Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

6η Σειρά Ασκήσεων

Μάθημα: Εργαστήριο Μικροϋπολογιστών

Εξάμηνο: 7ο

Ονοματεπώνυμα: Αλεξοπούλου Γεωργία, Γκενάκου Ζωή

Ζήτημα 6.1

Να υλοποιηθεί κώδικας για το μικροελεγκτή ATmega328PB, σε γλώσσα C, ο οποίος να διαβάζει το πληκτρολόγιο της ntuAboard_G1, και να αναντιστοιχεί τέσσερα από τα πλήκτρα του πληκτρολογίου σε τέσσερα led σύμφωνα με τον παρακάτω πίνακα:

“1”	“5”	“9”	“D”
Led στο PB0	Led στο PB1	Led στο PB2	Led στο PB3

Κάθε φορά που θα πιέζεται κάποιο από τα παραπάνω τέσσερα πλήκτρα να ανάβει το αντίστοιχο led. Το led θα παραμένει αναμμένο όσο διάστημα το πλήκτρο είναι πατημένο.

Ο κώδικας θα εμπεριέχει τις συναρτήσεις: scan_row, scan_keypad, scan_keypad_rising_edge και keypad_to_ascii, όπως ορίζει η εκφώνηση, ώστε να μπορούμε και να διαβάζουμε την είσοδο του πληκτρολογίου.

```c
#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

// 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;
}

// Define an array to store memory and current key states
unsigned char memory[2];
unsigned char currentKeyState[2];
unsigned char currentKey;

// Function to scan a specific row of the keypad
unsigned char scan_row(int rowIndex) {
    // Create a mask to select the desired row
    unsigned char mask = 0b00000001;
    rowIndex = rowIndex - 1;
    mask = (mask << rowIndex);
    mask = ~mask;

    // Set the selected row to logic 0 (inverted logic) using I2C
communication
    PCA9555_0_write(REG_OUTPUT_1, mask);

    // Read the input state to check if a column is tapped on the selected
row
    uint8_t inputState = PCA9555_0_read(REG_INPUT_1);

    // Keep 4 MSB, which represent the column state, and invert the logic
    return ~(inputState | 0x0F);
}

// Function to swap the low-order with high-order bits
unsigned char swapNibbles(unsigned char x) {
    return ((x & 0x0F) << 4 | (x & 0xF0) >> 4);
}

// Function to scan the entire keypad and update the currentKeyState array
void scan_keypad() {
    unsigned char keyState;

    // Scan each row of the keypad
    keyState = scan_row(1);
```

```
    currentKeyState[0] = swapNibbles(keyState);

    keyState = scan_row(2);
    currentKeyState[0] += keyState;

    keyState = scan_row(3);
    currentKeyState[1] = swapNibbles(keyState);

    keyState = scan_row(4);
    currentKeyState[1] += keyState;
}

// Function to check if there is a falling edge in the keypad state
int scan_keypad_falling_edge() {
    scan_keypad();
    // Check if both key state arrays are 0, indicating no keys are pressed
    if ((currentKeyState[0] == 0b00000000) && (currentKeyState[1] ==
0b00000000)) return 0;
    else return 1;
}

// Function to check if there is a rising edge in the keypad state
int scan_keypad_rising_edge() {
    scan_keypad();

    unsigned char tempKeyState[2];
    tempKeyState[0] = currentKeyState[0];
    tempKeyState[1] = currentKeyState[1];

    // Wait for a short delay
    _delay_ms(10);

    scan_keypad();

    // Compare the states before and after the delay
    currentKeyState[0] &= tempKeyState[0];
    currentKeyState[1] &= tempKeyState[1];

    tempKeyState[0] = memory[0];
    tempKeyState[1] = memory[1];

    memory[0] = currentKeyState[0];
    memory[1] = currentKeyState[1];
```

```
    currentKeyState[0] &= ~tempKeyState[0];
    currentKeyState[1] &= ~tempKeyState[1];

    // Return true if any key is pressed
    return (currentKeyState[0] || currentKeyState[1]);
}

// Function to convert the current key state to ASCII representation
unsigned char keypad_to_ascii() {
    // Check each bit of the key states and return the corresponding ASCII
character
    if (currentKeyState[0] & 0x01)
        return '*';

    if (currentKeyState[0] & 0x02)
        return '0';

    if (currentKeyState[0] & 0x04)
        return '#';

    if (currentKeyState[0] & 0x08)
        return 'D';

    if (currentKeyState[0] & 0x10)
        return '7';

    if (currentKeyState[0] & 0x20)
        return '8';

    if (currentKeyState[0] & 0x40)
        return '9';

    if (currentKeyState[0] & 0x80)
        return 'C';

    if (currentKeyState[1] & 0x01)
        return '4';

    if (currentKeyState[1] & 0x02)
        return '5';

    if (currentKeyState[1] & 0x04)
```

```
        return '6';

    if (currentKeyState[1] & 0x08)
        return 'B';

    if (currentKeyState[1] & 0x10)
        return '1';

    if (currentKeyState[1] & 0x20)
        return '2';

    if (currentKeyState[1] & 0x40)
        return '3';

    if (currentKeyState[1] & 0x80)
        return 'A';

    return 0; // error
}

// Function to control LEDs based on the pressed key
void controlLEDs(char key) {
    // Check which key is pressed and light up the corresponding LED
    switch (key) {
    case '1':
        PORTB = 0b00000001;
        break;
    case '5':
        PORTB = 0b00000010;
        break;
    case '9':
        PORTB = 0b00000100;
        break;
    case 'D':
        PORTB = 0b00001000;
        break;
    default:
        break;
    }
}

// Main function
int main() {
```

```
    // Initialize I2C communication and set up LED configuration
    twi_init();
    DDRB = 0xFF;
    PCA9555_0_write(REG_CONFIGURATION_0, 0x00);
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0);

    memory[0] = 0;
    memory[1] = 0;

    // Main loop
    while (1) {
        // Check for a falling edge in the keypad state
        while (scan_keypad_falling_edge() == 1) {
            // Get the ASCII representation of the pressed key
            char pressedKey = keypad_to_ascii();
            // Control LEDs based on the pressed key
            controlLEDs(pressedKey);
        }

        // Turn off all LEDs when no key is pressed
        PORTB = 0b00000000;
    }

    return 0;
}
```

