Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο Μικροϋπολογιστών

7ο εξάμηνο, Ακαδημαϊκή περίοδος 2025-2026

5η Εργαστηριακή Άσκηση

Χρήση εξωτερικών Θυρών Επέκτασης στον AVR

Εργαστηριακή Αναφορά

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

Ομάδα: 30

Ημερομηνία: 15/11/2025

Άσκηση 1:

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

```c
/*
 * main.c
 *
 * Created: 11/13/2025 11:52:24 AM
 *  Author: anast
 */

#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

// Fscl = Fcpu / (16 + 2 * TWBR0_VALUE * PRESCALER_VALUE)
#define TWBR0_VALUE (((F_CPU / SCL_CLOCK) - 16) / 2)

// ===== INPUT DEFINITIONS =====
// Inputs A, B, C, D connected to PORTB pins 0-3 on ntuAboard_G1
#define A   PB0
#define B   PB1
#define C   PB2
#define D   PB3

// 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)
```

```c
// 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
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));
}

// Write to PCA9555
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();
}

// Read from PCA9555
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;
}

int main(void) {
	DDRB &= ~((1 << A) | (1 << B) | (1 << C) | (1 << D)); // Set PB0-PB3 as inputs
	uint8_t a, b, c, d;
	uint8_t F0;
	uint8_t F1;
	twi_init();
	PCA9555_0_write(REG_CONFIGURATION_0, 0x00); // Set EXT_PORT0 as output

	while (1) {
		a = (PINB >> A) & 1;
		b = (PINB >> B) & 1;
		c = (PINB >> C) & 1;
		d = (PINB >> D) & 1;

		F0 = ~((a & ~b) | (c & b & d)) & 0x01;    //F0 = (AB'+CBD)'
		F1 = (a | c) & (b & d);                            //F1 = (A+C)(BD)

		uint8_t output = (F1 << 1) | (F0 << 0);  // bit1 = F1, bit0 = F0

		PCA9555_0_write(REG_OUTPUT_0, output);
		_delay_ms(100);
	}

}
```

<u>Σχόλια/Περιγραφή του κώδικα:</u>

Η λειτουργία του παραπάνω κώδικα είναι πολύ απλή. Προσπαθούμε να υλοποιήσουμε τις λογικές συναρτήσεις F0 και F1 με τον εξής τρόπο: Υπολογίζουμε τις τιμές τους απ τον ATMega328PB και στη συνέχεια μέσω του PORT EXPANDER ανάβουμε τα LED που θέλουμε. ‘Αρα ο ATMega328PB υπολογίζει και ο PORT EXPANDER τα προωθεί στην έξοδο. Για να το κάνει αυτό χρησιμοποιεί διάφορα πρωτόκολλα όπως το πρωτόκολλο αποφυγής σύγκρουσης δεδομένων στο δίαυλο. Χρησιμοποιεί 2 buses πάνω στο οποίο συνδέονται μέχρι και 127 συσκευές , το bus του ρολογιού και το bus των δεδομένων. Μετά φτιάχνουμε τις συναρτήσεις που αφορούν τον συγχρονισμό των συσκευών με την MASTER συσκευή που “διατάζει” να είναι ο ATMega328PB και η SLAVE συσκευή να είναι το PORT EXPANDER.

Άσκηση 2:

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

```c
/*
 * main.c
 *
 * Created: 11/13/2025 12:18:26 PM
 *  Author: anast
 */

#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

// Fscl = Fcpu / (16 + 2 * TWBR0_VALUE * PRESCALER_VALUE)
#define TWBR0_VALUE (((F_CPU / SCL_CLOCK) - 16) / 2)

// ===== INPUT DEFINITIONS =====
// Inputs A, B, C, D connected to PORTB pins 0-3 on ntuAboard_G1
#define A   PB0
#define B   PB1
#define C   PB2
#define D   PB3

// 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;
```

```c
			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
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));
}

// Write to PCA9555
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();
}

// Read from PCA9555
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;
}

int main(void) {
	DDRB &= ~((1 << A) | (1 << B) | (1 << C) | (1 << D)); // Set PB0-PB3 as inputs
	uint8_t a, b, c, d;
	uint8_t output;
	uint8_t keypad;
	twi_init();
	PCA9555_0_write(REG_CONFIGURATION_0, 0x00);			// Set EXT_PORT0 as
output
	PCA9555_0_write(REG_CONFIGURATION_1, 0b11110000);	// Set EXT_PORT1 IO1_3 as
output and IO1_4 - IO0_7 as input

	while (1) {
		PCA9555_0_write(REG_OUTPUT_1, ~(0x08));		// IO1_3 = 0 so we can
check which key is pressed
		keypad = PCA9555_0_read(REG_INPUT_1);
		output = ~((keypad >> 4) & 0x0F);			// Shift, mask and invert
input from keypad so we can output

		PCA9555_0_write(REG_OUTPUT_0, output);
		_delay_ms(100);
	}

}
```

