

GROEIDOCUMENT - MICROCONTROLLERS

NAAM: LUC VAN DER SAR & TEUN LAUWERIJSSEN
STUDENTEN NUMMERS: 2197154 & 2200876 OPLEIDING: TECHNISCHE INFORMATICA
PERIODE: 3
APRIL 2, 2024

Contents

1	Les 1	2
1.1	Opgave A	2
1.1.1	JTAG	2
1.2	Opgave B	2
1.2.1	1	3
1.2.2	2	4
1.2.3	3	5
1.2.4	4	5
1.2.5	5	6
1.2.6	6	7
1.2.7	7	8
2	Week 2	10
2.1	Opgave A	10
2.1.1	1	10
2.1.2	2	10
2.2	Opgave B	14
2.2.1	1	14
2.2.2	2	15
2.2.3	3	16
2.2.4	4	17
2.2.5	5	18
3	Week 3	21
3.1	Opgave A	21
3.2	Opgave B	21
3.2.1	1	21
3.2.2	2	22
4	Week 4	23
4.1	Opdracht B	23
4.1.1	1	23
4.1.2	2	25
4.1.3	3	26
4.1.4	4	27
5	Week 5	30
5.1	Opgave B	30
5.1.1	1	30
5.1.2	2	35

Alle projecten staan in een github repository. : [Microcontrollers-ATmega128](#)

1 Les 1

1.1 Opgave A

Alle Materiale hebben wij opgehaald bij de bali en deze werken naar behoren. de instalatie van de Drivers en Programma's zijn ook allemaal gelukt. Het runnen en debuggen van de simulator is ook goed gegaan. Alles werkt.

1.1.1 JTAG

The screenshot shows the Microsoft Visual Studio IDE interface for a Microchip Atmel ICE (AVR100000662) project. The main window displays assembly code for a file named `mainfile.s`. The assembly code includes definitions for `DDRA`, `DDRB`, `DDRC`, and `DDRD`, and defines a constant `F_CPU` as `8000000UL`. It contains a loop that increments a counter from 0 to 250, with a `delay_us(200)` instruction between each increment. The code uses `PORTA` and `PORTB` registers. The Solution Explorer on the right shows the project structure with files like `Dependencies`, `Output Files`, and `Userns`.

1.2 Opgave B

Tijdens het maken van deze opdrachten hebben wij kennis gemaakt met het ATmega128 bord. In de opdrachten hebben we dan ook voornamelijk ledjes aan moeten sturen en de input van knoppen moeten lezen. Om gebruik te kunnen maken van de knoppen en ledjes was het eerst van belang dat we ze op de juiste stand zette. Hierbij moesten de ledjes op output staan en de knoppen op input. Dit konden we doen door in de register het ledje of knop op 0 of op 1 te zetten, een 0 is hierbij input en 1 is output. Verder hebben we bij de opdrachten 2,3 en 6 gebruik gemaakt van een mask. Met deze mask konden we door middel van XOR makkelijk het juiste ledje laten knipperen.

1.2.1 1

Vragen

- Hoe groot is het program memory van de ATmega128?
- Wat is het adres van Data direction register van PORTE (DDRE)?
- Uit hoeveel byte bestaat de instructie ‘IN R3, PORTA’ ?
- Hoeveel RS232 poorten zitten er op het BIGAVR6/UNI-DS6 development board?
- Op welke pin van de microcontroller zit de ingang voor Analog digitaalconverter, channel 1?
- Hoe groot is het data geheugen van de microcontroller maximaal?
- Hoeveel I/O-registers zijn er op de ATmega128?
- De pinnen van PORTA kunnen met een weerstand naar 0V (pull-down) of met een weerstand naar de +5V verbonden worden (pull-up). Hoe is dat standaard ingesteld op het BIGAVR6/UNI-DS6 development board?

Antwoorden

- Het programmageheugen (flashgeheugen) van de ATmega128 is 128 kilobyte (KB).
- Het adres van het Data Direction Register (DDRE) van de ATmega128 is 0x21.
- De instructie ‘IN R3, PORTA’ bestaat uit 1 byte.
- Het BIGAVR6/UNI-DS6 development board heeft 1 RS232-poort.
- De ingang voor de Analog-to-Digital Converter (ADC), kanaal 1, is meestal verbonden met pin A1 op de microcontroller.
- Het maximale datageheugen (RAM) van de ATmega128 is 4 kilobyte (KB).
- Op de ATmega128 zijn er 87 I/O-registers.
- De standaardinstelling op het BIGAVR6/UNI-DS6 development board is dat de pinnen van PORTA zijn ingesteld als pull-up.

1.2.2 2

Maak een nieuwe applicatie die beurtelings de LED op PORTD, pin 7 (PORTD.7) en de LED op PORTD, pint 6 (PORTD.6) om de 500ms laat oplichten. Ontwikkel de applicatie in de simulator en programmeer daarna het board (gaat veel sneller!)

```
#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>

int main(void)
{
    DDRA = 0xFF;
    DDRB = 0xFF;
    DDRC = 0xFF;
    DDRD = 0xFF;
    signed int counter = 0;
    int flipMask = 0b01100000;
    PORTD = 1 << 5;
    while (1)
    {
        PORTD ^= flipMask;
        _delay_ms(500);
    }
    return 0;
}
```

Om te zorgen dat de juiste leds aan en uit gaan, hebben wij gebruik gemaakt van een mask. Vervolgens hebben we een van de ledjes alvast aangezet. Dit om er voor te zorgen dat tijdens het flippen de waarden tegenovergesteld zijn.

1.2.3 3

Maak een applicatie die de led op PORTD.7 laat knipperen als drukknop PORTC.0 laag (0) is (ingedrukt) en stopt bij het losslaten van de drukknop.

```
#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>

int main(void)
{
    DDRC = 0;
    DDRD = 0xFF;
    int flipMask = 0b10000000;
    while (1)
    {
        if (PINC & 0b00000001)
        {
            PORTD ^= flipMask;
            _delay_ms(500);
        }
        else{
            PORTD = 0;
        }
    }
    return 0;
}
```

[Video](#)

1.2.4 4

Implementeer een looplicht applicatie op de LED's van PORTD. Tussen elke verandering van output zit 50ms (milliseconden). Hoe zou je dit kunnen meten? Om een eenvoudig looplicht te maken kun je gebruik maken van de shift operatoren in C (de » en de «). Dit heb je ook al gedaan in periode TI-1.1 op het GUI board.

```
#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>

int main(void)
{
    DDRD = 0xFF;
    while (1)
    {
        for (int i = 0; i < 8; i++)
        {
            PORTD ^= 1 << i;
            _delay_ms(50);
        }
    }
    return 0;
}
```

[Video](#)

1.2.5 5

Een loopplicht kun je implementeren met een schuifoperatie. Als het gewenste patroon niet zo eenvoudig is kun je e.a. met een grote if-then-else of switch-case constructie implementeren. Dit levert, in het algemeen, slecht onderhoudbare en starre implementaties op. Beter is om een lichtpatroon te sturen vanuit een datastructuur, bijvoorbeeld een C array. Enig idee hoe dit moet? Zie ook het voorbeeld in de code repository. Implementeer een lichteffect met behulp van deze techniek

```
#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>

int main(void)
{
    int animation[] = {0b00000000,
                      0b00011000,0b00111100,
                      0b01111110,0b11111111,
                      0b01111111,0b00111111,
                      0b00011111,0b00000111,
                      0b00000011,0b00000011,
                      0b00000001,0b00000001,
                      0b00000011,0b00000111,
                      0b00011111,0b00111111,
                      0b01111111,0b11111111,
                      0b11111110,0b11111100,
                      0b11111000,0b11110000,
                      0b11100000,0b11000000,
                      0b10000000,0b10000000,
                      0b11100000,0b11110000,
                      0b11111000,0b11111100,
                      0b11111110,0b11111111,
                      0b01111110,0b00111100,
                      0b00011000,0b00000000
    };
    DDRD = 0xFF;
    while (1)
    {
        for (int i = 0; i < sizeof(animation); i++)
        {

            PORTD = animation[i];
            _delay_ms(150);
        }
    }
    return 0;
}
```

[Video](#)

