Χρήση πληκτρολογίου 4⊠4 σε θύρα επέκτασης στον AVR

Εργαστήριο Μικροϋπολογιστών

Κριθαρίδης Κωνσταντίνος, *el*21045 Μπαλάτος Δημήτριος, *el*21170

26 Νοεμβρίου 2024

1 Ζήτημα 6.1

Αρχικά, μεταφέρουμε τον κώδικα για χρήση του `Port Expander` . Θέτουμε τα bits 0…3 του `EXT_PORT1` (συνδεδεμένα στο PIND) ως έξοδο και τα 4…7 ως είσοδο για να μπορούμε να καταλαβαίνουμε ποιά κουμπιά πατήθηκαν στο πληκτρολόγιο. Δημιουργούμε συνάρτηση `scan_row(row)` που δέχεται ως είσοδο το ποιά γραμμή πρέπει να διαβάσει και επιστρέφει σε 4 bits (σε θετική λογική) το ποιά κουμπιά της γραμμής είναι πατημένα. Η `scan_keypad()` επιστρέφει σε 16 bits ποιά κουμπιά του πληκτρολογίου είναι πατημένα καλώντας την scan_row για κάθε γραμμή και ενώνοντας σε έναν uint16_t το αποτέλεσμα. Η `scan_keypad_rising_edge()` διατηρεί σε μια static μεταβλητή την προηγούμενη κατάσταση του πληκτρολογίου, διαβάζει τη νέα κατάσταση (δύο φορές, με καθυστέρηση ενδιάμεσα, για να αποφύγει τον σπινθηρισμό) και εντοπίζει ποιό κουμπί πατήθηκε μόλις. Η `keypad_to_ascii()` καλεί τη `scan_keypad()` (γιατί θέλουμε όταν μένει πατημένο ένα κουμπί να μένει αναμμένο το λεντάκι, επίσης, πάλι αποφεύγουμε τον σπινθηρισμό) και επιστρέφει τον χαρακτήρα που αντιστοιχεί στο κουμπί που έχει πατηθεί. Τέλος, η `char_to_led()` καλεί την `keypad_to_ascii()` και ανάλογα με τον χαρακτήρα που διαβάστηκε, ανάβει το αντίστοιχο LED του `PORTB` .

```c
/*
 * main.c
 *
 * Created: 11/22/2024 8:58:52 AM
 *  Author: User
 */


#include <xc.h>

#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
```

```
22
23  #define NOP() do { __asm__ __volatile__ ( "nop "); } while (0)
24
25  // PCA9555 REGISTERS
26  typedef enum {
27          REG_INPUT_0 = 0,
28          REG_INPUT_1 = 1,
29          REG_OUTPUT_0 = 2,
30          REG_OUTPUT_1 = 3,
31          REG_POLARITY_INV_0 = 4,
32          REG_POLARITY_INV_1 = 5,
33          REG_CONFIGURATION_0 = 6,
34          REG_CONFIGURATION_1 = 7
35  } PCA9555_REGISTERS;
36
37  //----------- Master Transmitter/Receiver -------------------
38  #define TW_START 0x08
39  #define TW_REP_START 0x10
40
41  //---------------- Master Transmitter ----------------------
42  #define TW_MT_SLA_ACK 0x18
43  #define TW_MT_SLA_NACK 0x20
44  #define TW_MT_DATA_ACK 0x28
45
46  //---------------- Master Receiver ----------------
47  #define TW_MR_SLA_ACK 0x40
48  #define TW_MR_SLA_NACK 0x48
49  #define TW_MR_DATA_NACK 0x58
50
51  #define TW_STATUS_MASK 0b11111000
52  #define TW_STATUS (TWSR0 & TW_STATUS_MASK)
53
54  //initialize TWI clock
55  void twi_init(void)
56  {
57          TWSR0 = 0; // PRESCALER_VALUE=1
58          TWBR0 = TWBR0_VALUE; // SCL_CLOCK 100KHz
59  }
60
61  // Read one byte from the twi device (request more data from device)
62  unsigned char twi_readAck(void)
63  {
64          TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
65          while(!(TWCR0 & (1<<TWINT)));
66          return TWDR0;
67  }
68
69  //Read one byte from the twi device, read is followed by a stop condition
70  unsigned char twi_readNak(void)
71  {
72          TWCR0 = (1<<TWINT) | (1<<TWEN);
```

```
        while(!(TWCR0 & (1<<TWINT)));
        return TWDR0;
}

// Issues a start condition and sends address and transfer direction.