Το πρόγραμμα χρησιμοποιεί τη διεπαφή δύο καλωδίων (TWI) για επικοινωνία I2C. Καθορίζει τους καταχωρητές PCA9555 και διάφορους κωδικούς κατάστασης TWI. Η κύρια λειτουργία περιλαμβάνει λειτουργίες για την προετοιμασία του ρολογιού TWI, την εγγραφή και την ανάγνωση από τους καταχωρητές PCA9555 και τη σάρωση του πληκτρολογίου για πατήματα πλήκτρων. Το πρόγραμμα παρακολουθεί συνεχώς την κατάσταση του πληκτρολογίου, ανιχνεύει τα falling edges για να αναγνωρίσει τα πατήματα των πλήκτρων και στη συνέχεια ανάβει τα LED που σχετίζονται με τα πατημένα πλήκτρα. Τα LED ελέγχονται μέσω των εξόδων PCA9555. Ο κύριος βρόχος ελέγχει συνεχώς για τη δραστηριότητα του πληκτρολογίου, ενημερώνει τις καταστάσεις LED (μέσω της `controlLEDs()`) ανάλογα και σβήνει όλα τα LED όταν δεν πατιέται κανένα πλήκτρο.

Συγκεκριμένα για τις συναρτήσεις που κληθήκαμε να υλοποιήσουμε:

1. Συνάρτηση **‘scan_row’:**

    - **Σκοπός:** Αυτή η συνάρτηση έχει σχεδιαστεί για τη σάρωση του πληκτρολογίου 4x4 που είναι συνδεδεμένο μέσω μιας επέκτασης I2C I/O (PCA9555).

    - **Παράμετροι**: Το "rowIndex" καθορίζει τη σειρά (1 έως 4) που πρέπει να σαρώσει η συνάρτηση.

    - **Λειτουργία:**
        - Δημιουργεί μια μάσκα bit με ένα «1» στη θέση που αντιστοιχεί στην επιθυμητή σειρά και αντιστρέφει τα bit.