Σχόλια:

Εδώ πέρα ο κώδικας είναι ελάχιστα διαφορετικός απ ότι πριν. Χρησιμοποιούμε όλες τις συναρτήσεις επικοινωνίας και συγχρονισμού που είχαμε παραπάνω απλά εδώ πέρα είναι σημαντικό να καταλάβουμε πως δουλεύει το keypad. Στον κώδικα μας ενεργοποιούμε μόνο την πρώτη γραμμή (βάζοντας 0) και θέτουμε όλες τις άλλες σε 1. Με αυτό το τρόπο μετά μπορούμε να σιγουρευτούμε ότι όταν θα διαβάσει τις στήλες, σε περίπτωση που έχει πατηθεί κάποιο πλήκτρο της ίδιας στήλης, αλλά γραμμής που έχει είσοδο 1, δεν θα γίνει 0 η στήλη αυτή. Η στήλη θα γίνει 0 μόνο αν έχει πατηθεί κουμπί το οποίο ανήκει στη γραμμή που έχει γίνει και αυτή 0. Εδώ πέρα ανάλογα το πλήκτρο που έχει πατηθεί, θα έχουμε και ανάλογη έξοδο στα LEDS

Άσκηση 3:

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

```
/*
 * main.c
 *
 * Created: 11/13/2025 12:29:26 PM
```

```c
 *  Author: anast
 */

#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdint.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)
// ===== INPUT DEFINITIONS =====
// Inputs A, B, C, D connected to PORTB pins 0-3 on ntuAboard_G1
#define A   PB0
#define B   PB1
#define C   PB2
#define D   PB3

// 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;
```

```c
                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
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));
}

// Write to PCA9555
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();
}

// Read from PCA9555
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);
}



int main(void) {
	DDRB &= ~((1 << A) | (1 << B) | (1 << C) | (1 << D)); // Set PB0-PB3 as inputs
	uint8_t a, b, c, d;
	uint8_t output;
	twi_init();
	PCA9555_0_write(REG_CONFIGURATION_0, 0x00);			// Set EXT_PORT0 as
output
	PCA9555_0_write(REG_CONFIGURATION_1, 0b11110000);	// Set EXT_PORT1 IO1_3 as
output and IO1_4 - IO0_7 as input
	lcd_init();
	char nameA1[] = "Anastasis";
	char nameA2[] = "Papadatos";
	char nameB1[] = "Sergio";
	char nameB2[] = "Guri";

	while (1)
	{

		lcd_clear_display();

		lcd_command(0x80);
		for (uint8_t i = 0; nameA1[i] != '\0'; i++) {
			lcd_data(nameA1[i]);
		}

		lcd_command(0xC0);
		for (uint8_t i = 0; nameA2[i] != '\0'; i++) {
			lcd_data(nameA2[i]);
		}

		_delay_ms(2000);


		lcd_clear_display();

		lcd_command(0x80);
		for (uint8_t i = 0; nameB1[i] != '\0'; i++) {
			lcd_data(nameB1[i]);
		}

		lcd_command(0xC0);
		for (uint8_t i = 0; nameB2[i] != '\0'; i++) {
			lcd_data(nameB2[i]);
		}

		_delay_ms(2000);
	}

}
```

Σχόλια:

Εδώ πέρα ο κώδικας μοιάζει αρκετά με αυτόν από τη προηγούμενη εργαστηριακή άσκηση. Ουσιαστικά κάθε 2 δευτερόλεπτα, θα εναλλάσσεται η οθόνη από την μια κατάσταση στην άλλη, έχοντας ως έξοδο τα ονόματα μας και επίσης θα έχουμε έξοδο στα LED.