// return 0 = device accessible, 1= failed to access device
unsigned char twi_start(unsigned char address)
{
        uint8_t twi_status;

        // send START condition
        TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);

        // wait until transmission completed
        while(!(TWCR0 & (1<<TWINT)));

        // check value of TWI Status Register.
        twi_status = TW_STATUS & 0xF8;
        if ( (twi_status != TW_START) && (twi_status != TW_REP_START)) return 1;

        // send device address
        TWDR0 = address;
        TWCR0 = (1<<TWINT) | (1<<TWEN);

        // wail until transmission completed and ACK/NACK has been received
        while(!(TWCR0 & (1<<TWINT)));
        // check value of TWI Status Register.
        twi_status = TW_STATUS & 0xF8;
        if ( (twi_status != TW_MT_SLA_ACK) && (twi_status != TW_MR_SLA_ACK) )
        {
                return 1;
        }
        return 0;
}

// Send start condition, address, transfer direction.
// Use ack polling to wait until device is ready
void twi_start_wait(unsigned char address)
{
        uint8_t twi_status;
        while ( 1 )
        {
                // send START condition
                TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);

                // wait until transmission completed
                while(!(TWCR0 & (1<<TWINT)));

                // check value of TWI Status Register.
                twi_status = TW_STATUS & 0xF8;
                if ( (twi_status != TW_START) && (twi_status != TW_REP_START))
                 ↪  continue;
```

```
124
125                 // send device address
126                 TWDR0 = address;
127                 TWCR0 = (1<<TWINT) | (1<<TWEN);
128
129                 // wail until transmission completed
130                 while(!(TWCR0 & (1<<TWINT)));
131
132                 // check value of TWI Status Register.
133                 twi_status = TW_STATUS & 0xF8;
134                 if ( (twi_status == TW_MT_SLA_NACK )||(twi_status
                     ↪  ==TW_MR_DATA_NACK) )
135                 {
136                         /* device busy, send stop condition to terminate write
                             ↪  operation */
137                         TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
138
139                         // wait until stop condition is executed and bus released
140                         while(TWCR0 & (1<<TWSTO));
141                         continue;
142                 }
143                 break;
144         }
145 }
146
147 // Send one byte to twi device, Return 0 if write successful or 1 if write failed
148 unsigned char twi_write( unsigned char data )
149 {
150         // send data to the previously addressed device
151         TWDR0 = data;
152         TWCR0 = (1<<TWINT) | (1<<TWEN);
153
154         // wait until transmission completed
155         while(!(TWCR0 & (1<<TWINT)));
156         if( (TW_STATUS & 0xF8) != TW_MT_DATA_ACK) return 1;
157         return 0;
158 }
159
160 // Send repeated start condition, address, transfer direction
161 //Return: 0 device accessible
162 // 1 failed to access device
163 unsigned char twi_rep_start(unsigned char address)
164 {
165         return twi_start( address );
166 }
167
168 // Terminates the data transfer and releases the twi bus
169 void twi_stop(void)
170 {
171         // send stop condition
172         TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
```

```
173
174         // wait until stop condition is executed and bus released
175         while(TWCR0 & (1<<TWSTO));
176 }
177
178 uint8_t LAST;
179
180 void PCA9555_0_write(PCA9555_REGISTERS reg, uint8_t value)
181 {
182         twi_start_wait(PCA9555_0_ADDRESS + TWI_WRITE);
183         twi_write(reg);
184         twi_write(value);
185         twi_stop();