        - Χρησιμοποιεί το PCA9555 για να ορίσει την επιλεγμένη σειρά στη λογική 0 (ανεστραμμένη λογική) γράφοντας στον καταχωρητή ‘REG_OUTPUT_1`.

        - Διαβάζει την κατάσταση εισόδου από το PCA9555 για την αντίστοιχη σειρά (`REG_INPUT_1`).

        - Διατηρεί τα 4 πιο σημαντικά bit (MSB) της εισόδου, που αντιπροσωπεύουν την κατάσταση της στήλης, και αντιστρέφει τη λογική πριν επιστρέψει το αποτέλεσμα.

2. Συνάρτηση **`swapNibbles`:**

    - **Σκοπός:** Αυτή η συνάρτηση ανταλλάσσει τα bit low-orderbit με high-order bit σε ένα δεδομένο byte.

    - **Παράμετροι**: Το "x" είναι το byte του οποίου τα nibbles (4 bit) πρέπει να αντικατασταθούν.

    - **Λειτουργία:**
        - Παίρνει το byte εισόδου, μετατοπίζει το low-order nibble προς τα αριστερά κατά 4 θέσεις και μετατοπίζει το high-order nibble προς τα δεξιά κατά 4 θέσεις.

        - Στη συνέχεια συνδυάζει αυτά τα δύο nibbles χρησιμοποιώντας bitwise OR, εναλλάσσοντας αποτελεσματικά τις θέσεις τους και επιστρέφει το αποτέλεσμα.

3. Συνάρτηση **`scan_keypad`:**

- **Σκοπός:** Αυτή η συνάρτηση σαρώνει ολόκληρο το πληκτρολόγιο 4x4 χρησιμοποιώντας τη συνάρτηση "scan_row" και ενημερώνει τον πίνακα "currentKeyState".

- **Λειτουργία:**
    - - Καλεί την «scan_row» για κάθε σειρά του πληκτρολογίου και ενημερώνει τον πίνακα «currentKeyState» ανάλογα.
    - - Το αποτέλεσμα αποθηκεύεται σε δύο byte ("currentKeyState[0]" και "currentKeyState[1]", το καθένα αντιπροσωπεύει την κατάσταση μιας σειράς στο πληκτρολόγιο.

4. Συνάρτηση **`scan_keypad_rising_edge`**:

- **Σκοπός:** Αυτή η συνάρτηση ελέγχει εάν υπάρχει rising edge στην κατάσταση του πληκτρολογίου συγκρίνοντας την τρέχουσα κατάσταση με την προηγούμενη κατάσταση μετά από μια μικρή καθυστέρηση.

- **Λειτουργία:**
    - Καλεί «scan_keypad» για να πάρει την τρέχουσα κατάσταση.
    - Περιμένει μια μικρή καθυστέρηση χρησιμοποιώντας το `_delay_ms(10)`.
    - Καλεί ξανά το «scan_keypad» για να λάβει την ενημερωμένη κατάσταση.
    - Συγκρίνει τις καταστάσεις πριν και μετά την καθυστέρηση, αναζητώντας αλλαγές.
    - Ενημερώνει τον πίνακα "μνήμης" με την προηγούμενη κατάσταση.
    - Επιστρέφει true εάν πατηθεί οποιοδήποτε πλήκτρο (υποδηλώνει μια ανερχόμενη άκρη).

5. Συνάρτηση **`keypad_to_ascii`**:

- **Σκοπός:** Αυτή η συνάρτηση μετατρέπει την τρέχουσα κατάσταση του πλήκτρου (που αντιπροσωπεύει πατημένα πλήκτρα) σε χαρακτήρες ASCII.

- **Λειτουργία:**
    - Ελέγχει κάθε bit των καταστάσεων του κλειδιού στο «currentKeyState» και επιστρέφει τον αντίστοιχο χαρακτήρα ASCII για το πατημένο πλήκτρο.
    - Η λειτουργία χρησιμοποιεί συγκεκριμένες αντιστοιχίσεις για κάθε πλήκτρο, όπως αριθμούς, γράμματα και ειδικούς χαρακτήρες.
    - Εάν δεν πατηθεί κανένα πλήκτρο, επιστρέφει το 0 για να υποδείξει σφάλμα ή κανένα έγκυρο πλήκτρο.