1.2.6 6

Toestanden. Maak een applicatie die de led op PORTD.7 laat knipperen met een frequentie van circa 1Hz (1 keer per seconde). Als nu PORTC.0 kort wordt ingedrukt gaat (en blijft) de led sneller knipperen (bijvoorbeeld 4Hz). Bij nogmaals kort drukken gaat (en blijft) de led weer knipperen met een frequentie van 1Hz.

```
#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>

void variableDelay(int delay);

int main(void)
{
    int flipmask = 0b00000001;
    int delay= 1000;

    DDRD = 0xFF;
    DDRC = 0b11111110;
    while (1)
    {

        if (PINC & 0b00000001)
        {
            if (delay == 1000)
            {
                delay = 250;
            }
            else
            {
                delay = 1000;
            }
        }
        PORTD ^= flipmask;
        variableDelay(delay);
    }
    return 0;
}

void variableDelay(int delay)
{
    if(delay == 250){
        _delay_ms(250);
    }
    if(delay == 1000){
        _delay_ms(1000);
    }
}
```

[Video](#)

1.2.7 7

Implementeer onderstaande ‘eindige toestandsmachine’ (eng: finite state machine of fsm) . Een fsm is de basis van bijna elke embedded applicatie. Koffiemachines, televisies, pacemakers, ABS computers, alarmsystemen enz. zijn voorbeelden van applicaties waar de main-loop vaak bestaat uit een (ingewikkelde) eindige toestandsmachine.

Een fsm machine kan uitgroeien tot iets ingewikkelds en het is van belang dat software engineers (jij!) grip hebben op een correcte werking. Denk ook aan uitbreidbaarheid en onderhoudbaarheid. Interessant is ook het artikel op Blackboard over het aantal regels code in alledaagse apparaten.

Hoe zou je een fsm implementeren? Met een hele uitgebreide switch/case of if/then loop? Probeer het maar!

```
#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>

#define button(number) (1 << number)

int main(void)
{
    DDRA = 0xFF;
    DDRB = 0xFF;
    DDRD = 0x00;

    DDRC = 0xFF;
    int state = 0;
    while (1)
    {
        _delay_ms(500);
        switch(state){
            case 0:
                PORTA=0xFF;
                PORTB=0xFF;
                PORTC=0xFF;
                if(PIND & button(6)){
                    state = 1;
                }
                if (PIND & button(5))
                {
                    state = 2;
                }
                break;
            case 1:
                PORTA=0x00;
                PORTB=0xFF;
                PORTC=0xFF;
                if (PIND & button(5))
                {
                    state = 2;
                }
                break;
            case 2:
                PORTA=0x00;
                PORTB=0x00;
                PORTC=0xFF;
                if (PIND & button(5))
                {
                    state = 3;
                }
        }
    }
}
```

```
        }
        break;
    case 3:
        PORTA=0x00;
        PORTB=0x00;
        PORTC=0x00;
        if (PIND & button(5)||PIND & button(6))
        {
            state = 4;
        }
        break;
    default:
        PORTA=0xF0;
        PORTB=0xF0;
        PORTC=0xF0;
        if (PIND & button(7))
        {
            state = 0;
        }
        break;
    }
}
return 0;
}
```

[Video](#)

2 Week 2

2.1 Opgave A

2.1.1 1

The good:

Interrupts zijn goed te gebruiken in situaties waarbij timing en consistentie belangrijk zijn. Dit is vooral het geval bij het communiceren met andere apparaten.

The bad:

Als interrupts te lang duren kan dit zorgen voor starvation bij de main loop. Dit komt doordat de main loop dan moet wachten tot de interrupt klaar is. Dit kan verholpen worden door enkele variabelen bereikbaar te maken in de ISR en main loop. Op deze manier kan de ISR aangeven wanneer het bijbehorende proces afgehandeld moet worden. Dit kan de main loop vervolgens uitvoeren wanneer nodig. Ook kan er starvation optreden als de er te vaak interrupts worden aangeroepen.

The Ugly:

Op het moment dat er variabelen worden aangepast vanuit de interrupt kan er niet meer vanuit gegaan worden dat ze constant blijven. Dit omdat je niet weet op welke momenten de interrupt aangeroepen wordt. Om dit op te lossen kunnen de interrupts tijdelijk uitgezet worden op het moment dat er een belangrijk stuk code uitgevoerd moet worden.

2.1.2 2

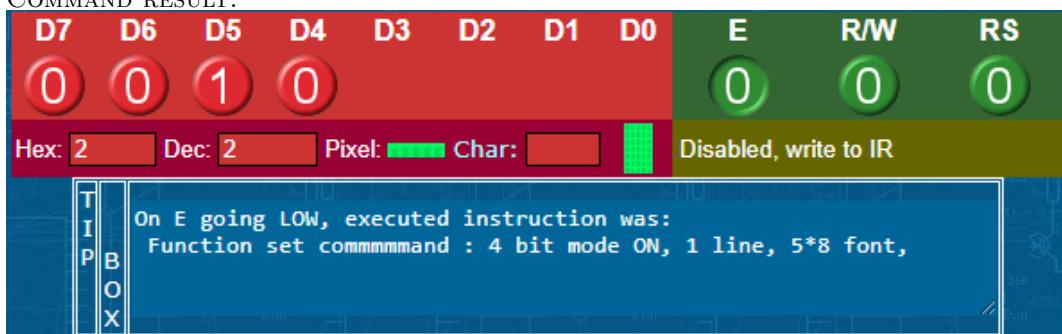
Onderdeel 1:

om het LCD in 4-bits mode te krijgen moet je een function-set commando sturen. Om dit commando te kunnen sturen moet je data-bit 5 de waarde 1 geven. Vervolgens geeft data-bit 4(DL) aan op welke bit-mode het LCD komt te staan. Hierbij is een waarde van 0 de 4-bits mode en een waarde van 1 is 8-bits mode. We hebben 4-bits nodig dus deze data-bit laten we op 0. Verder kan er met data-bit 3(N) aangegeven worden of het LCD gebruik maakt van 2 regels of 1 regel. Als data-bit 3(F) op 1 staat worden er 2 regels gebruikt, bij een waarde van 0 maar 1 regel. Tot slot kan er met data-bit 2 de resolutie van de karakters aangegeven worden. Er kan hier gekozen worden tussen 5x10 dots en 5x8 dots. Op het moment dat er een 1 wordt meegegeven wordt er gebruik gemaakt van 5x10 dots, in het geval van 0 zal dit 5x8 dots zijn.

COMMAND INPUT:



COMMAND RESULT:



COMMAND TABLE:

Table 6 Instructions

Instruction	Code										Description	Execution Time (max) (when f_{cp} or f_{osc} is 270 kHz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs						
Read busy flag & address	0	1	BF	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs						

COMMAND TABLE-VALUE DESCRIPTION:

Table 6 Instructions (cont)

Instruction	Code										Description	Execution Time (max) (when f_{cp} or f_{osc} is 270 kHz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs $t_{ADD} = 4 \mu s^*$
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs $t_{ADD} = 4 \mu s^*$
I/D = 1: Increment I/D = 0: Decrement S = 1: Accompanies display shift S/C = 1: Display shift S/C = 0: Cursor move R/L = 1: Shift to the right R/L = 0: Shift to the left DL = 1: 8 bits, DL = 0: 4 bits N = 1: 2 lines, N = 0: 1 line F = 1: 5 × 10 dots, F = 0: 5 × 8 dots BF = 1: Internally operating BF = 0: Instructions acceptable											DDRAM: Display data RAM CGRAM: Character generator RAM ACG: CGRAM address ADD: DDRAM address (corresponds to cursor address) AC: Address counter used for both DD and CGRAM addresses	Execution time changes when frequency changes Example: When f_{cp} or f_{osc} is 250 kHz, $37 \mu s \times \frac{270}{250} = 40 \mu s$

HEX-VALUE COMMAND TABLE:

