



## Oblig 3 - Halvdupleksprotokoll

Fredrik Sandhei, Mathias Haukås <sup>1</sup>

1. mai 2017

<sup>1</sup>UiT - AUT-1001, obligatorisk innlevering 3

# Innhold

Introduksjon . . . . .	2
Hensikt . . . . .	2
Teori . . . . .	2
Protokoll . . . . .	2
USART og SPI . . . . .	2
Interrupts og Timer2 . . . . .	3
ADC - Analog to Digital Converter . . . . .	3
Løsning . . . . .	4
Implementasjon . . . . .	4
Diskusjon . . . . .	5
Konklusjon . . . . .	6
Vedlegg - kildekode . . . . .	7

# Introduksjon

I samsvar med arbeidskravene til faget 'Programmering med mikrokontrollere', trengs det tre arbeidskrav, i form av tre obligatoriske innleveringer, godkjent. I dette dokumentet ligger vår besvarelse på den siste obligatoriske innleveringen - oblig 3.

## Hensikt

Formålet med denne oppgaven er å bruke det meste vi har lært i AUT-1001 i en og samme oppgave. Dette gjør at man må ha en forståelse av hvordan en mikrokontroller fungerer, og at man klarer å programmere den på en slik måte at man kan «snakke» med de forskjellige komponentene i den.

På bakgrunn av alle tidligere øvinger og tidligere innleveringer har vi lært de forskjellige aspektene av denne oppgaven. Det som skiller denne oppgaven fra de er at denne oppgaven krever at vi setter sammen de tidligere innleveringene og øvingsoppgavene. Vi bruker dermed ikke bare en del av kontrolleren i gangen, slik so vi tidligere har gjort. I denne oppgaven bruker vi både timer, ADC, USART, SPI, lys, leser av verdier av knapper og skriver tekst og verdier inn på en lcd (-display).

## Teori

I denne oppgaven skulle man bruke en programvare utgitt, Oblig3.exe, til å kommunisere med mikrokontrolleren. Det som skulle skrives fra datamaskinen er tekst, nåværende dag og tidspunkt. Man skulle også styre LED-lys og stille inn en frekvens på en høyttaler. Fra mikrokontrolleren skal det sendes hvilken verdi ADC-en måler og konvertere denne til en spenning mellom 0V og 5V, en fritt valgt tekst, hvilken stilling knappene er satt til og hvilken HEX-verdi velgeren på displaykortet har.

## Protokoll

Protokollen som skulle brukes i oppgaven er halv-dupleks: En datapakke sendes først fra datamaskinen til mikrokontrolleren ved hjelp av RS-232 - kommunikasjon. Deretter forventer protokollen en transmit fra mikrokontrolleren. Dataen som sendes og mottas er forventet å være i ASCII-format. Hver Rx - prosedyre er kontrollert ved hjelp av rekkefølgen på pakkeelementene: Hver pakke begynner på en bokstav, enten *A*, *B*, *C* eller *D* etterfulgt med line feed,  $\backslash n$ . Når en Rx-prosedyre er gjennomført, avsluttes pakken med *R* og  $\backslash n$ , og protokollen er klar for å motta en Tx-prosedyre, som ikke er like strengt lagt opp. Den forventer igjen det samme som i Rx, men rekkefølgen spiller ingen rolle, og ingen *R* trengs ikke for å konkludere transmit.

## USART og SPI

For å opprette kommunikasjon mellom PC og displaykortet er USART - Universal Synchronous Asynchronous Receive Transmit - et nyttig verktøy. Det er en type seriell kommunikasjon, der dataelementene/bitsene shiftes over en kabel mellom enhetene. Ved hjelp av en ekstra kabel kan en bruke en seriell klokke mellom enhetene for å synkronisere dataregistreringen hos mottakeren. I vårt tilfelle benyttes ikke den serielle klokken. Kommunikasjonen blir dermed asynkron - UART. Asynkron USART bruker et start-bit og et (evt. to) stopp-bit i tillegg til den dataen som sendes for å bestemme overføringshastigheten. UART-kommunikasjonen åpner for bruk av 'peripherene' på displaykortet.