Ζήτημα 6.2

Να υλοποιηθεί κώδικας για το μικροελεγκτή ATmega328PB, σε γλώσσα C, ο οποίος θα κάνει χρήση των συναρτήσεων του ζητήματος 6.1 και θα απεικονίζει στην οθόνη LCD 2x16 το χαρακτήρα που αντιστοιχεί στο πλήκτρο που πατήθηκε τελευταίο. Θεωρείστε δεδομένο ότι κάθε φορά μπορεί να πατηθεί μόνο ένα πλήκτρο.

```
#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

// 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;
}

// Define an array to store memory and current key states
unsigned char memory[2];
unsigned char currentKeyState[2];
unsigned char currentKey;

// Function to scan a specific row of the keypad
unsigned char scan_row(int rowIndex) {
    // Create a mask to select the desired row
    unsigned char mask = 0b00000001;
    rowIndex = rowIndex - 1;
    mask = (mask << rowIndex);
    mask = ~mask;

    // Set the selected row to logic 0 (inverted logic) using I2C
communication
    PCA9555_0_write(REG_OUTPUT_1, mask);

    // Read the input state to check if a column is tapped on the selected
row
    uint8_t inputState = PCA9555_0_read(REG_INPUT_1);

    // Keep 4 MSB, which represent the column state, and invert the logic
    return ~(inputState | 0x0F);
}

// Function to swap the low-order with high-order bits
unsigned char swapNibbles(unsigned char x) {
    return ((x & 0x0F) << 4 | (x & 0xF0) >> 4);
```

```
}

// Function to scan the entire keypad and update the currentKeyState array
void scan_keypad() {
    unsigned char keyState;

    // Scan each row of the keypad
    keyState = scan_row(1);
    currentKeyState[0] = swapNibbles(keyState);

    keyState = scan_row(2);
    currentKeyState[0] += keyState;

    keyState = scan_row(3);
    currentKeyState[1] = swapNibbles(keyState);

    keyState = scan_row(4);
    currentKeyState[1] += keyState;
}


// Function to check if there is a rising edge in the keypad state
int scan_keypad_rising_edge() {
    scan_keypad();

    unsigned char tempKeyState[2];
    tempKeyState[0] = currentKeyState[0];
    tempKeyState[1] = currentKeyState[1];

    // Wait for a short delay
    _delay_ms(10);

    scan_keypad();

    // Compare the states before and after the delay
    currentKeyState[0] &= tempKeyState[0];
    currentKeyState[1] &= tempKeyState[1];

    tempKeyState[0] = memory[0];
    tempKeyState[1] = memory[1];

    memory[0] = currentKeyState[0];
    memory[1] = currentKeyState[1];
```

```
    currentKeyState[0] &= ~tempKeyState[0];
    currentKeyState[1] &= ~tempKeyState[1];

    // Return true if any key is pressed
    return (currentKeyState[0] || currentKeyState[1]);
}

// Function to convert the current key state to ASCII representation
unsigned char keypad_to_ascii() {
    // Check each bit of the key states and return the corresponding ASCII
character
    if (currentKeyState[0] & 0x01)
        return '*';

    if (currentKeyState[0] & 0x02)
        return '0';

    if (currentKeyState[0] & 0x04)
        return '#';

    if (currentKeyState[0] & 0x08)
        return 'D';

    if (currentKeyState[0] & 0x10)
        return '7';

    if (currentKeyState[0] & 0x20)
        return '8';

    if (currentKeyState[0] & 0x40)
        return '9';

    if (currentKeyState[0] & 0x80)
        return 'C';

    if (currentKeyState[1] & 0x01)
        return '4';

    if (currentKeyState[1] & 0x02)
        return '5';

    if (currentKeyState[1] & 0x04)
```

```
        return '6';

    if (currentKeyState[1] & 0x08)
        return 'B';

    if (currentKeyState[1] & 0x10)
        return '1';

    if (currentKeyState[1] & 0x20)
        return '2';

    if (currentKeyState[1] & 0x40)
        return '3';

    if (currentKeyState[1] & 0x80)
        return 'A';

    return 0; // error
}

// Function to write two nibbles of data to the LCD
void write_2_nibbles(uint8_t input) {
    uint8_t temp = input;
    uint8_t pin = PIND;
    pin &= 0x0F;
    input &= 0xF0;
    input |= pin;
    PORTD = input;
    PORTD |= 0x08;
    PORTD &= 0xF7;

    input = temp;
    input &= 0x0F;
    input = input << 4;
    input |= pin;
    PORTD = input;
    PORTD |= 0x08;
    PORTD &= 0xF7;

    return;
}

// Function to send data to the LCD
```

```
void LCD_data(uint8_t x) {
    PORTD |= 0x04;
    write_2_nibbles(x);
    _delay_us(250);
    return;
}

// Function to send a command to the LCD
void LCD_command(uint8_t x) {
    PORTD &= 0xFB;
    write_2_nibbles(x);
    _delay_us(250);
    return;
}

// Function to clear the LCD display
void LCD_clear_display() {
    uint8_t x = 0x01;
    LCD_command(x);
    _delay_ms(5);
}

// Function to initialize the LCD
void LCD_init(void) {
    _delay_ms(200);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(250);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(250);

    PORTD = 0x20;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(250);

    LCD_command(0x28);
    LCD_command(0x0C);
```

```
    LCD_clear_display();

    LCD_command(0x06);
}

int main ()
{
        // Initialize TWI (Two-Wire Interface) communication
    twi_init();

    // Set all bits of PORTD as output for LCD control
    DDRD = 0xFF;

    // Configure PCA9555_0 (assuming it's some kind of external device) -
specific to the application
    PCA9555_0_write(REG_CONFIGURATION_0, 0x00);
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0);

    // Initialize the LCD
    LCD_init();

    // Main loop
    while(1)
    {
        memory[0] = 0; // Clear memory
        memory[1] = 0;

        while (1) {
            if (scan_keypad_rising_edge()) {

            char pressedKey = keypad_to_ascii();
            LCD_init();
            LCD_data(pressedKey);
            break;
            }
        }
    }
}
```