186         LAST = value;
187         //if (reg != REG_CONFIGURATION_0) exit(0);
188 }
189
190 uint8_t PCA9555_0_read(PCA9555_REGISTERS reg)
191 {
192         uint8_t ret_val;
193         twi_start_wait(PCA9555_0_ADDRESS + TWI_WRITE);
194         twi_write(reg);
195         twi_rep_start(PCA9555_0_ADDRESS + TWI_READ);
196         ret_val = twi_readNak();
197         twi_stop();
198         return ret_val;
199 }
200
201 uint8_t scan_row(uint8_t row){ //row = 0, 1, 2, 3
202         uint8_t mask = 0x0f & ~(1<<row);
203         PCA9555_0_write(REG_OUTPUT_1, mask); //enable row as input
204         _delay_us(100);
205         uint8_t in = ~PCA9555_0_read(REG_INPUT_1); //read columns of row pressed
          ↪  in positive logic
206         in >>= 4; //remove IO1[0:3]
207         return in; //4 bits
208 }
209
210 uint16_t scan_keypad(){
211         uint16_t row0 = scan_row(0);
212         uint16_t row1 = scan_row(1);
213         uint16_t row2 = scan_row(2);
214         uint16_t row3 = scan_row(3);
215         return row0 | (row1<<4) | (row2<<8) | (row3<<12);
216 }
217
218 uint16_t scan_keypad_rising_edge(){
219         static uint16_t pressed_keys = 0;
220         uint16_t pressed_keys_tempo = scan_keypad();
221         _delay_ms(15); //wait to avoid triggering
222         pressed_keys_tempo &= scan_keypad(); //only keep the actual buttons
 ↪  pressed
```

```
223         uint16_t keys_just_pressed = pressed_keys_tempo & (~pressed_keys);
224         pressed_keys = pressed_keys_tempo;
225         return keys_just_pressed;
226 }
227 
228 char keypad_to_ascii(){
229         uint16_t key = scan_keypad();
230         _delay_ms(15); //wait to avoid triggering
231         key &= scan_keypad(); //only keep the actual buttons pressed
232         if(key&(1<<0)) return '*';
233         if(key&(1<<1)) return '0';
234         if(key&(1<<2)) return '#';
235         if(key&(1<<3)) return 'D';
236         if(key&(1<<4)) return '7';
237         if(key&(1<<5)) return '8';
238         if(key&(1<<6)) return '9';
239         if(key&(1<<7)) return 'C';
240         if(key&(1<<8)) return '4';
241         if(key&(1<<9)) return '5';
242         if(key&(1<<10)) return '6';
243         if(key&(1<<11)) return 'B';
244         if(key&(1<<12)) return '1';
245         if(key&(1<<13)) return '2';
246         if(key&(1<<14)) return '3';
247         if(key&(1<<15)) return 'A';
248         return 0;
249 }
250 
251 void char_to_led(){
252         char c = keypad_to_ascii();
253         switch(c){
254         case 'A':
255                 PORTB = 0x01;
256                 break;
257         case '8':
258                 PORTB = 0x02;
259                 break;
260         case '6':
261                 PORTB = 0x04;
262                 break;
263         case '*':
264                 PORTB = 0x08;
265                 break;
266         default:
267                 PORTB = 0;
268                 break;
269         }
270         return;
271 }
272 
273 int main(void) {
```

```
274         DDRB = 0xff; //Set PORTB as output
275         twi_init();
276         PCA9555_0_write(REG_CONFIGURATION_1, 0xf0); //Set EXT_PORT1 as: 0:3 ->
    ↪  output
277
278         while(1){
279                 _delay_ms(50);
280                 char_to_led();
281         }
282 }
```

2 Ζήτημα 6.2

Στην άσκηση αυτή, μεταφέραμε τον κώδικα της Άσκησης 6.1 (όπου στο `keypad_to_ascii()` καλούμε πλέον την `scan_keypad_rising_edge()`) και τον κώδικα για επικοινωνία με την οθόνη LCD μέσω του `EXT_PORT0` του `PORT Expander` , αφού τα `PIND` χρησιμοποιούνται ήδη από το `EXT_PORT1` του πληκτρολογίου. Δημιουργούμε επιπλέον συνάρτηση `char_to_lcd()` που τυπώνει στην οθόνη τον χαρακτήρα του κουμπιού που μόλις πατήθηκε. Ο χαρακτήρας αυτός παραμένει στην οθόνη μέχρι να πατηθεί διαφορετικό κουμπί.