Kommunikasjonen mellom ATmega644A og de ulike komponentene kalles SPI. Serial Peripheral Interface er seriell kommunikasjon over 'korte avstander', og benyttes for å kommunisere mellom flere enheter og mikrokontrolleren. ATmega644 fungerer som master, og periferene fungerer som slaver. Dataoverføring ved SPI fungerer som en loop: Data blir sendt fra master til slave via MOSI samtidig som data mottas fra slave til master via MISO. På denne måten kan slaven og master veksle informasjon på en enkelt kommunikasjonslinje. SPI-kommunikasjon ble brukt mellom følgende enheter:

- LCD - display: Liquid Crystal Display - display
- LED - lys
- Potensialmeter
- Piezo-buzzer
- ISP-connector

## Interrupts og Timer2

Flere av de periferene som ble brukt ble brukt i sammenheng med en interrupt. En interrupt, eller ISR - Interrupt Service Routine - 'forstyrrer' arbeidsflyten til prosessoren for å gjennomføre et sett med oppgaver implementert av programmereren til denne ISR-rutinen. Da settes alt annet på mikrokontrolleren på vent, og hovedprogrammet fortsettes etter at det nødvendige flagget fra interruptets opphavsregister registreres høyt. I dette tilfellet bruktes interrupts for UART-kommunikasjon, Timer2 - output-compare-mode - non-pwm-mode, samt ADC.

## ADC - Analog to Digital Converter

Når vi skal måle spenning på uK må man først åpne databladet til uK for å finne ut hvilke(n) av pinnen(e) som har mulighet til å fungere som ADC. Det er først etter dette at man kan begynne å programere en ADC. I vårt tilfelle er ADC0 den som er lettest å bruke, så da er det den som blir tatt i bruk.

Når man skal ta i bruk ADC0, må man først vite om det er 8- eller 10-bits konverter, dette finner man i databladet. når man har funnet ut av dette vet man hvor mange diskrete nivåer det er. Man må også finne ut hvor stort område det er man gjør målinger på, ettersom man kun har ett bestemt antall forskjellige nivåer vil spenningsvidden vi skal måle på ha stor innvirkning på hvor stor presisjon vi kan oppnå. Ettersom verdien ut fra ADC vil være:

$$Output = Bitstrrelse \cdot \frac{Input(V)}{Spenningsvidde(V)} \quad (1)$$

På ATmega644A er ADC0 en 10-bits konverter, og spenningsvidden er på 5V dette gir likningen:

$$Output = 1023 \cdot \frac{Input(V)}{5V} \quad (2)$$

# Løsning

## Implementasjon

Til vår løsning av problemstillingen benyttet vi oss av interrupts til å behandle informasjon mottatt fra PCen og til å sende informasjon. Tanken for vår løsning var å holde mikrokontrolleren og PCen i en konstant kommunikasjonsloop i samsvar med protokollen: Data blir mottatt, og data blir sendt. Når noe ble mottatt, gikk RXC - flagget i *UCSR0A* (USART Control And Status Register A) høyt, og RXC-ISR-rutinen mottok informasjonen shiftet fra PC og mottatt i UDR-registeret i ASCII-format. Datainnholdet ble lagret i et globalt array som ble sjekket for kjennetegnene for endring av pakkeinformasjon, i dette tilfellet en bokstav fra 'A' til 'D' eller 'R' og line-feed. Informasjonen ble lagret i ulike arrays (*A\_array*, *B\_array*, *C\_array*, *D\_array*) for videre behandling. Da det siste pakkeelementet 'R' og linefeed var mottatt, ble tilstandsvariabelen *receive\_done* satt høyt.

