6η Εργαστηριακή Άσκηση

Πίκουλας Κωνσταντίνος *03120112*
Στάμος Ανδρέας *03120018*

1η Άσκηση

Εξήγηση

scan_row

Αρχικά φτιάχνουμε την συναρτηση *scan_row* η οποία μηδενίζει την γραμμή *row_to_check* την οποία δεχεται ως παράμετρο. Αφού μηδενιστεί η γραμμή, διαβάζονται τα pins IO1[4:7] που αντιστοιχούν σε inputs. Όποιο bit διαβάζουμε και ειναι 0 αντιστοιχεί σε πατημένο κουμπι (στην αντίστοιχη στήλη), ενώ ότι ειναι 1 δεν αντιστοιχεί σε πατημένο κουμπί. Περνάμε τα bits I01[4:7] που διαβάσαμε απο μια μάσκα 0xF0 και τα κάνουμε shift 4 θέσεις δεξιά (ώστε να τοποθετηθούν στα MSBs).

```
uint8_t scan_row(uint8_t row_to_check) {
    PCA9555_0_write(REG_OUTPUT_1, ~(1 << row_to_check));
    _delay_ms(10); // small delay
    uint8_t input_char = PCA9555_0_read(REG_INPUT_1);
    input_char = (input_char & 0xF0) >> 4;
    return input_char;
}
```

scan_keypad

Η *scan_keypad* φτιάχνει μια 16-bit μεταβλητή pressed_values την οποία στο τέλος της θα την επιστρέφει. Για κάθε γραμμή καλείται η συνάρτηση *scan_row* με παράμετρο την αντίστοιχη τιμή του loop ώστε να διαβάζονται τα πατημένα κουμπιά της γραμμής που εξετάζουμε. Με κάθε κλήση της *scan_row* (εκτός της τελευτάιας) κάνουμε shift αριστερά κατα 4 θεσεις το περιεχόμενο της μεταβλητής pressed_values και κάνουμε or την τιμή της με την τιμή που επιστρέφει η scan_row. Στο τέλος του loop η τιμή της pressed_values θα είναι 4 τετράδες που η κάθε τετράδα αντιστοιχέι στα πατημένα κουμπιά της αντίστοιχης γραμμής.

(Δηλαδή τα 4 MSBs θα αντιστοιχούν στην πρωτη γραμμή και τα 4 LSBs στην 4^{η} γραμμή). Η Boolean μεταβήτή activated χρησιμοποιείται μόνο στην άσκηση 6.1 έτσι ώστε το αντίστοιχο λαμπάκι να μένει αναμμένο όσο το αντίστοιχο κουμπί του πληκτρολογίου είναι πατημένο.

```
bool activated = false;
int scan_keypad() {
  int pressed_values = 0;
  for(uint8_t row = 0; row <= 3; ++row) {
    uint8_t pressed_pad = scan_row(row);
    pressed_values |= pressed_pad;
    if (row != 3) pressed_values = pressed_values << 4;
    asm("nop");
  };
```

scan_keypad_rising_edge

Η *scan_keypad_rising_edge* χρησιμοποιείται ώστε να «φιλτράρει» τα πατημένα κουμπιά. Αρχικά διαβάζουμε 2 φορες το πληκτρολογίο με ένα μικρό delay και κάνουμε or τα 2 αποτελέσματα ώστε να αποφύγουμε να διαβάσουμε σπινθυρισμούς. Έχουμε μια global 16-bit μεταβλητη (*pressed_pads_prev*) την οποία χρησιμοποιούμε ώστε να γνωρίζουμε το state του πληκτρολογίου την προηγούμενη φορά που το πατήσαμε, ώστε να γνωρίζουμε ποια είναι τα νέα κουμπιά που πατήθηκαν (τα πατημένα κουμπιά απο το προηγούμενο iteration δεν μετράνε ως πατημένα δηλαδή). Για να τα φιλτράρουμε κάνουμε or με το συμπλήρωμα της global μεταβλητής. Έπειτα πρωτού επιστρέψουμε τα «φιλτραρισμένα κουμπιά» θέτουμε την *pressed_pads_prev* στην τιμή που επιστρέφει η *scan_keypad*.

```
// in the beginning nothing is pressed
int pressed_pads_prev = 0xffff;
int scan_keypad_rising_edge() {
    int pressed_pads = scan_keypad();
    _delay_ms(10); // small delay to check pads
    int verified_pressed_pads = scan_keypad();
    // get rid of the pads that weren't previously pressed;
    verified_pressed_pads |= pressed_pads;
    // compare with previous pressed pads
    verified_pressed_pads |= ~pressed_pads_prev;
    pressed_pads_prev = pressed_pads;
    return verified_pressed_pads;
}
```

