((integer_part / 10) % 10));

                // Print units
                lcd_data('0' + (integer_part % 10));

                lcd_data('.');

                // Print decimal
                lcd_data('0' + decimal_part/100);
                lcd_data('0' + ((decimal_part / 10) % 10));
                lcd_data('0' + (decimal_part % 10));


                lcd_data(' ');
                lcd_data(223); // ASCII for degree
                lcd_data('C');
            }
            // 1 sec delay so we can read the temperature value before renewal
            _delay_ms(1000);
        }
}
```

Σχόλια:

Στον παραπάνω κώδικα χρησιμοποιήσαμε απλή λογική. Προσπαθήσαμε για αρχή να μεταφράσουμε τους κώδικες από Assembly σε C και μετέπειτα μέσα στην main καλούμε την συνάρτηση για τη μέτρηση θερμοκρασίας. Από εκεί και πέρα μετατρέπουμε την πληροφορία που πήραμε απ τις συναρτήσεις για τη θερμοκρασία σε πραγματικούς αριθμούς και μεταφέρουμε την πληροφορία αυτή μέσω του port-expander στην έξοδο της οθόνης LCD. Για να μετατρέψουμε τις πληροφορίες σε αριθμούς για την LCD οθόνη ακολουθούμε διάφορες μαθηματικές πράξεις όπως φαίνονται στη main. Από εδώ και πέρα περιγράφουμε τον υπόλοιπο κώδικα στα //σχόλια.