INSTRUCTION	Decimal	Hexadecimal
Function set (8-bit interface, 2 lines, 5*7 Pixels)	56	38
Function set (8-bit interface, 1 line, 5*7 Pixels)	48	30
Function set (4-bit interface, 2 lines, 5*7 Pixels)	40	28
Function set (4-bit interface, 1 line, 5*7 Pixels)	32	20
Entry mode set	See Below	See Below
Scroll display one character right (all lines)	28	1E
Scroll display one character left (all lines)	24	18
Home (move cursor to top/left character position)	2	2
Move cursor one character left	16	10
Move cursor one character right	20	14
Turn on visible underline cursor	14	0E
Turn on visible blinking-block cursor	15	0F
Make cursor invisible	12	0C
Blank the display (without clearing)	8	08
Restore the display (with cursor hidden)	12	0C
Clear Screen	1	01
Set cursor position (DDRAM address)	128 + addr	80+ addr
Set pointer in character-generator RAM (CG RAM address)	64 + addr	40+ addr
Read DDRAM/CGRAM & Check Busy Flag	See Below	See Below

Onderdeel 2:

Uitleg pinnen:

- **D0-D7:** Dit zijn de data pinnen. Met deze pinnen wordt aangegeven welk commando er uit gevoerd moet worden of welke data er gelezen/geschreven moet worden vanuit/naar de lcd.
- **E:** dit is de "enable" pin. Deze pin registreert een falling edge. Op het moment dat deze pin een falling edge registreert weet de lcd dat er een nieuw commando/nieuwe data is binnengekomen. Hierop kan de lcd zich vervolgens updaten
- **RS:** Hiermee wordt er aangegeven of er een commando of data verstuurd word. Bij waarde 0 = commando, bij waarde 1 = data.

Onderdeel 3:

```
void lcd_init(void)
{
    DDRC = 0xFF;
    PORTC = 0x00;

    // Step 2 (table 12), set interface data-length, and amount of lines
    lcd_write_cmd(0x28);

    // Step 3, clear display
    lcd_write_cmd(0x01);

    // Step 4 (table 12), turn on display
    lcd_write_cmd(0x0C);

    // Set cursor
    lcd_write_cmd(0x0F);
}

void lcd_write_data(unsigned char byte) {
    // First nibble.
    PORTC = byte & 0xf0;
    PORTC |= (1<<LCD_RS);
    lcd_strobe_lcd_e();

    // Second nibble
    PORTC = (byte<<4);
    PORTC |= (1<<LCD_RS);
    lcd_strobe_lcd_e();
}
```

Onderdeel 4:

```
void lcd_move_text(int position, char *str){
    lcd_write_cmd(0x01);
    lcd_set_cursor(position);
    lcd_display_text(str);
}
```

2.2 Opgave B

2.2.1 1

```

#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

void wait( int ms ) {
    for (int i=0; i<ms; i++) {
        _delay_ms( 1 );                                // library function (max 30 ms at 8MHz)
    }
}

ISR( INT0_vect ) {
    PORTD |= (1<<5);
}

ISR( INT1_vect ) {
    PORTD &= ~(1<<5);
}

int main( void ) {
    // Init I/O
    DDRD = 0xF0;                                     // PORTD(7:4) output, PORTD(3:0) input

    // Init Interrupt hardware
    EICRA |= 0x0B;                                    // INT1 falling edge, INT0 rising edge
    EIMSK |= 0x03;                                    // Enable INT1 & INT0

    // Enable global interrupt system
    //SREG = 0x80;                                     // Of direct via SREG or via wrapper
    sei();

    while (1) {
        PORTD ^= (1<<7);                            // Toggle PORTD.7
        wait( 500 );
    }
}

return 1;
}

```

Je set eerst de Interupts falling edge op INT1 dat is 10 en daarna Rising edge op INT0 11. Falling edge van INTn genereert een asynchrone interupt request. Dit geld ook voor de rising edge. Daarna stel je in welke pin INT1 & INT0 aan gekoppeld wordt. Daarna zet je het interup system aan met sei(); dit kan je ook weer uitzetten met cei(); Vervolgens laat je een LED knipperen op D7 om het interupt systeem te laten zien.

2.2.2 2

```

#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
void Next_Light_State();

void wait( int ms ) {
    for (int i=0; i<ms; i++) {
        _delay_ms( 1 );                                // library function (max 30 ms at 8MHz)
    }
}

ISR( INT1_vect ) {
    Next_Light_State();
}

ISR( INT2_vect ) {
    Next_Light_State();
}

int main( void ) {
    // Init I/O
    DDRD = 0xFO;                                     // PORTD(7:4) output, PORTD(3:0) input
    DDRA = 0xFF;
    // Init Interrupt hardware
    EICRA |= 0x2C;                                    // INT2 falling edge, INT1 rising edge
    EIMSK |= 0x06;                                    // Enable INT1 & INT2

    sei();

    while (1) {

    }

    return 1;
}
void Next_Light_State(){
    static int state = 0;

    if(state <8)
    {
        PORTA ^= 1 << state;
        state++;
        _delay_ms(10);
    }else
    {
        state = 0;
    }
}

```

Wat je hier technisch doet is je verschuift de INT1 & INT0 naar INT1 & INT2 dit doe je door in plaats van 1011, 101100 mee te geven. Vervolgens moet je ook in de methodes de juiste pins mee geven zoals INT1_VEC en INT2_VEC

[Video](#)

2.2.3 3

```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

#define BIT(x)(1 << (x))

int main( void ) {
    DDRB = 0xFF;
    int numbers[16] = {
        0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07,
        0x7F, 0x6F, 0x77, 0x7C, 0x39, 0x5E, 0x79, ~0x8E
    };
    int error = 0xF9;
    int index = 0;
    while (1) {
        if (PIND & BIT(0))
        {
            index++;
        }
        if (PIND & BIT(1))
        {
            index--;
        }
        if (PIND & BIT(0) && PIND & BIT(1))
        {
            index = 0;
        }

        if (index>15)
        {
            PORTB = error;
        }else
        {
            PORTB = numbers[index];
        }
        _delay_ms(100);
    }

    return 1;
}
```

Om te zorgen dat de 7 segment display de juiste tekens weergeeft hebben we gebruik gemaakt van een array. In deze array staan alle bitwaardes voor de juiste symbolen. Vervolgens hebben we een variabele aangemaakt die de index bijhoudt. Op deze manier kunnen we bij houden op welk getal we momenteel zitten. Zo kunnen we er ook bij optellen en aftrekken. Door middel van if-statements kunnen we dan ook optellen en aftrekken op het moment de juiste knop wordt ingedrukt. In het geval beide knoppen worden ingedrukt reset de 7 segment display naar 0. Verder wordt er ook een E van error getoond op het moment dat de waarde hoger is dan 15.

[Video](#)

2.2.4 4

```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

#define BIT(x)(1 << (x))

void wait(int delay){
    for(int i = 0; i < delay; i++){
        _delay_ms(1);
    }
}

typedef struct{
    unsigned char data;
    unsigned int delay;
}PATTERN_STRUCT;

PATTERN_STRUCT LightPattern[] = { {0x01, 100}, {0x02, 100}, {0x04, 100}, {0x08, 100}, {0x10, 100}, {0x20,
{0x01, 50}, {0x03, 50}, {0x07, 50}, {0x0F, 50}, {0x1F, 50}, {0x3F, 50}, {0x7F, 50},
{0x7F, 300}, {0x7E, 100}, {0x7C, 500}, {0x78, 100}, {0x70, 500}, {0x60, 100}, {0x40, 500}, {0x00, 100},
{0x00,0x00}};

int main( void ) {
    DDRB = 0xFF;

    while (1)
    {
        int index = 0;

        while(LightPattern[index].delay != 0){
            PORTB = LightPattern[index].data;
            wait(LightPattern[index].delay);
            index++;
        }
    }

    return 1;
}
```

In deze code hebben we een Animatie gemaakt op het segmenten display. Ook hier hebben we gebruik gemaakt van een array, alleen in deze array staat ook een delay door middel van een struct. Op deze manier kunnen we een variabele delay hebben.