Η κύρια λειτουργία περιλαμβάνει λειτουργίες για τη σάρωση του πληκτρολογίου για rising edges, τη μετατροπή της κατάστασης του πληκτρολογίου σε αναπαράσταση ASCII και τον έλεγχο μιας οθόνης LCD. Το πρόγραμμα αρχικοποιεί την επικοινωνία TWI, ρυθμίζει τα pins ελέγχου της LCD, διαμορφώνει το PCA9555 και προετοιμάζει την οθόνη LCD. Στον κύριο βρόχο, παρακολουθεί συνεχώς το πληκτρολόγιο για ένα rising edge και ενημερώνει την οθόνη LCD με τον αντίστοιχο χαρακτήρα ASCII όταν πατηθεί ένα πλήκτρο και επαναλαμβάνει τη διαδικασία. Ο χαρακτήρας αυτός εμφανίζεται στην LCD οθόνη μέχρι να πατηθεί άλλο κουμπί.

Ζήτημα 6.3

Γράψτε ένα πρόγραμμα «ηλεκτρονικής κλειδαριάς» το οποίο να ανάβει τα 6 leds PB0 έως PB5 για 4 sec, όταν δοθεί από το keypad 4×4 ο διψήφιος αριθμός της ομάδας σας (π.χ. 09). Εάν κάποιος σπουδαστής δεν έχει αριθμό ομάδας να κάνει χρήση του αριθμού 99. Αν δεν έχουν δοθεί σωστά τα δύο ψηφία του αριθμού της ομάδας τότε τα leds PB0 έως PB5 να αναβοσβήνουν(250 mSec ανάμενα, 250 mSec σβηστά) για 5 sec. Ανεξάρτητα από το χρονικό διάστημα για το οποίο θα μείνει πατημένο ένα πλήκτρο, το πρόγραμμά θα πρέπει να θεωρεί ότι πατήθηκε μόνο μια φορά. Μετά το πάτημα δύο αριθμών το πρόγραμμα να μην δέχεται για 5 sec άλλον αριθμό. Το πρόγραμμα να είναι συνεχόμενης λειτουργίας. Δώστε το διάγραμμα ροής και το πρόγραμμα σε γλώσσα C.

```
#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

// 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;
}

// Define an array to store memory and current key states
unsigned char memory[2];
unsigned char currentKeyState[2];
unsigned char currentKey;

// Function to scan a specific row of the keypad
unsigned char scan_row(int rowIndex) {
    // Create a mask to select the desired row
    unsigned char mask = 0b00000001;
    rowIndex = rowIndex - 1;
    mask = (mask << rowIndex);
    mask = ~mask;

    // Set the selected row to logic 0 (inverted logic) using I2C
```

```
communication
    PCA9555_0_write(REG_OUTPUT_1, mask);

    // Read the input state to check if a column is tapped on the selected
row
    uint8_t inputState = PCA9555_0_read(REG_INPUT_1);

    // Keep 4 MSB, which represent the column state, and invert the logic
    return ~(inputState | 0x0F);
}

// Function to swap the low-order with high-order bits
unsigned char swapNibbles(unsigned char x) {
    return ((x & 0x0F) << 4 | (x & 0xF0) >> 4);
}

// Function to scan the entire keypad and update the currentKeyState array
void scan_keypad() {
    unsigned char keyState;

    // Scan each row of the keypad
    keyState = scan_row(1);
    currentKeyState[0] = swapNibbles(keyState);

    keyState = scan_row(2);
    currentKeyState[0] += keyState;

    keyState = scan_row(3);
    currentKeyState[1] = swapNibbles(keyState);

    keyState = scan_row(4);
    currentKeyState[1] += keyState;
}

// Function to check if there is a rising edge in the keypad state
int scan_keypad_rising_edge() {
    scan_keypad();

    unsigned char tempKeyState[2];
    tempKeyState[0] = currentKeyState[0];
    tempKeyState[1] = currentKeyState[1];

    // Wait for a short delay
```

```
    _delay_ms(10);

    scan_keypad();

    // Compare the states before and after the delay
    currentKeyState[0] &= tempKeyState[0];
    currentKeyState[1] &= tempKeyState[1];

    tempKeyState[0] = memory[0];
    tempKeyState[1] = memory[1];

    memory[0] = currentKeyState[0];
    memory[1] = currentKeyState[1];

    currentKeyState[0] &= ~tempKeyState[0];
    currentKeyState[1] &= ~tempKeyState[1];

    // Return true if any key is pressed
    return (currentKeyState[0] || currentKeyState[1]);
}

// Function to convert the current key state to ASCII representation
unsigned char keypad_to_ascii() {
    // Check each bit of the key states and return the corresponding ASCII
character
    if (currentKeyState[0] & 0x01)
        return '*';

    if (currentKeyState[0] & 0x02)
        return '0';

    if (currentKeyState[0] & 0x04)
        return '#';

    if (currentKeyState[0] & 0x08)
        return 'D';

    if (currentKeyState[0] & 0x10)
        return '7';

    if (currentKeyState[0] & 0x20)
        return '8';
```

```
    if (currentKeyState[0] & 0x40)
        return '9';

    if (currentKeyState[0] & 0x80)
        return 'C';

    if (currentKeyState[1] & 0x01)
        return '4';

    if (currentKeyState[1] & 0x02)
        return '5';

    if (currentKeyState[1] & 0x04)
        return '6';

    if (currentKeyState[1] & 0x08)
        return 'B';

    if (currentKeyState[1] & 0x10)
        return '1';

    if (currentKeyState[1] & 0x20)
        return '2';

    if (currentKeyState[1] & 0x40)
        return '3';

    if (currentKeyState[1] & 0x80)
        return 'A';

    return 0; // error
}

// Function to write two nibbles of data to the LCD
void write_2_nibbles(uint8_t input) {
    uint8_t temp = input;
    uint8_t pin = PIND;
    pin &= 0x0F;
    input &= 0xF0;
    input |= pin;
    PORTD = input;
    PORTD |= 0x08;
    PORTD &= 0xF7;
```

