



Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο Μικροϋπολογιστών

7^ο εξάμηνο, Ακαδημαϊκή περίοδος 2025-2026

6^η Εργαστηριακή Άσκηση

Χρήση πληκτρολογίου 4×4 σε θύρα επέκτασης στον AVR

Εργαστηριακή Αναφορά

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

Ομάδα: 30

Ημερομηνία: 21/11/2025

Άσκηση 1:

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

```
#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

//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)
{
    TWCRC0 = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
    while(!(TWCRC0 & (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 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
    TWDRO = 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
        TWDRO = 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;
}

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
variable
        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 = 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
        above if (keypad & (1 << i)) return key_table[i];
    }
    return 0;
}

int main(void)
{
    twi_init();

    DDRB = 0xFF; // Initialize PORTB as output
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0); // Set IO1[0:3] as output and
    IO1[4:7] as input

    uint8_t input;

    while(1)
    {
        scan_keypad_rising_edge(); // Scan
        keypad input = keypad_to_ascii(pressed_keys); // Turn input to ASCII

        switch (input) // Output to PORTB
        {
            case '4': PORTB = 0x02; break; // light PB1
            case '2': PORTB = 0x04; break; // light PB2
            case '3': PORTB = 0x08; break; // light PB3
            case 'B': PORTB = 0x10; break; // light PB4
            default: PORTB = 0x00; break; // turn of all leds
        }
    }
}

```

Σχόλια/Περιγραφή του κώδικα:

Εδώ πέρα ο κώδικας είναι πολύ απλός. Μέσα στη main μας χρησιμοποιούμε την συνάρτηση `scan_keypad_rising_edge` για να διαβάσουμε την τιμή του πλήκτρου που έχει πατηθεί και το μετατρέπουμε ανάλογα με τη θέση του στον κατάλληλο χαρακτήρα ascii με τη χρήση της συνάρτησης `keypad_to_ascii`. Μετά αποθηκεύουμε την τιμή που βρήκαμε στη μεταβλητή `input` και στη συνέχεια ελέγχουμε αν αυτή η μεταβλητή είναι αυτή που ζητάει η άσκηση ώστε να ανοίξουν τα ανάλογα LEDs. Θα πρέπει όμως να ανάλυσουμε περαιτέρω το διαβάσμα του χαρακτήρα με τη συνάρτηση `keypad_to_ascii` και τη `scan_keypad_rising_edge`. Η συνάρτηση `scan_keypad_rising_edge` παίρνει και ελέγχει ποιο κουμπί έχει πατηθεί και στις 4 γραμμές με τη χρήση της συνάρτησης `scan_keypad` και χωρίζει τη μεταβλητή

new_pressed_key σε 4 τετράδες, μια για κάθε γραμμή. Επείτα καλεί την keypad_to_ascii και κάνει τη κατάλληλη μετατροπή.

Άσκηση 2:

Παρακάτω φαίνεται ο κώδικας C που φτιάξαμε:

```
#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

//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 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
    TWDRO = 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;
}

void write_2_nibbles(uint8_t lcd_data) {
    uint8_t temp;

    // Send the high nibble
    temp = (PCA9555_0_read(REG_OUTPUT_0) & 0x0F) | (lcd_data & 0xF0);    // Keep lower
4 bits of PIND and set high nibble of lcd_data
    PCA9555_0_write(REG_OUTPUT_0, temp);
    // Output the high nibble to PORTD
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));    //
Enable pulse high
    _delay_us(1);
    // Small delay to let the signal settle
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));    //
Enable pulse low

    // Send the low nibble
    lcd_data <<= 4;
    // Move low nibble to high nibble position
    temp = (PCA9555_0_read(REG_OUTPUT_0) & 0x0F) | (lcd_data & 0xF0);    // Keep lower
4 bits of PIND and set high nibble of new lcd_data
    PCA9555_0_write(REG_OUTPUT_0, temp);
    // Output the low nibble to PORTD
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));    //
Enable pulse high
    _delay_us(1);
    // Small delay to let the signal settle
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));    //
Enable pulse low
}

void lcd_data(uint8_t data)
{
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | 0x04);    //
LCD_RS = 1, (PD2 = 1) -> For Data
    write_2_nibbles(data);    // Send data
    _delay_ms(5);
    return;
}

void lcd_command(uint8_t data)
{
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & 0xFB);    //
LCD_RS = 0, (PD2 = 0) -> For Instruction
    write_2_nibbles(data);    // Send data
}

```

```

        _delay_ms(5);
        return;
    }

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
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3)); //
Enable pulse
    _delay_us(1);
    PCA9555_0_write(REG_OUTPUT_0, PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3)); //
Clear enable
    _delay_us(30); // Wait 250 µs

    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);
}

```

```

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
variable
        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 = 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
above
        if (keypad & (1 << i)) return key_table[i];
    }
    return 0;
}

```

```

int main(void)
{
    twi_init();

    DDRB = 0xFF; // Initialize PORTB as output
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0); // Set IO1[0:3] as output and
    IO1[4:7] as input
    PCA9555_0_write(REG_CONFIGURATION_0, 0x00); // Set EXT_PORT0 as output
    lcd_init();
    uint16_t new_key;

    while(1)
    {
        new_key = scan_keypad_rising_edge(); // Scan keypad
        if (new_key != 0)
        {
            uint8_t input = keypad_to_ascii(new_key); // Turn input to
ASCII
            lcd_clear_display();
            lcd_data(input);
        }
    }
}

```

Σχόλια:

Ο κώδικας εδώ λειτουργεί με τον ίδιο τρόπο που λειτουργεί και πάνω με τη μόνη διαφορά να είναι ο χρησιμοποιούμε και την LCD οθόνη να δείξουμε ποιο κουμπί έχει πατηθεί.