[Video](#)

2.2.5 5

header.h

```
#ifndef LCD_H
#define LCD_H

void lcd_init(void);
void lcd_display_text(char *str);
void lcd_set_cursor(int position);

#endif
```

main.c

```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "lcdController.h"

void setup(){

}

int main(){
    lcd_init();
    lcd_display_text("hey");
    lcd_set_cursor(0);
    while(1){
        for (int i = 0; i<32; i++)
        {
            lcd_set_cursor(i);
            _delay_ms(250);
            lcd_display_text("hey");
            _delay_ms(250);
        }
    }
    return 0;
}
```

lcdController.c

```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "lcdController.h"

#define LCD_E      3
#define LCD_RS     2

void lcd_strobe_lcd_e(void);
void lcd_write_data(unsigned char byte);
void lcd_write_cmd(unsigned char byte);

void lcd_init(void)
{
    DDRC = 0xFF;
    PORTC = 0x00;

    // Step 2 (table 12), set interface data-length, and amount of lines
    lcd_write_cmd(0x28);

    // Step 3, clear display
    lcd_write_cmd(0x01);

    // Step 4 (table 12), turn on display
    lcd_write_cmd(0x0C);

    // Set cursor
    lcd_write_cmd(0x0F);
}

void lcd_display_text(char *str)
{
    for(;*str; str++){
        lcd_write_data(*str);
    }
}

void lcd_set_cursor(int position)
{
    lcd_write_cmd(0x80 + position);
}

void lcd_strobe_lcd_e(void) {
    PORTC |= (1<<LCD_E);           // E high
    _delay_ms(1);                  // nodig
    PORTC &= ~(1<<LCD_E);        // E low
    _delay_ms(1);                  // nodig?
}
```

```
void lcd_write_cmd(unsigned char byte) {
    // First nibble.
    PORTC = byte & 0xf0;
    lcd_strobe_lcd_e();

    // Second nibble
    PORTC = (byte<<4);
    lcd_strobe_lcd_e();
}

void lcd_write_data(unsigned char byte) {
    // First nibble.
    PORTC = byte & 0xf0;
    PORTC |= (1<<LCD_RS);
    lcd_strobe_lcd_e();

    // Second nibble
    PORTC = (byte<<4);
    PORTC |= (1<<LCD_RS);
    lcd_strobe_lcd_e();
}
```

3 Week 3

3.1 Opgave A

N.V.M

3.2 Opgave B

3.2.1 1

```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include "lcdController.h"

int main(void)
{
    DDRA = 0xFF;
    PORTA = 0x00;

    TCCR2 = 0b00000111; // configure normal mode for the TCCR2 with external clock
    TCNT2 = 0x00;

    lcd_init();
    char buffer[20]; // Buffer to hold the converted string
    for (;;)
    {
        PORTA = TCNT2;
        lcd_write_cmd(0x01);
        sprintf(buffer, "%d", TCNT2); // Convert integer to string

        lcd_display_text(buffer);
        _delay_ms(100);
    }
    return 0;
}
```

In deze code maakt de Timer gebruik van T2 dat is Knop D7 als counter in plaats van een mhz internal counter. Vervolgens schrijven we dat in een LCD display om te visualiseren hoe hoog de counter is. Je zou dit ook kunnen realiseren met een IR sensor. [Video](#)

3.2.2 2

```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdio.h>

ISR(TIMER2_COMP_vect)
{
    if (OCR2 == 25)
    {
        PORTD |= (1<<7);
        OCR2 = 15;
    }
    else
    {
        PORTD &= ~(1<<7);
        OCR2 = 25;
    }
}

int main(void)
{
    TCCR2 = 0b00011100; // configure normal mode for the TCCR2 with external clock
    TCNT2 = 0x00;
    OCR2 = 25;
    DDRD = 0xFF;
    TIMSK |= (1<<7);
    sei();

    for (;;)
    {
    }
    return 0;
}
```

Als je de prescaler op 1024 zet in plaats van 256 dan wordt de frequentie van het signaal lager
 De frequentie veranderd dan van 774hz naar 194hz

Met een prescaler van 1024 krijg je een tijd interval van 0,129ms

Vervolgens is de waarde om High 15 ms te tellen stel je hem in op $116\frac{12}{43}$

In deze code maken wij een hoog signaal dat loopt op 15ms en een laag signaal dat loopt op 25ms. Hierbij hebben wij gebruik gemaakt van de Compare timer mode. [Video](#)

4 Week 4

IN SAMENWERKING MET NIELS BUIJS, I.V.M AFWEZIGHEID VAN LUC

4.1 Opdracht B

4.1.1 1

De prefactor hebben we ingesteld door de laatste 3 bits in de ADCSRA register aan te passen. In ons geval hadden we een deelfactor nodig van 64. Om dit te kunnen bereiken hebben we bit 2 en 1 op 1 gezet en bit 0 op 0. om de referentiespanning in te stellen hebben we in de ADMUX bit 7 en 6 aan moeten passen. In ons geval was dit bit 6 op 1 zetten. Vervolgens hebben we met de ADLAR ingesteld of de waardes werden opgeslagen "met hoge byte gevuld" of "met lage byte gevuld". Wij hadden met hoge byte gevuld nodig, hiervoor hebben we de ADLAR op 1 moeten zetten.

Bit	7	6	5	4	3	2	1	0	
ReadWrite	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	ADMUX
Initial Value	0	0	0	0	0	0	0	0	

Free running: OP het moment dat free running aangezet wordt, worden de data registers continu geüpdatet. Dit kan gedaan worden door in de ADCSRA register bit 5 aan te zetten.

Bit	7	6	5	4	3	2	1	0	
ReadWrite	R/W	ADCSRA							
Initial Value	0	0	0	0	0	0	0	0	

```

#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

#define BIT(x)      (1 << (x))

// wait(): busy waiting for 'ms' millisecond
// Used library: util/delay.h
void wait( int ms )
{
    for (int tms=0; tms<ms; tms++)
    {
        _delay_ms( 1 );           // library function (max 30 ms at 8MHz)
    }
}

// Initialize ADC: 10-bits (left justified), free running
void adcInit( void )
{
    ADMUX = 0b01100001;          // AREF=VCC, result left adjusted, channel1 at pin PF1, ADLAR = 1
    ADCSRA = 0b11100110;          // ADC-enable, no interrupt, start, free running, division by 64
}

// Main program: ADC at PF1
int main( void )
{
    DDRF = 0x00;                // set PORTF for input (ADC)
}

```

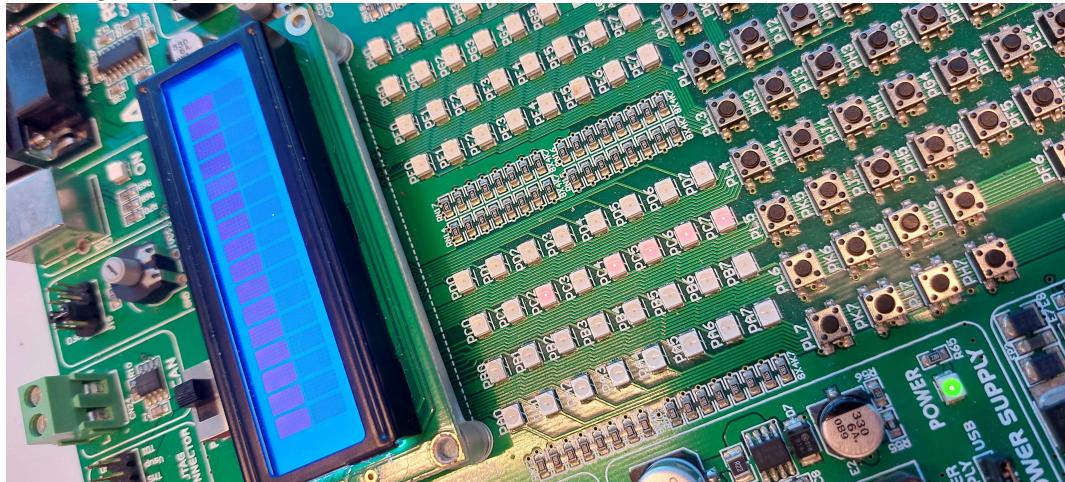
```
DDRA = 0xFF;           // set PORTA for output
DDRB = 0xFF;           // set PORTB for output
adcInit();             // initialize ADC

while (1)
{
    PORTB = ADCL;      // Show MSB/LSB (bit 10:0) of ADC
    PORTA = ADCH;
    wait(100);          // every 100 ms (busy waiting)
}
}
```