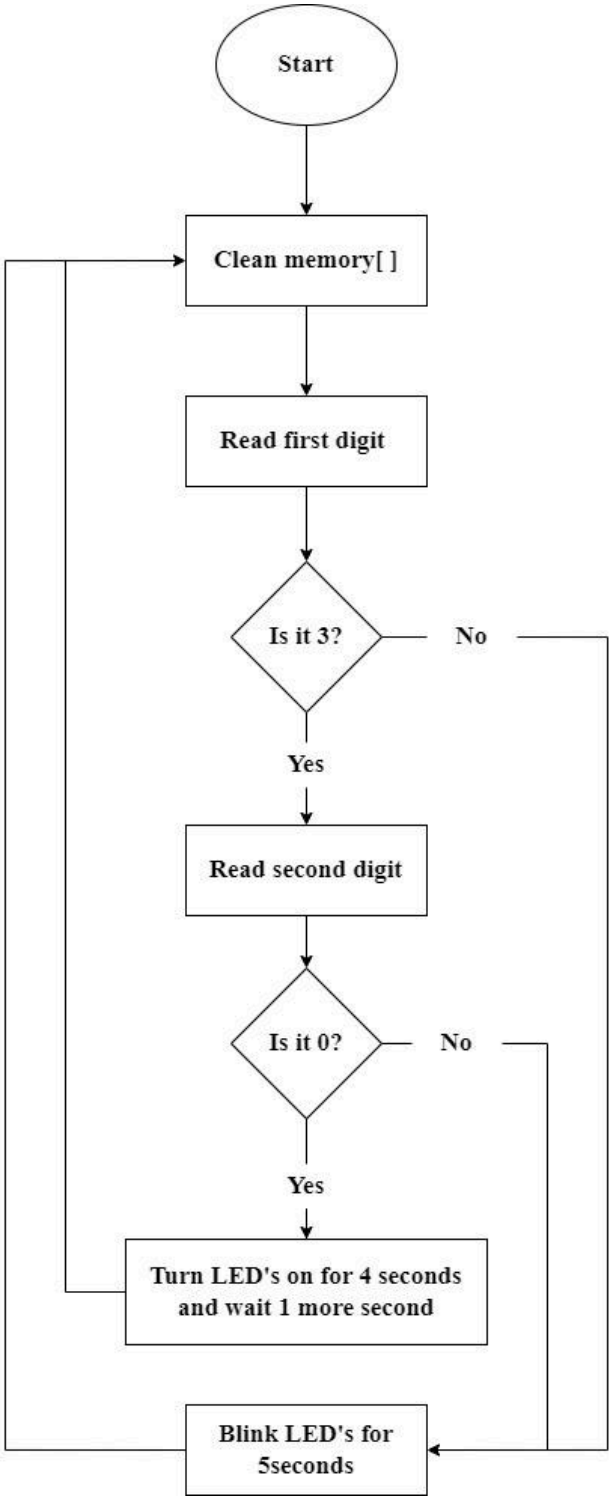
```
    input = temp;
    input &= 0x0F;
    input = input << 4;
    input |= pin;
    PORTD = input;
    PORTD |= 0x08;
    PORTD &= 0xF7;

    return;
}

// Function to send data to the LCD
void LCD_data(uint8_t x) {
    PORTD |= 0x04;
    write_2_nibbles(x);
    _delay_us(250);
    return;
}

// Function to send a command to the LCD
void LCD_command(uint8_t x) {
    PORTD &= 0xFB;
    write_2_nibbles(x);
    _delay_us(250);
    return;
}

// Function to clear the LCD display
void LCD_clear_display() {
    uint8_t x = 0x01;
    LCD_command(x);
    _delay_ms(5);
}

// Function to initialize the LCD
void LCD_init(void) {
    _delay_ms(200);

    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(250);
```

```
    PORTD = 0x30;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(250);

    PORTD = 0x20;
    PORTD |= 0x08;
    PORTD &= 0xF7;
    _delay_us(250);

    LCD_command(0x28);
    LCD_command(0x0C);

    LCD_clear_display();

    LCD_command(0x06);
}
// Function to display a "WRONG!" message on the LCD and wait for 2 seconds
void wrong() {
    LCD_init(); // Initialize the LCD
    for (int i = 0; i < 10; i++) {
        PORTB = 0xFF; // Set PORTB to 0xFF (all pins high)
        _delay_ms(250); // Wait for 250 milliseconds
        PORTB = 0x00; // Set PORTB to 0x00 (all pins low)
        _delay_ms(250); // Wait for 250 milliseconds
    }

    // Display "WRONG!" on the LCD
    LCD_data('W');
    LCD_data('R');
    LCD_data('O');
    LCD_data('N');
    LCD_data('G');
    LCD_data('!');
    _delay_ms(2000); // Wait for 2000 milliseconds
    LCD_clear_display(); // Clear the LCD display
}

int main() {
    twi_init(); // Initialize TWI (Two-Wire Interface)
    LCD_init(); // Initialize the LCD
    DDRB = 0xFF; // Set all pins of PORTB as output