keypad_to_ascii

Η *keypad_to_ascii* διαβάζει το πληκτρολόγιο καλώντας την *scan_keypad_rising_edge* και βρίσκει αρχικά την γραμμή και έπειτα την στήλη στην οποία υπαρχει πατημένο κουμπί. Έπειτα επιστρέφει τον κωδικα ascii που αντιστοιχεί στο κουμπί αυτό. Για να βρούμε την γραμμή στην οποία υπάρχει πατημένο κουμπί αρκεί να βρουμε σε ποια τετράδα υπαρχει μηδενικό (δηλαδή πατημένο κουμπί). Έπειτα για να βρούμε ποια στήλη έχει πατηθεί θα πρέπει να βρούμε την θέση στην οποία βρίσκεται το μηδενικό στην τετράδα. Έπειτα επιστρέφουμε τον αντίστοιχο ascii κωδικό που βρίσκεται στον διδιάστατο πίνακα *keymap* στην αντίστοιχη γραμμή και στήλη.

```
char keypad_to_ascii() {
  const char keymap[4][4] = {
    {'1', '2', '3', 'A'},
    {'4', '5', '6', 'B'},
    {'7', '8', '9', 'C'},
    {'*', '0', '#', 'D'}
  };
  pressed_pad = scan_keypad_rising_edge();
  // if nothing is pressed then lights out
  if (pressed_pad == 0xffff) { return 0;}
  uint8_t row;
  for(row = 0; row <= 3; ++row) {
    if((pressed_pad & (0x0f)) == 0x0f) {
      pressed_pad = pressed_pad >> 4;
    } else break;
  }
    // if the pressed button on in this row..
    for (uint8_t bitPosition = 0; bitPosition <= 3; ++bitPosition) {
      if ((((pressed_pad & (0x0f)) >> bitPosition) & 1) == 0) {
        // Found a bit with value 0
        activated = true;
        return keymap[row][bitPosition];
      }
    }
 }
```

Κώδικας

```
#define F_CPU 16000000UL
#include<avr/io.h>
#include<avr/interrupt.h>
#include<util/delay.h>
#include<stdbool.h>

#define PCA9555_0_ADDRESS 0x40 // Address of PCA9555
#define TWI_READ 1
#define TWI_WRITE 0
#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 REGISTER
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)

// initialise 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 to transfer direction
// return 0 = device accessible
// return 1 = failed to access device
unsigned char twi_start(unsigned char address) {
    uint8_t twi_status;

    // send START
    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 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 is 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 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(uint8_t row_to_check) {
    PCA9555_0_write(REG_OUTPUT_1, ~(1 << row_to_check));
    _delay_ms(10); // small delay
    uint8_t input_char = PCA9555_0_read(REG_INPUT_1);
    input_char = (input_char & 0xF0) >> 4;
    return input_char;
}
bool activated = false;
int scan_keypad() {
    int pressed_values = 0;
    for(uint8_t row = 0; row <= 3; ++row) {
        uint8_t pressed_pad = scan_row(row);
```

```
        pressed_values |= pressed_pad;
        if (row != 3) pressed_values = pressed_values << 4;
        asm("nop");
    };
    if (pressed_values == 0xffff) activated = false;
    return pressed_values;
}

// in the beginning nothing is pressed
int pressed_pads_prev = 0xffff;
int scan_keypad_rising_edge() {
    int pressed_pads = scan_keypad();
    _delay_ms(10); // small delay to check pads
    int verified_pressed_pads = scan_keypad();
    // get rid of the pads that weren't previously pressed;
    verified_pressed_pads |= pressed_pads;
    // compare with previous pressed pads
    verified_pressed_pads |= ~pressed_pads_prev;
    pressed_pads_prev = pressed_pads;
    return verified_pressed_pads;
}

int pressed_pad;
char keypad_to_ascii() {
    const char keymap[4][4] = {
        {'1', '2', '3', 'A'},
        {'4', '5', '6', 'B'},
        {'7', '8', '9', 'C'},
        {'*', '0', '#', 'D'}
    };
    pressed_pad = scan_keypad_rising_edge();
    // if nothing is pressed then lights out
    if (pressed_pad == 0xffff) { return 0;}
    uint8_t row;
    for(row = 0; row <= 3; ++row) {
        if((pressed_pad & (0x0f)) == 0x0f) {
            pressed_pad = pressed_pad >> 4;
        } else break;
    }
        // if the pressed button on in this row..
        for (uint8_t bitPosition = 0; bitPosition <= 3; ++bitPosition) {
            if ((((pressed_pad & (0x0f)) >> bitPosition) & 1) == 0) {
                // Found a bit with value 0
                activated = true;
                return keymap[row][bitPosition];
            }
```

```
        }
    }

int main(int argc, char** argv) {

    // init TWI process
    twi_init();

    // set PORTB as output
    DDRB = 0xFF;

    // IO1[0:3] -> output
    // IO1[4:7] -> input
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0);


    while (1) {
        char pressed = keypad_to_ascii();
        if (pressed == '1') PORTB = 0x01;
        if (pressed == '5') PORTB = 0x02;
        if (pressed == '9') PORTB = 0x04 ;
        if (pressed == 'D') PORTB = 0x08;
        if (pressed == 0 && !activated) PORTB = 0x00;
    }
}
```