```
1  /*
2   * main.c
3   *
4   * Created: 11/22/2024 10:09:08 AM
5   *  Author: User
6   */
7
8  #include <xc.h>
9
10 #define F_CPU 16000000UL
11 #include<avr/io.h>
12 #include<avr/interrupt.h>
13 #include<util/delay.h>
14 #define PCA9555_0_ADDRESS 0x40 //A0=A1=A2=0 by hardware
15 #define TWI_READ 1 // reading from twi device
16 #define TWI_WRITE 0 // writing to twi device
17 #define SCL_CLOCK 100000L // twi clock in Hz
18
19 //Fscl=Fcpu/(16+2*TWBR0_VALUE*PRESCALER_VALUE)
20 #define TWBR0_VALUE ((F_CPU/SCL_CLOCK)-16)/2
21
22 #define NOP() do { __asm__ __volatile__ ( "nop "); } while (0)
23
24 // PCA9555 REGISTERS
25 typedef enum {
26         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
```

```
78  unsigned char twi_start(unsigned char address)
79  {
80          uint8_t twi_status;
81
82          // send START condition
83          TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
84
85          // wait until transmission completed
86          while(!(TWCR0 & (1<<TWINT)));
87
88          // check value of TWI Status Register.
89          twi_status = TW_STATUS & 0xF8;
90          if ( (twi_status != TW_START) && (twi_status != TW_REP_START)) return 1;
91
92          // send device address
93          TWDR0 = address;
94          TWCR0 = (1<<TWINT) | (1<<TWEN);
95
96          // wail until transmission completed and ACK/NACK has been received
97          while(!(TWCR0 & (1<<TWINT)));
98          // check value of TWI Status Register.
99          twi_status = TW_STATUS & 0xF8;
100         if ( (twi_status != TW_MT_SLA_ACK) && (twi_status != TW_MR_SLA_ACK) )
101         {
102                 return 1;
103         }
104         return 0;
105 }
106
107 // Send start condition, address, transfer direction.
108 // Use ack polling to wait until device is ready
109 void twi_start_wait(unsigned char address)
110 {
111         uint8_t twi_status;
112         while ( 1 )
113         {
114                 // send START condition
115                 TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
116
117                 // wait until transmission completed
118                 while(!(TWCR0 & (1<<TWINT)));
119
120                 // check value of TWI Status Register.
121                 twi_status = TW_STATUS & 0xF8;
122                 if ( (twi_status != TW_START) && (twi_status != TW_REP_START))
                    ↪  continue;
123
124                 // send device address
125                 TWDR0 = address;
126                 TWCR0 = (1<<TWINT) | (1<<TWEN);
127
```

```
                // wail until transmission completed
                while(!(TWCR0 & (1<<TWINT)));

                // check value of TWI Status Register.
                twi_status = TW_STATUS & 0xF8;
                if ( (twi_status == TW_MT_SLA_NACK )||(twi_status
                 ↪  ==TW_MR_DATA_NACK) )
                {
                        /* device busy, send stop condition to terminate write
                         ↪  operation */
                        TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

                        // wait until stop condition is executed and bus released
                        while(TWCR0 & (1<<TWSTO));
                        continue;
                }
                break;
        }
}

// Send one byte to twi device, Return 0 if write successful or 1 if write failed
unsigned char twi_write( unsigned char data )