```

```
    DDRD = 0xFF; // Set all pins of PORTD as output
    PCA9555_0_write(REG_CONFIGURATION_0, 0x00); // Write to PCA9555
configuration register 0
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0); // Write to PCA9555
configuration register 1

    char pressedKey1;
    char pressedKey2;

    while (1) {
        memory[0] = 0;
        memory[1] = 0;

        // Wait for a key press and store the pressed key
        while (1) {
            if (scan_keypad_rising_edge()) {
                pressedKey1 = keypad_to_ascii();
                break;
            }
        }

        // Check if the first key pressed is not '3', then call the wrong()
function
        if (pressedKey1 != '3') {
            wrong();
        }

        // Wait for another key press and store the pressed key
        while (1) {
            if (scan_keypad_rising_edge()) {
                pressedKey2 = keypad_to_ascii();
                break;
            }
        }

        // Check if the second key pressed is not '0', then call the
wrong() function
        if (pressedKey2 != '0') {
            wrong();
        } else {
            // Display "CORRECT! :)" message on the LCD
            PORTB = 0xFF;
            _delay_ms(4000);
```

```
            PORTB = 0x00;
            _delay_ms(1000);

            LCD_data('C');
            LCD_data('O');
            LCD_data('R');
            LCD_data('R');
            LCD_data('E');
            LCD_data('C');
            LCD_data('T');
            LCD_data('!');
            LCD_data(' ');
            LCD_data(' ');
            LCD_data(':');
            LCD_data(')');
            _delay_ms(2000);
            LCD_clear_display();
        }

        _delay_ms(5000); // Wait for 5000 milliseconds
    }
    return 0;
}
```