2η Άσκηση

Εξήγηση

Για την υλοποίηση της άσκησης αρχικοποιήσαμε την οθόνη lcd. Διαβάζουμε το πληκτρολόγιο χρησιμοποιώντας την συνάρτηση *scan_keyp*ad και στέλνουμε τον χαρακτήρα τύπου char που διαβάσαμε, στην οθόνη χρησιμοποιώτας την *lcd_data.*

Οι αλλαγές στην περίπτωση της άσκησης 2 είναι πως δεν έχουμε Boolean μεταβλητή *activated* αλλά και ότι εαν δεν έχει πατηθεί τίποτα η keypad_to_ascii επιστρέφει «0», έτσι ώστε να μην στείλουμε δεδομένα στην οθόνη.

Κώδικας

```
#define F_CPU 16000000UL
#include<avr/io.h>
#include<avr/interrupt.h>
#include<util/delay.h>


#define PCA9555_0_ADDRESS 0x40 // Address of PCA9555
#define TWI_READ 1
#define TWI_WRITE 0
#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 REGISTER
typedef enum {
    REG_INPUT_0         = 0,
    REG_INPUT_1         = 1,
    REG_OUTPUT_0        = 2,
    REG_OUTPUT_1        = 3,
    REG_POLARITY_INV_0  = 4,
    REG_POLARITY_INV_1  = 5,
    REG_CONFIGURATION_0 = 6,
    REG_CONFIGURATION_1 = 7,
} PCA9555_REGISTERS;
```

```
// ------- Master Transmitter/Receiver ---------
#define TW_START       0x08
#define TW_REP_START    0x10

// ------- Master Transmitter ------------
#define TW_MT_SLA_ACK   0x18
#define TW_MT_SLA_NACK  0x20
#define TW_MT_DATA_ACK  0x28


// -------- Master Receiver ------------
#define TW_MR_SLA_ACK   0x40
#define TW_MR_SLA_NACK  0x48
#define TW_MR_DATA_NACK 0x58

#define TW_STATUS_MASK 0b1111000
#define TW_STATUS (TWSR0 & TW_STATUS_MASK)

// initialise 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 to transfer direction
// return 0 = device accessible
// return 1 = failed to access device
unsigned char twi_start(unsigned char address) {
   uint8_t twi_status;

   // send START
```

```
    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 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 is 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 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 command;
uint8_t command_temp;
void write_2_nibbles() {
    uint8_t r25 = PIND;
    r25 &= 0x0f;
    command_temp = command & 0xf0;
    command_temp |= r25;
    PORTD = command_temp;
    PORTD |= (1<<3); // Enable Pulse
    asm("nop");
    asm("nop");
    PORTD &= ~(1<<3); // Clear Pulse

    command_temp = (command & 0x0f) << 4;
    command_temp |= r25;
    PORTD = command_temp;
    PORTD |= (1<<3); // Enable Pulse
    asm("nop");
    asm("nop");
    PORTD &= ~(1<<3); // Clear Pulse
    return;
}

void lcd_data() {
    PORTD |= (1<<2); //RS=1 (data)