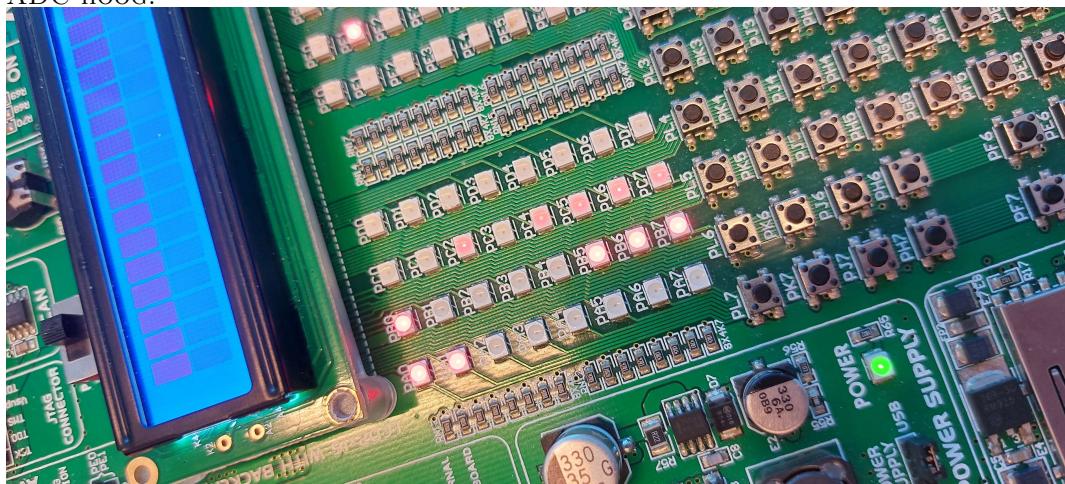
4.1.2 2

We hebben bij deze opdracht de ADC op kanaal 1 gezet. De waarden uit de ADMUX register hebben we vervolgens weergegeven op de ledjes. Hierbij is bit 0 led PB1 en bit 9 led PA1. Hieronder is te zien dat op het moment dat de ADC laag is dat alle ledjes uit zijn en op het moment dat de ADC hoog is staan deze aan.

ADC LAAG:



ADC HOOG:



4.1.3 3

De ADC laat nu alleen zijn waardes zien op de ledjes op het moment dat knop PB0 ingedrukt wordt. Dit hebben we gedaan door gebruik te maken van een if statement om te kijken of deze wordt ingedrukt. Op het moment dat dit het geval is zetten we in de ADCSRA bit 6 op 1. Dit geeft aan dat de ADC zijn data register moet updaten.

[Video](#)

```

void adcInit( void )
{
    ADMUX = 0b01100011;           // AREF=VCC, result left adjusted, channel1 at pin PF1, ADLAR = 1
    ADCSRA = 0b10000110;         // ADC-enable, no interrupt, not start, free running off, division by 64
}

// Main program: ADC at PF1
int main( void )
{
    DDRF = 0x00;                // set PORTF for input (ADC)
    DDRA = 0xFF;                // set PORTA for output
    DDRB = 0x00;                // set PORTB for input
    adcInit();                  // initialize ADC

    while (1)
    {
        // ADC Start Conversion
        if (PINB & BIT(0)){
            ADCSRA |= (1<<ADSC);
        }

        // Wait for conversion to complete
        while (ADCSRA & (1<<ADSC));

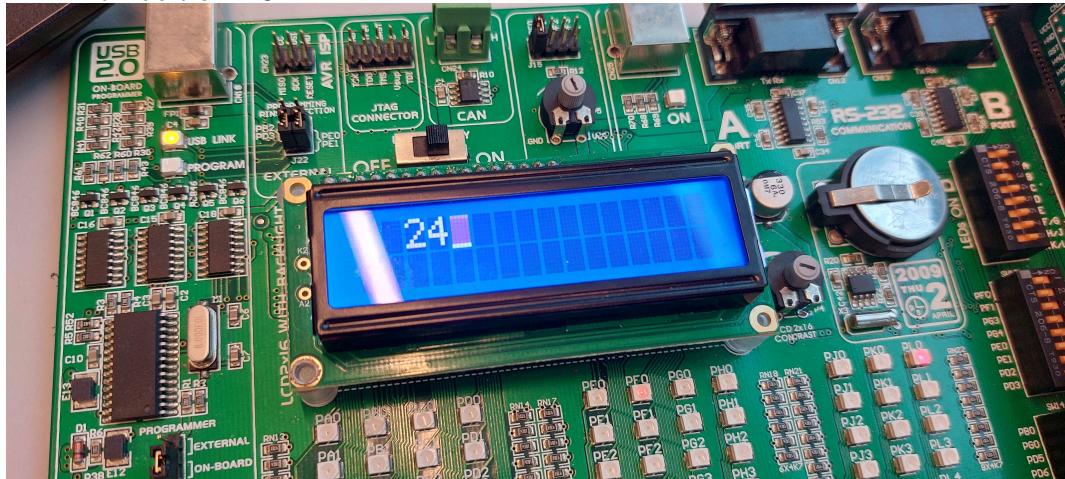
        //PORTA = ADCL;
        PORTA = ADCH;
    }
}

```

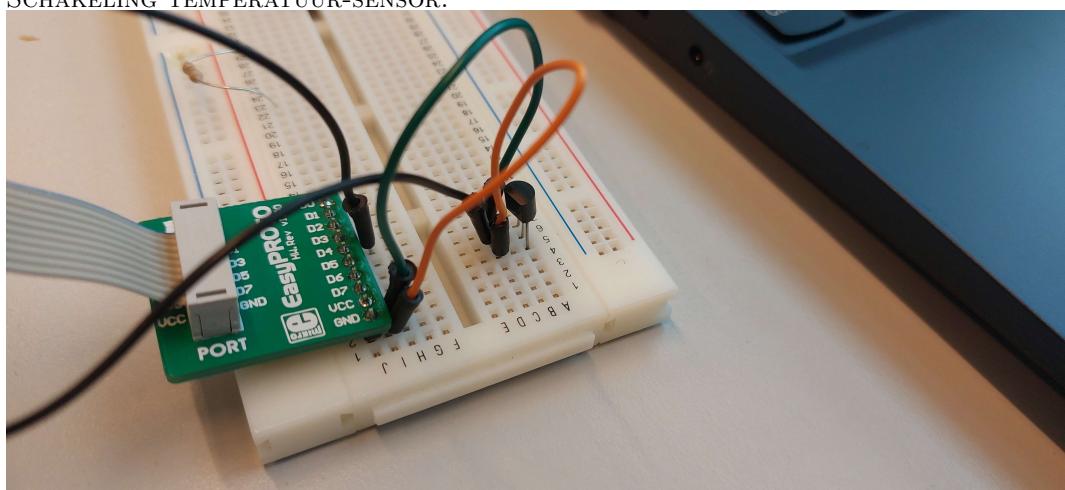
4.1.4 4

We hebben de temperatuur-sensor aangesloten op de ATmega128. hiervoor hebben we de op de ADC een referentiespanning van 2.56V gezet. Dit hebben we gedaan door in de ADMUX bit 7 en 6 op 1 te zetten. Hierdoor konden we de waarde van de sensor uitlezen. Door vervolgens te kijken naar bit 9-2 in de ADCH register hebben we waardes uit kunnen lezen. Deze waardes hebben we vervolgens op het lcd getoond.