Da de ulike dataene hadde ulike hensikter, var det passende å systematisere informasjonen i en felles behandlingsfunksjon, *received()* (linje 112 - 175) som hadde det nullte elementet i hver pakke som argument. Funksjonen baserte seg på en switch-case som enten printet en linje på LCD-displayet, regulerte LED-lysene eller aktiverte TIMER2-ISR for bruk av OCR. For tilfellene 'A' og 'B', var det simpelthen kalling av funksjonen *lcd\_printline()* fra *serlcd.h* - biblioteket. LED-lysene ble regulert ved å tukle litt med AND-operatoren mellom GPIO-pinnene og *C\_array*. 'D'-tilfellet ble litt mer arbeid, da det krevdes å gjøre elementene i *D\_array* om til desimal - 1 enn ASCII-1. Hvert element ble trukket fra med '0' og multiplisert med enten 1000, 100 eller 10, basert på deres posisjon i arrayet. Add-verdien, det vil si verdien som trengs for å skape en Output Compare - ISR, er antall klokke-steg prosessorklokken må gå. Da mikrokontrollerens frekvens er  $F_{CPU} = 14.7456\text{Mhz}$  passer det med bruk av en 64-bits prescaler for å sette antall klokkeverdier ned. Da blir antall klokke-steg per millisekund:

$$F_{CPU/64} \approx 0.2304\text{Mhz} \approx 230\text{Khz} = 230 \frac{\text{klokketikk}}{\text{ms}} \quad (3)$$

Da høytaleren skulle produsere kvadrattølger med frekvens mellom 1000Hz og 3000 Hz, blir antall klokke-steg slik:

$$\text{add}_{f=1000\text{Hz}} = \frac{\frac{T_f}{2}}{T_{CPU/64}} = \frac{\frac{1}{2000\text{s}}}{4.35 \cdot 10^{-6}} \approx 115 \frac{\text{klokketikk}}{\text{ms}} \quad (4)$$

$$\text{add}_{f=3000\text{Hz}} = \frac{\frac{T_f}{2}}{T_{CPU/64}} = \frac{\frac{1}{6000\text{s}}}{4.35 \cdot 10^{-6}} \approx 38 \frac{\text{klokketikk}}{\text{ms}} \quad (5)$$

For å implementere dette inn i C, ble det gjort på en litt annerledes metode. Noen av verdiene ble regnet ut for hånd, slik at en kunne minske sjansen for reduksjon av resolution/oppløsning (antall verdier mellom 0 og maks-verdi).

```
1 volatile unsigned char add;  
add = (long)115200/freq;
```

## uart\_send()

Etter behandling av mottatt informasjon via *received()* kaltes funksjonen *uart\_send()* (linje 181 - 283). For å sende data på den enkleste måten, ble de ulike pakke-dataene først lagt i egne arrays, lignende som i mottakerfunksjonen. Til slutt ble arrayene slått sammen til et stort array, *send\_array*. Tankegangen på sammenslåingen er enkel: Da hver array avsluttes med linefeed, kunne man ved hjelp av en for-loop med to separate tellere *i* og *sep\_cntr* og if-tester sjekke etter linefeeds i sendepakken. Hvis en linefeed ble oppdaget, var det indikasjon på 'end of transmission' på pakken, og den neste pakken kunne leses. Da alle pakken var klar for sending, ble det nullte elementet i *send\_array* shiftet inn i UDR-registeret for å sette i gang en Tx-interrupt.

## ADC0 - problemet

Når ADC0 merker en endring aktiveres ett interrupt som direkte lagrer verdien i en variabel som blir manipulert slik at det blir gjort om til mV, dette fordi det er tegnene vi ønsker å vise på dataen. Dette gjøres inni interruptet slik at variabelen som kommer ut er nesten klar for sending.