    write_2_nibbles();
    _delay_ms(0.250);
    return;
```

```
}

void lcd_command() {
    PORTD &= ~(1 << 2); // Clear Enable
    write_2_nibbles();
    _delay_ms(0.250);
    return;
}

void lcd_clear_display(){
    command = 0x01;
    lcd_command();
    _delay_ms(5);
    return;
}

uint8_t scan_row(uint8_t row_to_check) {
    PCA9555_0_write(REG_OUTPUT_1, ~(1 << row_to_check));
//    _delay_ms(10); // small delay
    uint8_t input_char = PCA9555_0_read(REG_INPUT_1);
    input_char = (input_char & 0xF0) >> 4;
    return input_char;
}


int scan_keypad() {
    int pressed_values = 0;
    for(uint8_t row = 0; row <= 3; ++row) {
        uint8_t pressed_pad = scan_row(row);
        pressed_values |= pressed_pad;
        if (row != 3) pressed_values = pressed_values << 4;
        asm("nop");
    };
    return pressed_values;
}

// in the beginning nothing is pressed
int pressed_pads_prev = 0xffff;

int scan_keypad_rising_edge() {
    int pressed_pads = scan_keypad();
    _delay_ms(30); // small delay to check pads
    int verified_pressed_pads = scan_keypad();
    // get rid of the pads that weren't previously pressed;
    verified_pressed_pads |= pressed_pads;
    int temp = verified_pressed_pads;
```

```
    // compare with previous pressed pads
    verified_pressed_pads |= ~pressed_pads_prev;
    pressed_pads_prev = temp;
    return verified_pressed_pads;
}

char keypad_to_ascii() {
    const char keymap[4][4] = {
        {'1', '2', '3', 'A'},
        {'4', '5', '6', 'B'},
        {'7', '8', '9', 'C'},
        {'*', '0', '#', 'D'}
    };
    uint16_t pressed_pad = scan_keypad_rising_edge();
    // if nothing is pressed then lights out
    if (pressed_pad == 0xffff) return 0;
    uint8_t row;
    for(row = 0; row <= 3; ++row) {
        if((pressed_pad & (0x0f)) == 0x0f) {
            pressed_pad = pressed_pad >> 4;
        } else break;
    }
        // if the pressed button on in this row..
        for (uint8_t bitPosition = 0; bitPosition <= 3; ++bitPosition) {
            if ((((pressed_pad & (0x0f)) >> bitPosition) & 1) == 0) {
                // Found a bit with value 0
                return keymap[row][bitPosition];
            }
        }
   }


int main(int argc, char** argv) {

    // init TWI process
    twi_init();

    // set PORTB as output
    DDRB = 0xFF;
    DDRD = 0xFF;
    // IO1[0:3] -> output
    // IO1[4:7] -> input
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0);
    PCA9555_0_write(REG_OUTPUT_1, 1);
```