TEMPERATUUR OP LCD:



SCHAKELING TEMPERATUUR-SENSOR:



```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "main.h"

#define BIT(x) (1 << (x))
```

```

// wait(): busy waiting for 'ms' millisecond
// Used library: util/delay.h
void wait( int ms )
{
    for (int tms=0; tms<ms; tms++)
    {
        _delay_ms( 1 );                                // library function (max 30 ms at 8MHz)
    }
}

// Initialize ADC: 10-bits (left justified), free running
void adcInit( void )
{
    ADMUX = Ob11100011;                            // AREF=VCC, result left adjusted, channel1 at pin PF1,
    ADCSRA = Ob10000110;                            // ADC-enable, no interrupt, not start, free running off, divi
}

// Main program: ADC at PF1
int main( void )
{
    DDRF = 0x00;                                    // set PORTF for input (ADC)
    DDRA = 0xFF;                                    // set PORTA for output
    DDRB = 0x00;                                    // set PORTB for input
    adcInit();                                     // initialize ADC

    DDRD = 0xFF;

    init();

    clear_lcd();

    set_cursor(42);
    display_text("brood");

    char buffer[20];

    while (1)
    {
        // ADC Start Conversion
        //if (PINB & BIT(0)){
        clear_lcd();
        ADCSRA |= (1<<ADSC);
        sprintf(buffer, "%d", ADCH);
        set_cursor(2);
        display_text(buffer);
        wait(75);

        //}

        // Wait for conversion to complete
        while (ADCSRA & (1<<ADSC));
    }
}

```

```
//PORTA = ADCL;  
PORTA = ADCH;  
}  
  
}
```

5 Week 5

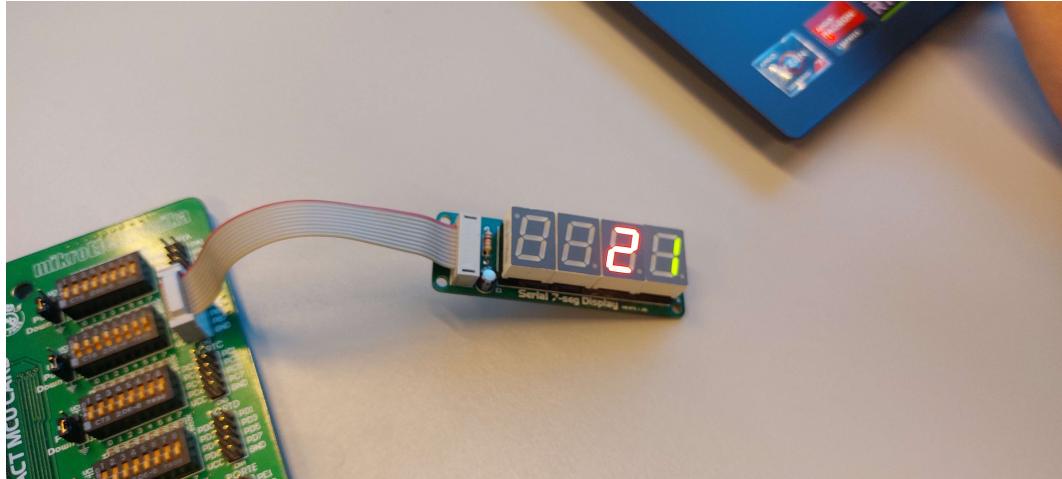
IN SAMENWERKING MET NIELS BUIJS, I.V.M AFWEZIGHEID VAN LUC

5.1 Opgave B

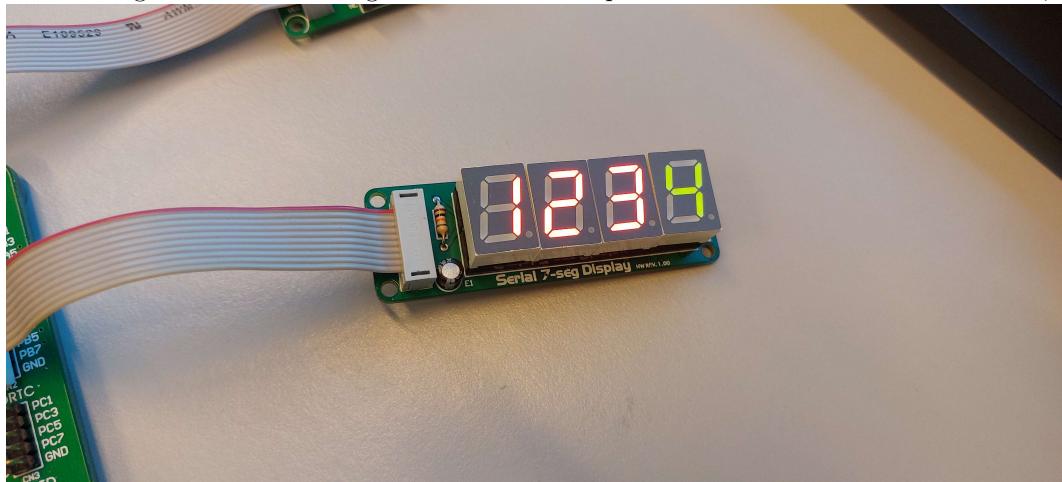
5.1.1 1

Om te testen of de communicatie tussen de ATMega128 en het 4 digit display werkt hebben we gebruik gemaakt van de demo-code. De code zou er dan voor moeten zorgen dat de achterste 2 digits worden aangestuurd. Na het runnen hiervan blijkt dat dit ook werkt.

TWEE SEGMENTS AAN:



Vervolgens was het de bedoeling om alle vier de segmenten aan te krijgen. Dit hebben we gedaan door de Scanlimit hoger te zetten. In dit geval was het op 4. Verder moesten we ook de licht intensiteit op het hoogst zetten. dit hebben we gedaan door in de register de intensiteit op "F" te zetten. VIER SEGMENTS AAN, HOOGSTE INTENSITEIT:



```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>

#define BIT(x)          ( 1<<x )
#define DDR_SPI         DDRB           // spi Data direction register
#define PORT_SPI        PORTB          // spi Output register
#define SPI_SCK          1             // PB1: spi Pin System Clock
#define SPI_MOSI         2             // PB2: spi Pin MOSI
#define SPI_MISO         3             // PB3: spi Pin MISO
#define SPI_SS           0             // PB0: spi Pin Slave Select
```

```

// wait(): busy waiting for 'ms' millisecond
// used library: util/delay.h
void wait(int ms)
{
    for (int i=0; i<ms; i++)
    {
        _delay_ms(1);
    }
}

void spi_masterInit(void)
{
    DDR_SPI = 0xff;
    DDR_SPI &= ~BIT(SPI_MISO);                                // All pins output: MOSI, SCK, SS, SS_display as ou
    PORT_SPI |= BIT(SPI_SS);                                 // except: MISO input
    SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR1);           // SS_ADC == 1: deselect slave
                                                               // or: SPCR = 0b11010010;                                // or: SPCR = 0b11010010;
                                                               // Enable spi, Mas
                                                               // Mode = 0: CPOL=0
}

// Write a byte from master to slave
void spi_write( unsigned char data )
{
    SPDR = data;                                            // Load byte to Data register
    while( !(SPSR & BIT(SPIF)) );                         // Wait for transmission complete
}

// Write a byte from master to slave and read a byte from slave
// nice to have; not used here
char spi_writeRead( unsigned char data )
{
    SPDR = data;                                            // Load byte to Data register
    while( !(SPSR & BIT(SPIF)) );                         // Wait for transmission complete
    data = SPDR;                                             // New received data (eventual
    return data;                                              // Return received byte
}

// Select device on pinnummer PORTB
void spi_slaveSelect(unsigned char chipNumber)
{
    PORTB &= ~BIT(chipNumber);
}

// Deselect device on pinnummer PORTB
void spi_slaveDeSelect(unsigned char chipNumber)
{
    PORTB |= BIT(chipNumber);
}

// Initialize the driver chip (type MAX 7219)
void displayDriverInit()
{
}