En ting man kan legge merke til er at adressebryteren er koblet på en litt merkelig måte. Vi måtte dermed finne en måte å speile signalet som kom fra den. Signalene var nemlig slik:

Verdi på bryter	Output	Egentlig binær verdi
1	1000	0001
2	0100	0010
4	0010	0100
8	0001	1000
A	0101	1010

Vår løsning på dette ble å kjøre if-tester slik at vi isolerte hver enkelt pinne. Dermed kunne vi addere sammen verdiene på enkeltpinnene slik at den inputverdien vi fikk ble samme som binærverdi. For å gjøre verdien om til ASCII verdier slo vi opp en tabell som sa at 0 i ASCII var 48 (HEX 30). Derfor satte vi startverdi på variabelen til å være 48, og om verdien ble større eller lik 57 la vi til 7. På denne måten kom vi forbi tegnene som er mellom 9 og A i ASCII, og vi fikk ut et resultat vi kunne sende til programmet på dataen.

Se fra linje 220 til 236 i vedlegg main.c (vedlegg).

## Diskusjon

Mens vi jobbet med overføring mellom pc og uK møtte vi på et problem med informasjonsflyten. Dette var et problem som førte til at vi ikke klarte å overføre noe data. Etter at både hjelpelærer og faglærer hadde sett på koden, hvor ingen klarte å se ett spesifikt problem med koden tok faglærer å hentet ett oscilloskop hvor vi da kunne se dataflow. Det viste seg da at problemet var knyttet til sendetidspunkt fra uK til pc. Dette så vi vet at det var uregelmessige sendinger fra uK til pc samtidig som pc sendte data til uK, da vi så dette forsto vi hva som var feil med koden å endret den fra å gjøre operasjonen i en switch-case til en if-test. Dette gjorde at vi kunne kontrollere at vi hadde fått hele pakken fra pc-en før vi sendte data fra uK til pc. Dataen mottatt ble aldri sjekket for hvilket element som ble mottatt, bare etter linefeed, noe som gjorde vanskelig for protokollen å bestemme hva slags pakke den hadde mottatt.

Etter hvert som vi kodet og vi ikke fikk de dataene som vi ønsket frem på dataen tok vi i bruk LCD

til feilsøking. Dette ble gjort ved å skrive ut dataene på linjene og så dermed hva som kom ut, og ettersom vi visste hvilke tall eller tegn vi ønsket å se, kunne vi jobbe ut fra det og klarte dermed å finne ut hva som skulle til for å få det til. I motsetning til når vi så hva som kom på datamaskinen, hvor det, pga. feil med sendingen til PC, ikke kom noe som helst.

## Konklusjon

Koden implementert for denne problemstillinga var basert på å følge den halvduplekse protokollen som var utgitt. Ved bruk av interrupts ble det dannet en mottaks-og sendestruktur på den serielle kommunikasjonslinjen mellom uK og PC. Denne implementasjonen fungerte godt til dette formålet, selv om bruken av all interrupts kan gjøre at koden 'stopper opp', dvs ikke kommer seg av videre pga gjentakende interrupts. Men da denne protokollen bare drev med sending og mottaking av spesifikk informasjon, ble dette ikke noe stort faremoment å ta til vurdering.