{
        // send data to the previously addressed device
        TWDR0 = data;
        TWCR0 = (1<<TWINT) | (1<<TWEN);

        // wait until transmission completed
        while(!(TWCR0 & (1<<TWINT)));
        if( (TW_STATUS & 0xF8) != TW_MT_DATA_ACK) return 1;
        return 0;
}

// Send repeated start condition, address, transfer direction
//Return: 0 device accessible
// 1 failed to access device
unsigned char twi_rep_start(unsigned char address)
{
        return twi_start( address );
}

// Terminates the data transfer and releases the twi bus
void twi_stop(void)
{
        // send stop condition
        TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

        // wait until stop condition is executed and bus released
        while(TWCR0 & (1<<TWSTO));
}
```

```
uint8_t LAST;

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();
        LAST = value;
        //if (reg != REG_CONFIGURATION_0) exit(0);
}

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){ //row = 0, 1, 2, 3
        uint8_t mask = 0x0f & ~(1<<row);
        PCA9555_0_write(REG_OUTPUT_1, mask); //enable row as input
        _delay_us(100);
        uint8_t in = ~PCA9555_0_read(REG_INPUT_1); //read columns of row pressed
         ↪  in positive logic
        in >>= 4; //remove IO1[0:3]
        return in; //4 bits
}

uint16_t scan_keypad(){
        uint16_t row0 = scan_row(0);
        uint16_t row1 = scan_row(1);
        uint16_t row2 = scan_row(2);
        uint16_t row3 = scan_row(3);
        return row0 | (row1<<4) | (row2<<8) | (row3<<12);
}

uint16_t scan_keypad_rising_edge(){
        static uint16_t pressed_keys = 0;
        uint16_t pressed_keys_tempo = scan_keypad();
        _delay_ms(15); //wait to avoid triggering
        pressed_keys_tempo &= scan_keypad(); //only keep the actual buttons
 ↪  pressed
        uint16_t keys_just_pressed = pressed_keys_tempo & (~pressed_keys);
        pressed_keys = pressed_keys_tempo;
        return keys_just_pressed;
}
```

```
226
227 char keypad_to_ascii(){
228         uint16_t key = scan_keypad_rising_edge();
229         if(key&(1<<0)) return '*';
230         if(key&(1<<1)) return '0';
231         if(key&(1<<2)) return '#';
232         if(key&(1<<3)) return 'D';
233         if(key&(1<<4)) return '7';
234         if(key&(1<<5)) return '8';
235         if(key&(1<<6)) return '9';
236         if(key&(1<<7)) return 'C';
237         if(key&(1<<8)) return '4';
238         if(key&(1<<9)) return '5';
239         if(key&(1<<10)) return '6';
240         if(key&(1<<11)) return 'B';
241         if(key&(1<<12)) return '1';
242         if(key&(1<<13)) return '2';
243         if(key&(1<<14)) return '3';
244         if(key&(1<<15)) return 'A';
245         return 0;
246 }
247
248 void flash ()
249 {
250         _delay_us(50);
251         uint8_t tmp = PCA9555_0_read(REG_INPUT_0);
252         PCA9555_0_write(REG_OUTPUT_0, tmp | (1 << 3));
253         _delay_us(50);
254         PCA9555_0_write(REG_OUTPUT_0, tmp & ~(1 << 3));
255 }
256
257 void write_2_nibbles(uint8_t data){
258         uint8_t temp = LAST & 0x0f;
259         uint8_t out = data & 0xf0 | temp;
260         PCA9555_0_write(REG_OUTPUT_0, out);
261         flash();
262
263         out = (data << 4) & 0xf0 | temp;
264         PCA9555_0_write(REG_OUTPUT_0, out);
265         flash();
266 }
267
268 void lcd_data (uint8_t data)
269 {
270         uint8_t tmp = LAST;