```

```

    spi_slaveSelect(0);                                // Select display chip (MAX7219)
    spi_write(0x09);                                  // Register 09: Decode Mode
    spi_write(0xFF);                                  //           -> 1's = BCD mode for all digits
    spi_slaveDeSelect(0);                            // Deselect display chip

    spi_slaveSelect(0);                                // Select dispaly chip
    spi_write(0x0A);                                  // Register 0A: Intensity
    spi_write(0x0F);                                  //   -> Level 4 (in range [1..F])
    spi_slaveDeSelect(0);                            // Deselect display chip

    spi_slaveSelect(0);                                // Select display chip
    spi_write(0x0B);                                  // Register 0B: Scan-limit
    spi_write(0x04);                                  //           -> 1 = Display digits 0..1
    spi_slaveDeSelect(0);                            // Deselect display chip

    spi_slaveSelect(0);                                // Select display chip
    spi_write(0x0C);                                  // Register 0B: Shutdown register
    spi_write(0x01);                                  //           -> 1 = Normal operation
    spi_slaveDeSelect(0);                            // Deselect display chip
}

// Set display on ('normal operation')
void displayOn()
{
    spi_slaveSelect(0);                                // Select display chip
    spi_write(0x0C);                                  // Register 0B: Shutdown register
    spi_write(0x01);                                  //           -> 1 = Normal operation
    spi_slaveDeSelect(0);                            // Deselect display chip
}

// Set display off ('shut down')
void displayOff()
{
    spi_slaveSelect(0);                                // Select display chip
    spi_write(0x0C);                                  // Register 0B: Shutdown register
    spi_write(0x00);                                  //           -> 1 = Normal operation
    spi_slaveDeSelect(0);                            // Deselect display chip
}

int main()
{
    // initialize
    DDRB=0x01;                                         // Set PBO pin as output for display select
    spi_masterInit();                                 // Initialize spi module
    displayDriverInit();                             // Initialize display chip

    // clear display (all zero's)
    for (char i =1; i<=4; i++)
    {
        spi_slaveSelect(0);                            // Select display chip
        spi_write(i);                                //           digit adress: (digit place)
        spi_write(0);                                //           digit value: 0
        spi_slaveDeSelect(0);                          // Deselect display chip
    }
    wait(1000);
}

```

```

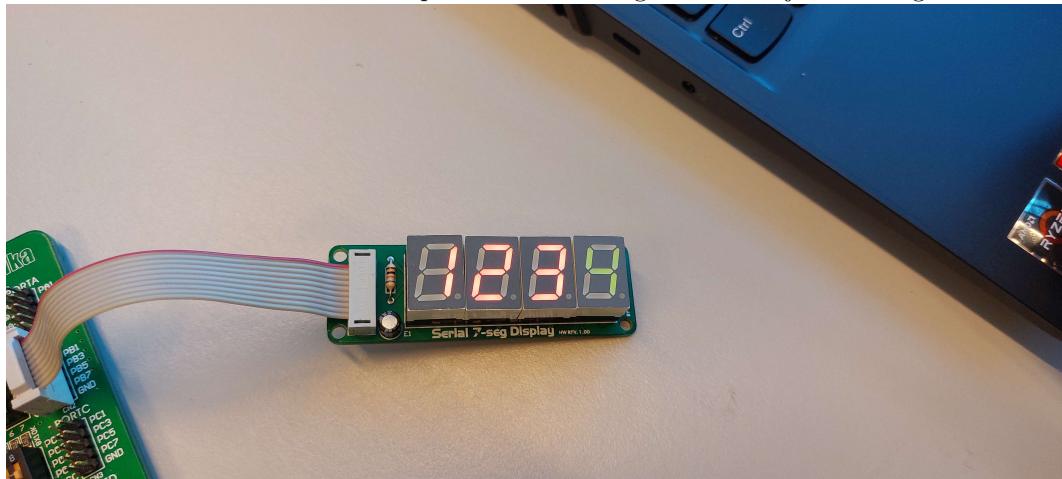
// write 4-digit data
for (char i =1; i<=4; i++)
{
    spi_slaveSelect(0);      // Select display chip
    spi_write(i);            // digit adress: (digit place)
    spi_write(5 - i);        // digit value: i (= digit place)
    spi_slaveDeSelect(0);    // Deselect display chip

    wait(1000);
}
wait(1000);

return (1);
}

```

Hierna moesten we de segmenten een lagere intensiteit geven. Dit hebben we kunnen doen door de intensiteit op 1 te zetten. Dit hebben we dan ook op dezelfde manier gedaan als bij het verhogen van de intensiteit.



Bij deel C wordt er van ons verwacht om de commando's die verstuurd moeten worden om de 7 segmenten aan te passen in een methode te stoppen. Dit omdat elk commando bestaat uit 4 berichten naar de segmenten. Op het moment dat dan in een methode gestopt wordt verbeterd dit de leesbaarheid en verminderd het de grote van de code.

```

#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>

#define BIT(x)          ( 1<<x )
#define DDR_SPI         DDRB
#define PORT_SPI        PORTB
#define SPI_SCK          1
#define SPI_MOSI         2
#define SPI_MISO         3
#define SPI_SS           0

// wait(): busy waiting for 'ms' millisecond
// used library: util/delay.h
void wait(int ms)
{
    for (int i=0; i<ms; i++)
    {

```

```

        _delay_ms(1);
    }

void spi_masterInit(void)
{
    DDR_SPI = 0xff;                                // All pins output: MOSI, SCK
    DDR_SPI &= ~BIT(SPI_MISO);                      // except: MISO input
    PORT_SPI |= BIT(SPI_SS);                        // SS_ADC == 1: deselect slave
    SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR1);      // or: SPCR = 0b11010010;
                                                       // Enable spi, Master mode
                                                       // Mode = 0: CPOL=0
}

// Write a byte from master to slave
void spi_write( unsigned char data )
{
    SPDR = data;                                    // Load byte to Data register
    while( !(SPSR & BIT(SPIF)) );                  // Wait for transmission complete
}

// Write a byte from master to slave and read a byte from slave
// nice to have; not used here
char spi_writeRead( unsigned char data )
{
    SPDR = data;                                    // Load byte to Data register
    while( !(SPSR & BIT(SPIF)) );                  // Wait for transmission complete
    data = SPDR;                                    // New received data (eventually)
    return data;                                  // Return received byte
}

// Select device on pinnummer PORTB
void spi_slaveSelect(unsigned char chipNumber)
{
    PORTB &= ~BIT(chipNumber);
}

// Deselect device on pinnummer PORTB
void spi_slaveDeSelect(unsigned char chipNumber)
{
    PORTB |= BIT(chipNumber);
}

void spi_writeWord ( unsigned char adress, unsigned char data ){
    spi_slaveSelect();                            // Select display chip
    spi_write(adress);                          // Register select
    spi_write(data);                           // set data value
    spi_slaveDeSelect();                      // Deselect display chip
}

// Initialize the driver chip (type MAX 7219)
void displayDriverInit()
{
    spi_writeWord(0x09, 0xFF); // Register 09: Decode Mode / -> 1's = BCD mode for all digits
}

```

```

    spi_writeWord(0x0A, 0x0F); // Register 0A: Intensity | -> Level 4 (in range [1..F])
    spi_writeWord(0x0B, 0x04); // Register 0B: Scan-limit | -> 1 = Display digits 0..1
    spi_writeWord(0x0C, 0x01); // Register 0B: Shutdown register | -> 1 = Normal operation
}

// Set display on ('normal operation')
void displayOn()
{
    spi_writeWord(0x0C, 0x01); // Register 0B: Shutdown register | -> 1 = Normal operation
}

// Set display off ('shut down')
void displayOff()
{
    spi_writeWord(0x0C, 0x00); // Register 0B: Shutdown register | -> 1 = Normal operation
}

int main()
{
    // initialize
    DDRB=0x01;                                     // Set PBO pin as output for display
    spi_masterInit();                                // Initialize spi module
    displayDriverInit();                            // Initialize display chip

    // clear display (all zero's)
    for (char i =1; i<=4; i++)
    {
        spi_writeWord(i, 0); // digit adress: (digit place) | digit value: 0
    }
    wait(1000);

    // write 4-digit data
    for (char i =1; i<=4; i++)
    {
        spi_writeWord(i, 5 - i); // digit adress: (digit place) | digit value: i (= digit place)

        wait(1000);
    }
    wait(1000);

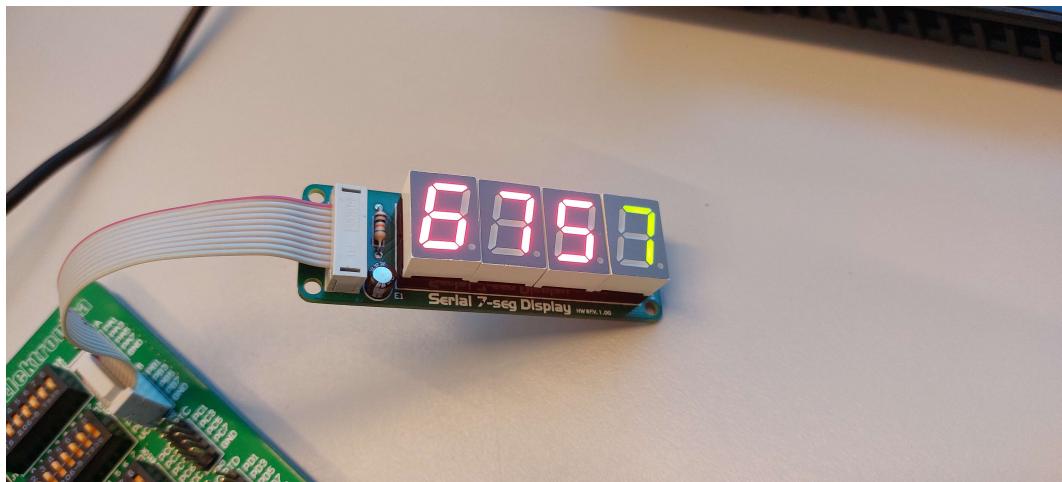
    return (1);
}