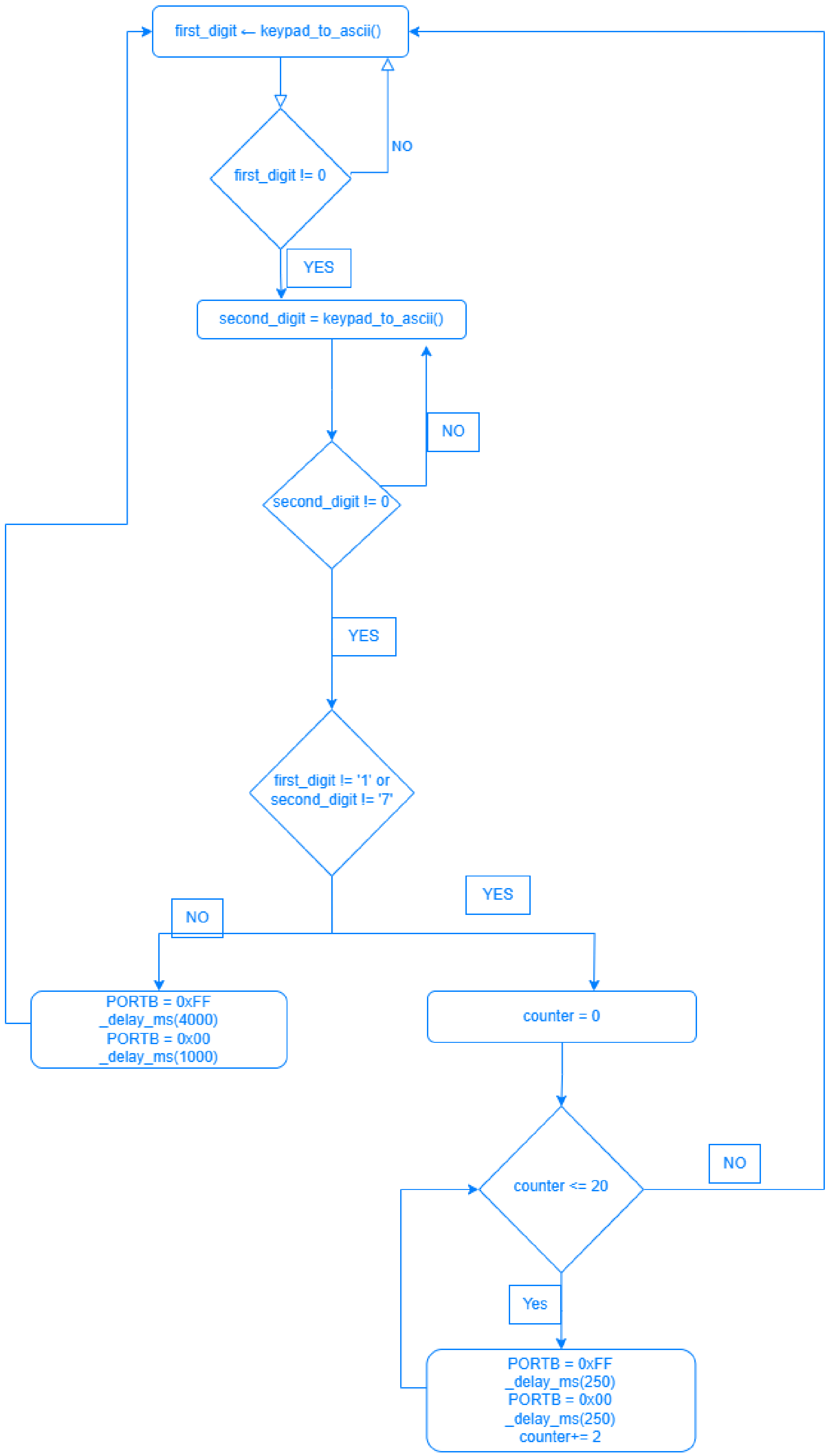
```
    // lcd initialization
    _delay_ms(200); // wait for screen to initialize
    for(uint8_t i =0; i<3; ++i) {
        PORTD = 0x30; // set 8 bit mode 3 times
        PORTD |= (1<<3); // Enable pulse
        asm("nop");
        asm("nop");
        PORTD &= ~(1 << 3); // Clear Enable
        _delay_ms(0.250);
    }

    // switch to 4bit mode
    PORTD = 0x20;
    PORTD |= (1<<3); // Enable pulse
    asm("nop");
    asm("nop");
    PORTD &= ~(1 << 3); // Clear Enable
    _delay_ms(0.250);

    command = 0x28;
    lcd_command();

    command = 0x0c;
    lcd_command();

    while (1) {
        char pressed = keypad_to_ascii();
        if (pressed == 0) continue;
        lcd_clear_display();
        command = pressed; lcd_data();
    }
}
```

3η Άσκηση

Εξήγηση

Ο κωδικός της ομάδας μας είναι «17». Όταν στο πληκτρολόγιο δίνονται τα 2 ψηφία με την σωστή σειρά (πρώτα το «1» και μετα το «7»), τότε ανάβουν τα led PB0 με PB5 για 4 δευτερόλεπτα (έπειτα σβήνουν) και καλούμε ύστερα την *_delay_ms(1000)* ώστε να περιμένουμε 1 δευτερόλεπτο πρωτού δεχθούμε άλλο input (συνολικά 5 δευτερόλεπτα απο την στιγμή που το πρόγραμμα μας δέχθηκε έναν συνδυασμό). Έαν δεν δοθεί ο κατάλληλος συνδυαμός τότε τα led αναβοσβήνουν (250ms αναμμένα, 250ms σβηστά) για 5 δευτερόλεπτα. Τα ψηφία διαβαζονται από το πληκτρολόγιο κάθε φορά, καλώντας την *keypad_to_ascii* για κάθε ψηφίο που θέλουμε να διαβάσουμε.