271         PCA9555_0_write(REG_OUTPUT_0, tmp | (1 << 2));
272         write_2_nibbles(data);
273         _delay_us(500);
274 }
275
276 void lcd_command (uint8_t instr)
```

```
{
        uint8_t tmp = LAST;
        PCA9555_0_write(REG_OUTPUT_0, tmp & ~(1 << 2));
        write_2_nibbles(instr);
        _delay_us(500);
}

void lcd_clear_display(){
        lcd_command(0x01);
        _delay_ms(200);
}

void lcd_init ()
{
        _delay_ms(200);

        uint8_t out = 0x30;
        for (int i=0; i<3; ++i) {
                PCA9555_0_write(REG_OUTPUT_0, out);
                flash();
                _delay_us(250);
        }
        PCA9555_0_write(REG_OUTPUT_0, 0x20);
        flash();
        _delay_us(250);

        lcd_command(0x28);
        lcd_command(0x0c);
        lcd_clear_display();
        lcd_command(0x06);
}

void lcd_string (const char* str)
{
        lcd_clear_display();
        for (; *str; str++) {
                if (*str == '\n')
                lcd_command(0xc0);
                else
                lcd_data(*str);
        }
}

void lcd_digit(uint8_t digit){
        lcd_data(0x30 + digit);
}

void lcd_number(uint32_t number){
        uint8_t digits[10];
        int i = 0;
        if(number == 0){
```

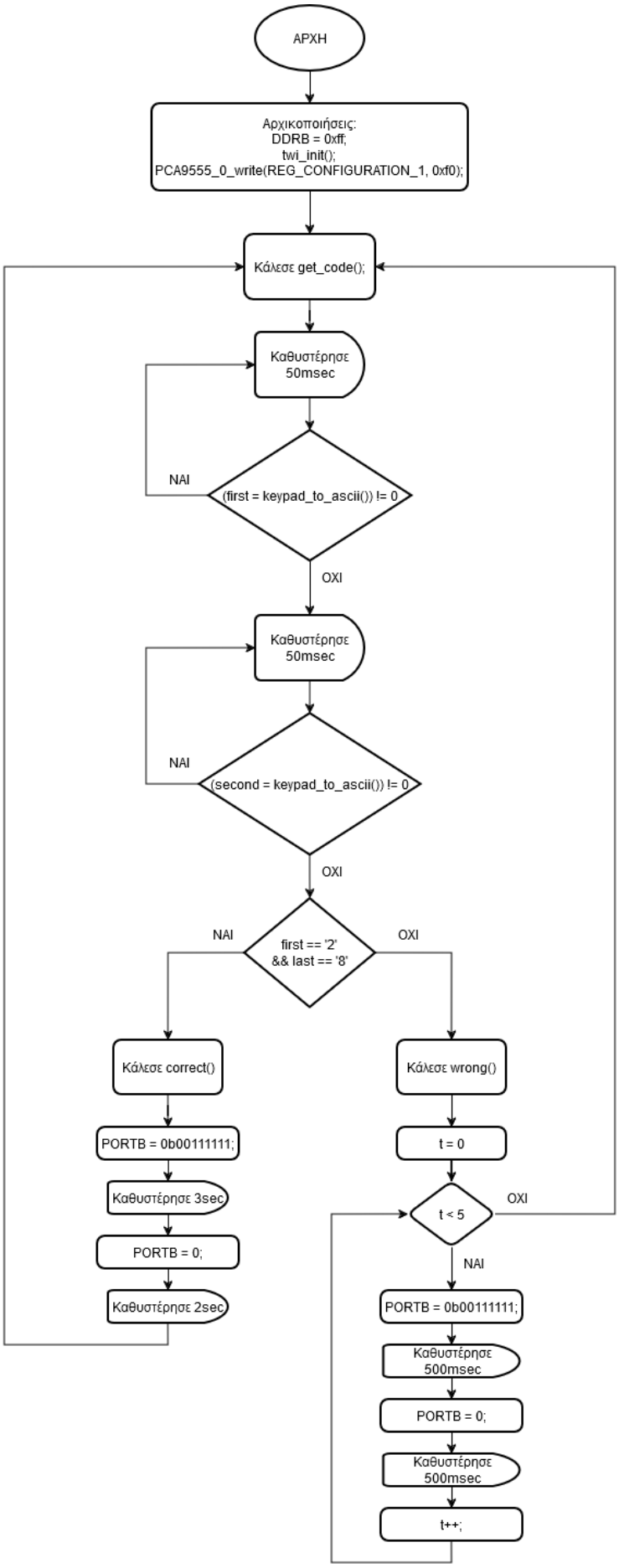
```
328                 lcd_digit(0);
329                 return;
330         }
331         do{
332                 digits[i++] = number%10;
333                 number /= 10;
334         } while(number > 0);
335         for(; i > 0; ) lcd_digit(digits[--i]);
336 }
337
338 void char_to_lcd(){
339         static char last_char = 0;
340         char c = keypad_to_ascii();
341         if(!c || c == last_char) return;
342         last_char = c;
343         lcd_clear_display();
344         if('0' <= c && c <= '9') lcd_digit(c-'0');
345         else if('A' <= c && c <= 'D') lcd_data(c);
346         else if(c == '*') lcd_data(0b00101010);
347         else if(c == '#') lcd_data(0b00100011);
348 }
349
350 int main(void) {
351         DDRB = 0xff; //Set PORTB as output
352         twi_init();
353         PCA9555_0_write(REG_CONFIGURATION_0, 0x00); //Set EXT_PORT0 as output
354         PCA9555_0_write(REG_CONFIGURATION_1, 0xf0); //Set EXT_PORT1 as: 0:3 ->
 ↪  output
355
356         lcd_init(); //uses Port Expander
357         while(1){
358                 _delay_ms(50);
359                 char_to_lcd();
360         }
361 }
```

3 Ζήτημα 6.3

Μεταφέρουμε τον προηγούμενο κώδικα για διάβασμα του πληκτρολογίου και προσθέτουμε συνάρτηση `get_code()` που διαβάζει το πληκτρολόγιο μέχρι να πατηθούν διαδοχικά δύο κουμπιά. Φροντίζουμε να λάβουμε υπόψιν μας τον σπινθηρισμό και τη περίπτωση που ένα κουμπί μένει πατημένο για πολλή ώρα, μέσω του `scan_keypad_rising_edge()` που είχαμε δημιουργήσει. Αν τα κουμπιά με τη σειρά αντιστοιχούν στον αριθμό της ομάδας μας (28), τότε καλούμε την συνάρτηση `correct()` που ανάβει όλα τα LEDs του `PORTB` για 3 sec, και μετά τα σβήνει και περιμένει άλλα 2 sec (ώστε η ρουτίνα να κρατήσει 5 sec συνολικά). Αν τα κουμπιά δεν αντιστοιχούν στον αριθμό της ομάδας μας, καλούμε την συνάρτηση `wrong()` που αναβοσβήνει 5 φορές τα LEDs για συνολικά 5 sec. Με αυτόν

τον τρόπο, το πρόγραμμα δεν δέχεται επόμενη είσοδο από το πληκτρολόγιο για 5 sec μετά από κάθε προσπάθεια εισαγωγής κωδικού. Η συνάρτηση `get_code()` καλείται διαρκώς, οπότε το πρόγραμμα έχει συνεχή λειτουργία. Ακολουθεί το διάγραμμα ροής και ο κώδικας.