Στην κύρια συνάρτηση, όπως και στα παραπάνω ζητήματα, προετοιμάζεται το TWI, ρυθμίζεται η LCD οθόνη, διαμορφώνονται οι θύρες εισόδου-εξόδου και προετοιμάζεται η επέκταση I/O PCA9555. Στη συνέχεια, εκτελείται ένας συνεχής βρόχος, στον οποίο εξετάζονται τα πλήκτρα που πιέζονται στο keyboard. Ελέγχοντας την είσοδο, σε περίπτωση λάθους το πρόγραμμα εκτελεί τη λειτουργία void wrong(), κατά την οποία η LCD οθόνη εκτυπώνει το μήνυμα ‘WRONG!’ ενώ τα LEDs της PORTB αναβοσβήνουν ανά 0.25 sec για συνολική διάρκεια 5 sec. Αντίθετα, αν η είσοδος από το πληκτρολόγιο είναι η σωστή, δηλαδή ‘30’, τότε το πρόγραμμα εκτυπώνει στην LCD οθόνη το μήνυμα ‘CORRECT :)’ και τα τα LEDs της PORTB παραμένουν αναμμένα για 4 sec. Και στις δύο περιπτώσεις, μετά την οποιαδήποτε είσοδο διασφαλίζουμε πως στα επόμενα 5 sec δεν λαμβάνεται υπόψη άλλη είσοδος, αξιοποιώντας την εντολή `_delay_ms(5000)`.
Παρακάτω, φαίνεται το διάγραμμα ροής του ζητούμενου προγράμματος, που περιγράφει με ακρίβεια τη λειτουργία της συνάρτησης main().


Start
Clean memory[ ]
Read first digit
Is it 3?
No
Yes
Read second digit
Is it 0?
No
Yes
Turn LED's on for 4 seconds and wait 1 more second
Blink LED's for 5seconds