Διάγραμμα Ροής



Κώδικας

```
#define F_CPU 16000000UL
#include<avr/io.h>
#include<avr/interrupt.h>
#include<util/delay.h>


#define PCA9555_0_ADDRESS 0x40 // Address of PCA9555
#define TWI_READ 1
#define TWI_WRITE 0
#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 REGISTER
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
// initialise 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 to transfer direction
// return 0 = device accessible
// return 1 = failed to access device
unsigned char twi_start(unsigned char address) {
    uint8_t twi_status;

    // send START
    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 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 is 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 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;
}
```

```c
uint8_t scan_row(uint8_t row_to_check) {
    PCA9555_0_write(REG_OUTPUT_1, ~(1 << row_to_check));
    _delay_ms(10); // small delay
    uint8_t input_char = PCA9555_0_read(REG_INPUT_1);
    input_char = (input_char & 0xF0) >> 4;
    return input_char;
}


int scan_keypad() {
    int pressed_values = 0;
    for(uint8_t row = 0; row <= 3; ++row) {
        uint8_t pressed_pad = scan_row(row);
        pressed_values |= pressed_pad;
        if (row != 3) pressed_values = pressed_values << 4;
        asm("nop");
    };
    return pressed_values;
}

// in the beginning nothing is pressed
int pressed_pads_prev = 0xffff;

int scan_keypad_rising_edge() {
    int pressed_pads = scan_keypad();
    _delay_ms(10); // small delay to check pads
    int verified_pressed_pads = scan_keypad();
    // get rid of the pads that weren't previously pressed;
    verified_pressed_pads |= pressed_pads;
    int temp = verified_pressed_pads;
    // compare with previous pressed pads
    verified_pressed_pads |= ~pressed_pads_prev;
    pressed_pads_prev = temp;
    return verified_pressed_pads;
}

char keypad_to_ascii() {
    const char keymap[4][4] = {
        {'1', '2', '3', 'A'},
        {'4', '5', '6', 'B'},
        {'7', '8', '9', 'C'},
        {'*', '0', '#', 'D'}
    };
```

```
    uint16_t pressed_pad = scan_keypad_rising_edge();
    // if nothing is pressed then lights out
    if (pressed_pad == 0xffff) return 0;
    uint8_t row;
    for(row = 0; row <= 3; ++row) {
        if((pressed_pad & (0x0f)) == 0x0f) {
            pressed_pad = pressed_pad >> 4;
        } else break;
    }
        // if the pressed button on in this row..
        for (uint8_t bitPosition = 0; bitPosition <= 3; ++bitPosition) {
            if ((((pressed_pad & (0x0f)) >> bitPosition) & 1) == 0) {
                // Found a bit with value 0
                return keymap[row][bitPosition];
            }
        }
   }

int main(int argc, char** argv) {

    // init TWI process
    twi_init();

    // set PORTB as output
    DDRB = 0xFF;

    // IO1[0:3] -> output
    // IO1[4:7] -> input
    PCA9555_0_write(REG_CONFIGURATION_1, 0xF0);

    // TEAM CODE 17

    while(1) {
        char first_digit = keypad_to_ascii();
        if (first_digit == 0) continue;
        // if first_digit has been given
        asm("nop");
        while(1){
            char second_digit = keypad_to_ascii();
            if (second_digit == 0) continue;
            asm("nop");
            // if second digit has been given
            if (first_digit != '1' || second_digit != '7') {
                uint8_t counter = 0;
                while(counter <= 20) {
                    PORTB = 0xFF;
```

```
                    _delay_ms(250);
                    PORTB = 0x00;
                    _delay_ms(250);
                    counter+=2;
                }
            } else {
                PORTB = 0xff;
                _delay_ms(4000);
                PORTB = 0x00;
                _delay_ms(1000);
            }
            break;
            }
        }
    }
```