```

5.1.2 2

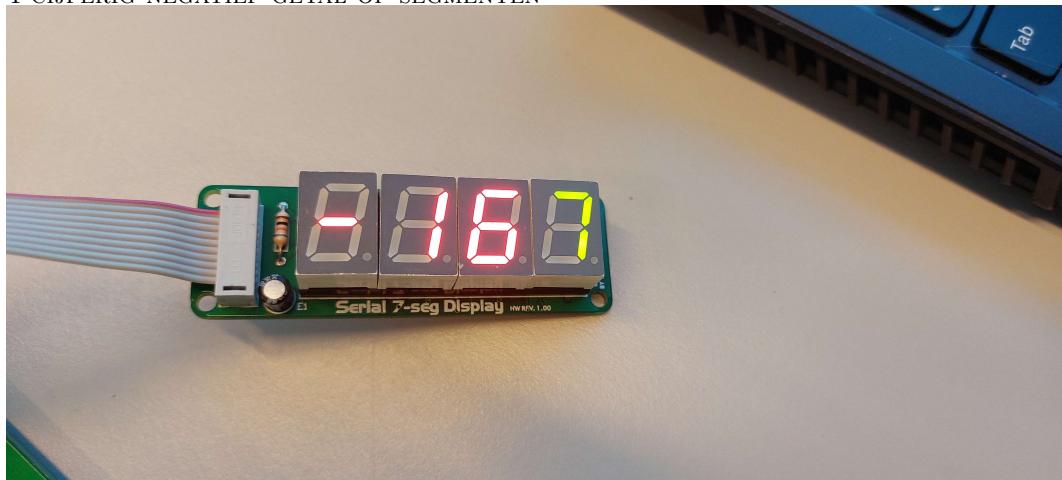
Hier was het de bedoeling een methode te maken die een getal van 4 cijfers op het 7 segmenten display laat zien. Hiervoor pakken we het gekozen getal met modulo 10. Dit zorgt ervoor dat we het achterste cijfer krijgen. Dit cijfer zetten we vervolgens op het segment. Vervolgens delen we het gekozen getal door 10 om te zorgen dat we bij het volgende cijfer kunnen. Dit doen we dan in totaal 4 keer. Op deze manier krijgen we het hele getal op de segmenten.

4 CIJFERIGE GETAL OP SEGMENTEN



Nadat we dit gedaan hadden was het de bedoeling om een negatief getal te tonen op de segmenten. Hiervoor hebben we in de methode eerst gekeken of het getal kleiner is dan 0. Als dit het geval is doen we voor de 3 cijfers de manier met de modulo van 10 en het delen door 10 toepassen ,om de cijfers op de juiste plek te tonen. Vervolgens sturen we er nog een commando om er een "-" voor te zetten.

4 CIJFERIG NEGATIEF GETAL OP SEGMENTEN



```
#define F_CPU 8e6
#include <avr/io.h>
#include <util/delay.h>

#define BIT(x)          ( 1<<x )
#define DDR_SPI         DDRB
#define PORT_SPI        PORTB
#define SPI_SCK          1
#define SPI_MOSI         2
#define SPI_MISO         3
#define SPI_SS           0

// spi Data direction register
// spi Output register
// PB1: spi Pin System Clock
// PB2: spi Pin MOSI
// PB3: spi Pin MISO
// PB0: spi Pin Slave Select
```

```

// wait(): busy waiting for 'ms' millisecond
// used library: util/delay.h
void wait(int ms)
{
    for (int i=0; i<ms; i++)
    {
        _delay_ms(1);
    }
}

void spi_masterInit(void)
{
    DDR_SPI = 0xff;                                     // All pins output: MOSI, SCK
    DDR_SPI &= ~BIT(SPI_MISO);                         // except: MISO input
    PORT_SPI |= BIT(SPI_SS);                           // SS_ADC == 1: deselect slave
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR1);          // or: SPCR = 0b11010010;
    // Enable spi, MasterMode, Clock rate fck/64, bitrate=125kHz
    // Mode = 0: CPOL=0, CPPH=0;
}

// Write a byte from master to slave
void spi_write( unsigned char data )
{
    SPDR = data; // Load byte to Data register --> starts transmission
    while( !(SPSR & BIT(SPIF)) );                      // Wait for transmission complete
}

// Write a byte from master to slave and read a byte from slave
// nice to have; not used here
char spi_writeRead( unsigned char data )
{
    SPDR = data; // Load byte to Data register --> starts transmission
    while( !(SPSR & BIT(SPIF)) );                      // Wait for transmission complete
    data = SPDR; // New received data (eventually, MISO) in SPDR
    return data;                                         // Return received byte
}

// Select device on pinnummer PORTB
void spi_slaveSelect(unsigned char chipNumber)
{
    PORTB &= ~BIT(chipNumber);
}

// Deselect device on pinnummer PORTB
void spi_slaveDeSelect(unsigned char chipNumber)
{
    PORTB |= BIT(chipNumber);
}

void spi_writeWord ( unsigned char adress, unsigned char data ){
    spi_slaveSelect(0);                                // Select display chip
    spi_write(adress);                                // Register select
    spi_write(data);                                  // set data value
    spi_slaveDeSelect(0);                            // Deselect display chip
}

```

```

}

// Initialize the driver chip (type MAX 7219)
void displayDriverInit()
{
    spi_writeWord(0x09, 0xFF); // Register 09: Decode Mode | -> 1's = BCD mode for all digits
    spi_writeWord(0x0A, 0x0F); // Register 0A: Intensity | -> Level 4 (in range [1..F])
    spi_writeWord(0x0B, 0x04); // Register 0B: Scan-limit | -> 1 = Display digits 0..1
    spi_writeWord(0x0C, 0x01); // Register 0B: Shutdown register | -> 1 = Normal operation
}

// Set display on ('normal operation')
void displayOn()
{
    spi_writeWord(0x0C, 0x01); // Register 0B: Shutdown register | -> 1 = Normal operation
}

// Set display off ('shut down')
void displayOff()
{
    spi_writeWord(0x0C, 0x00); // Register 0B: Shutdown register | -> 1 = Normal operation
}

void writeLedDisplay( int value ){
    if (value >= 0){
        for (char i =1; i<=4; i++)
        {
            int digit = value % 10;
            spi_writeWord(i, digit); // digit adress: (digit place) | digit value: 0
            value = value/10;
        }
    }
    else if (value < 0 & value > -1000){
        for (char i =1; i<=3; i++)
        {
            int digit = (value * -1) % 10;
            spi_writeWord(i, digit); // digit adress: (digit place) | digit value: 0
            value = value/10;
        }
        spi_writeWord(4, 10);
    }
}

int main()
{
    // initialize
    DDRB=0x01; // Set PBO pin as output for display s
    spi_masterInit(); // Initialize spi module
    displayDriverInit(); // Initialize display chip

    // clear display (all zero's)
    for (char i =1; i<=4; i++)
    {
}

```

```
        spi_writeWord(i, 0); // digit adress: (digit place) / digit value: 0
    }
    wait(1000);

    // write 4-digit data
    writeLedDisplay(6757);
    wait(1000);

}

return (1);
}
```