```
/*
 * main.c
 *
 * Created: 11/22/2024 10:28:20 AM
 *  Author: User
 */

#include <xc.h>

#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

#define NOP() do { __asm__ __volatile__ ( "nop "); } while (0)

// 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
```



Σχήμα 1: Διάγραμμα Ροής Κώδικα

```
49
50 #define TW_STATUS_MASK 0b11111000
51 #define TW_STATUS (TWSR0 & TW_STATUS_MASK)
52
53 //initialize TWI clock
54 void twi_init(void)
55 {
56         TWSR0 = 0; // PRESCALER_VALUE=1
57         TWBR0 = TWBR0_VALUE; // SCL_CLOCK 100KHz
58 }
59
60 // Read one byte from the twi device (request more data from device)
61 unsigned char twi_readAck(void)
62 {
63         TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
64         while(!(TWCR0 & (1<<TWINT)));
65         return TWDR0;
66 }
67
68 //Read one byte from the twi device, read is followed by a stop condition
69 unsigned char twi_readNak(void)
70 {
71         TWCR0 = (1<<TWINT) | (1<<TWEN);
72         while(!(TWCR0 & (1<<TWINT)));
73         return TWDR0;
74 }
75
76 // Issues a start condition and sends address and transfer direction.
77 // return 0 = device accessible, 1= failed to access device
78 unsigned char twi_start(unsigned char address)
79 {
80         uint8_t twi_status;
81
82         // send START condition
83         TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
84
85         // wait until transmission completed
86         while(!(TWCR0 & (1<<TWINT)));
87
88         // check value of TWI Status Register.
89         twi_status = TW_STATUS & 0xF8;
90         if ( (twi_status != TW_START) && (twi_status != TW_REP_START)) return 1;
91
92         // send device address
93         TWDR0 = address;
94         TWCR0 = (1<<TWINT) | (1<<TWEN);
95
96         // wail until transmission completed and ACK/NACK has been received
97         while(!(TWCR0 & (1<<TWINT)));
98         // check value of TWI Status Register.
99         twi_status = TW_STATUS & 0xF8;
```

```
100         if ( (twi_status != TW_MT_SLA_ACK) && (twi_status != TW_MR_SLA_ACK) )
101         {
102                 return 1;
103         }
104         return 0;
105 }
106
107 // Send start condition, address, transfer direction.
108 // Use ack polling to wait until device is ready
109 void twi_start_wait(unsigned char address)
110 {
111         uint8_t twi_status;
112         while ( 1 )
113         {
114                 // send START condition
115                 TWCR0 = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
116
117                 // wait until transmission completed
118                 while(!(TWCR0 & (1<<TWINT)));
119
120                 // check value of TWI Status Register.
121                 twi_status = TW_STATUS & 0xF8;
122                 if ( (twi_status != TW_START) && (twi_status != TW_REP_START))
                    ↪  continue;
123
124                 // send device address
125                 TWDR0 = address;
126                 TWCR0 = (1<<TWINT) | (1<<TWEN);
127
128                 // wail until transmission completed
129                 while(!(TWCR0 & (1<<TWINT)));
130
131                 // check value of TWI Status Register.