Άσκηση 3:

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

```

#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

//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 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);
    // 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;
}

void write_2_nibbles(uint8_t lcd_data) {
    uint8_t temp;

    // Send the high nibble
    temp = (PCA9555_0_read(REG_OUTPUT_0) & 0x0F) | (lcd_data & 0xF0);    // Keep lower
4 bits of PIND and set high nibble of lcd_data
    PCA9555_0_write(REG_OUTPUT_0 , temp);
    // Output the high nibble to PORTD
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));    //
Enable pulse high
    _delay_us(1);
    // Small delay to let the signal settle

```

```

        PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));    //
Enable pulse low

        // Send the low nibble
        lcd_data <=< 4;
        // Move low nibble to high nibble position
        temp = (PCA9555_0_read(REG_OUTPUT_0) & 0x0F) | (lcd_data & 0xF0);    // Keep lower
4 bits of PIND and set high nibble of new lcd_data
        PCA9555_0_write(REG_OUTPUT_0 , temp);
        // Output the low nibble to PORTD
        PCA9555_0_write(REG_OUTPUT_0 , PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));    //
Enable pulse high
        _delay_us(1);
        // Small delay to let the signal settle
        PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));    //
Enable pulse low
    }

void lcd_data(uint8_t data)
{
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) | 0x04);    //
LCD_RS = 1, (PD2 = 1) -> For Data
    write_2_nibbles(data);    // Send data
    _delay_ms(5);
    return;
}

void lcd_command(uint8_t data)
{
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) & 0xFB);    //
LCD_RS = 0, (PD2 = 0) -> For Instruction
    write_2_nibbles(data);    // Send data
    _delay_ms(5);
    return;
}

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
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) | (1 << PD3));    //
Enable pulse
    _delay_us(1);
    PCA9555_0_write(REG_OUTPUT_0,PCA9555_0_read(REG_OUTPUT_0) & ~(1 << PD3));    //
Clear enable
    _delay_us(30);    // Wait 250 µs

    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);
}

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 variable
        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)
    // Check new pressed keys
    uint16_t new_pressed_keys = pressed_keys_tempo & ~pressed_keys;
    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 above
        if (keypad & (1 << i)) return key_table[i];
    }
    return 0;
}

void light_leds_3s()
{
    PORTB = 0x3F;           // light PB0-PB5
    _delay_ms(3000);
    PORTB = 0x00;           // turn off PB0-PB5
    return;
}

void blink_leds_6s()
{
    for (int i=0; i<6; i++)
    {
        PORTB = 0x3F;           // light PB0-PB5
        _delay_ms(500);
        PORTB = 0x00;           // turn off PB0-PB5
        _delay_ms(500);
    }
    return;
}

int main(void)
{
    twi_init();

    DDRB = 0xFF;           // Initialize PORTB as output
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0); // Set IO1[0:3] as output and
    IO1[4:7] as input
    PCA9555_0_write(REG_CONFIGURATION_0, 0x00); // Set EXT_PORT0 as output
    lcd_init();
}

```

```

uint16_t input;
uint8_t new_key_1, new_key_2;

while(1)
{
    while ((input = scan_keypad_rising_edge()) == 0);    // Scan keypad for
1st number
    uint8_t new_key_1 = keypad_to_ascii(input);          // Turn input to
ASCII
    lcd_data(new_key_1);

    while ((input = scan_keypad_rising_edge()) == 0);    // Scan keypad for
2nd number
    uint8_t new_key_2 = keypad_to_ascii(input);          // Turn input
to ASCII
    lcd_data(new_key_2);

    // Check if input is number 30
    if ((new_key_1 == '3') && (new_key_2 == '0')) light_leds_3s(); // if YES
light leds for 3s
    else blink_leds_6s();                                // if NO blink leds for 6s

    _delay_ms(5000);                                     // block input for another 5s
    lcd_clear_display();
}
}

```

Σχόλια:

Η λογική στο κώδικα εδώ πέρα είναι παρόμοια με των προηγούμενων. Η διαφορά είναι ότι εδώ περιμένουμε να κάνει trigger η σύνθηκη για τα LEDs μόνο αν πατηθούν 2 πλήκτρα τα οποία δίνουν τον συνδυασμό που θέλουμε. Ο συνδυασμός που θέλουμε είναι να πατηθεί το νούμερο 30. Μολις γίνει αυτό αναβούν όλα τα LEDs PB0-PB5 για 3 δευτερόλεπτα ενώ σε περίπτωση που έχουμε άλλο συνδυασμό για αριθμό, αναβοσβήνουν τα ίδια LEDs για 6 δευτερόλεπτα.