132                 twi_status = TW_STATUS & 0xF8;
133                 if ( (twi_status == TW_MT_SLA_NACK )||(twi_status
                    ↪  ==TW_MR_DATA_NACK) )
134                 {
135                         /* device busy, send stop condition to terminate write
                            ↪  operation */
136                         TWCR0 = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
137
138                         // wait until stop condition is executed and bus released
139                         while(TWCR0 & (1<<TWSTO));
140                         continue;
141                 }
142                 break;
143         }
144 }
145
146 // Send one byte to twi device, Return 0 if write successful or 1 if write failed
147 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));
}

uint8_t LAST;

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();
        LAST = value;
        //if (reg != REG_CONFIGURATION_0) exit(0);
}

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;
}
```

```
199
200 uint8_t scan_row(uint8_t row){ //row = 0, 1, 2, 3
201         uint8_t mask = 0x0f & ~(1<<row);
202         PCA9555_0_write(REG_OUTPUT_1, mask); //enable row as input
203         _delay_us(100);
204         uint8_t in = ~PCA9555_0_read(REG_INPUT_1); //read columns of row pressed
            ↪  in positive logic
205         in >>= 4; //remove IO1[0:3]
206         return in; //4 bits
207 }
208
209 uint16_t scan_keypad(){
210         uint16_t row0 = scan_row(0);
211         uint16_t row1 = scan_row(1);
212         uint16_t row2 = scan_row(2);
213         uint16_t row3 = scan_row(3);
214         return row0 | (row1<<4) | (row2<<8) | (row3<<12);
215 }
216
217 uint16_t scan_keypad_rising_edge(){
218         static uint16_t pressed_keys = 0;
219         uint16_t pressed_keys_tempo = scan_keypad();
220         _delay_ms(15); //wait to avoid triggering
221         pressed_keys_tempo &= scan_keypad(); //only keep the actual buttons
 ↪  pressed
222         uint16_t keys_just_pressed = pressed_keys_tempo & (~pressed_keys);
223         pressed_keys = pressed_keys_tempo;
224         return keys_just_pressed;
225 }
226
227 char keypad_to_ascii(){
228         uint16_t key = scan_keypad_rising_edge();
229         if(key&(1<<0)) return '*';
230         if(key&(1<<1)) return '0';
231         if(key&(1<<2)) return '#';
232         if(key&(1<<3)) return 'D';
233         if(key&(1<<4)) return '7';
234         if(key&(1<<5)) return '8';
235         if(key&(1<<6)) return '9';
236         if(key&(1<<7)) return 'C';
237         if(key&(1<<8)) return '4';
238         if(key&(1<<9)) return '5';
239         if(key&(1<<10)) return '6';
240         if(key&(1<<11)) return 'B';
241         if(key&(1<<12)) return '1';
242         if(key&(1<<13)) return '2';
243         if(key&(1<<14)) return '3';
244         if(key&(1<<15)) return 'A';
245         return 0;
246 }
247
```

```
void char_to_led(){
        char c = keypad_to_ascii();
        switch(c){
                case 'A':
                PORTB = 0x01;
                break;
                case '8':
                PORTB = 0x02;
                break;
                case '6':
                PORTB = 0x04;
                break;
                case '*':
                PORTB = 0x08;
                break;
                default:
                break;
        }
        return;
}

void correct(){
        PORTB = 0b00111111;
        _delay_ms(3000);
        PORTB = 0;
        _delay_ms(2000);
}

void wrong(){
        for(int t = 0; t < 5; t++){
                PORTB = 0b00111111;
                _delay_ms(500);
                PORTB = 0;
                _delay_ms(500);
        }
}

void get_code(){
        char first, second;
        do{
                _delay_ms(50);
        } while(!(first = keypad_to_ascii())); //read first code digit
        do{
                _delay_ms(50);
        } while(!(second = keypad_to_ascii())); //read second code digit
        if(first == '2' && second == '8') correct();
        else wrong();
}

int main(void) {
        DDRB = 0xff; //Set PORTB as output
```

```
299         twi_init();
300         PCA9555_0_write(REG_CONFIGURATION_1, 0xf0); //Set EXT_PORT1 as: 0:3 ->
    ↪ output
301         //                  4:7 -> input
302         while(1){
303                 get_code();
304         }
305 }
```