## Vedlegg - kildekode

```
/*
2 * oblig3.c
*   OBLIG 3
4 *   Gjennomført av Fredrik Sandhei og Mathias Haukås
*
6 */

8 #define F_CPU 14745600

10 //Inkluderinger av ulike bibliotek
#include <avr/io.h>
12 #include <avr/interrupt.h>
#include "serlcd.h"
14 #include <util/delay.h>

16 //Funksjonsprototyper
void received(char input);
18 void uart_send();

20 //Deklarasjon av variabler
char A_array[18], B_array[18], C_array[7], D_array[7], C_to_send[6], D_to_send[4],
    A_to_send[8], send_array[50];
22 char rec_array[20];
volatile unsigned char rx_counter = 0, sendcounter = 1, receive_done, add;
24 unsigned int freq, pwm_on, voltage, adc_reader;
//Fri tekst i B
26 char B_to_send[22] = { "BE Emmy slår..\n" };

28 //ADC interrupt
ISR(ADC_vect) {
30     //Bruker et pseudoregister sammenslått av ACDL-og ADCH - registrene
    adc_reader = ADCW;
32     voltage = 5000 * (long)adc_reader / 1023; //regner om til mV
}

34 //Timer2 Output Compare Interrupt
//som produserer kvadratbølger med frekvens
//mellom 1000 - 3000Hz
38 ISR(TIMER2_COMPA_vect)
{
40     OCR2A = OCR2A + add;
}

42 //Shifter ut innholdet fra send_array() som ble lagt sammen i uart_send()
44 ISR(USART0_TX_vect) {
    //Dersom det er noen elementer i index sendcounter, shift ut på UDR.
46     if (send_array[sendcounter]) {
        UDR0 = send_array[sendcounter++];
48     } //Hvis ikke, tilbakestill sendcounter
    else {
50         sendcounter = 1;
    }
52 }
```



```

54 //Behandler dataen mottatt over UDR-registeret fra PC
55 //og legger dataen inn i sine respektive arrayer.
56 ISR(USART0_RX_vect) {
57     unsigned char uart = UDR0;
58     unsigned char i = 0;
59     if (uart == '\n') {
60         if (rec_array[0] == 'A')
61             for (i = 0; i<16; i++)
62                 A_array[i] = rec_array[i + 1];
63         else if (rec_array[0] == 'B')
64             for (i = 0; i<16; i++)
65                 B_array[i] = rec_array[i + 1];
66         else if (rec_array[0] == 'C')
67             for (i = 0; i<4; i++)
68                 C_array[i] = rec_array[i + 1];
69         else if (rec_array[0] == 'D')
70             for (i = 0; i<5; i++)
71                 D_array[i] = rec_array[i + 1];
72         else if (rec_array[0] == 'R')
73             receive_done = 1;
74         rx_counter = 0;
75     }
76     else
77         rec_array[rx_counter++] = uart;
78 }

80 int main(void)
81 {
82     init_lcd();
83     lcd_cursoron();
84     //Oppsett av porter, input/output
85     //Da de ulike enhetene var koblet til jord
86     //vil uK-portene registre logisk 0.
87     //Det er derfor viktig å aktivere de interne pull-up-resistanene
88     //til uK-portene, for å hindre at verdiene som blir lest bare

90     DDRB = 0xf0;
91     PORTB = 0x0f;
92     DDRC = 0x10;
93     PORTC = 0x0f;
94     DDRD = 0xf0;

96     //Timer 2 – konfigurasjon
97     TCCR2B = (1 << CS22);
98     TIMSK2 = (1 << OCIE2A);
99     //UART – konfigurasjon
100    //Rx enable, Tx enable, Rx interrupt enable, Tx interrupt enable
101    UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0) | (1 << TXCIE0);
102    //Bitrade 9600 – standard hastighet på Rs-232
103    UBRR0H = 0;
104    UBRR0L = 95;
105    //ADC – konfigurasjon
106    //Bruker ADC0 som input
107    ADMUX = 0x00;
108    //ADC enable, interrupt enable, clock speed = 115200Hz
109    ADCSRA = (1 << ADEN) | (1 << ADIF) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

```

```

110 //Assembly – Global Interrupt Enable
    sei();
112 while (1)
    {
114     //Mottar data først fra PC
    //For så å sende data fra uK til PC
116     //ved hjelp av halv-dupleks-protokollen/Bernt-protokollen
    if (receive_done)
118     {
        received('A');
120        received('B');
        received('C');
122        received('D');
        uart_send();
124        receive_done = 0;
    }
126 }
}

128 void received(char input) {
    //Behandler informasjonen mottatt
130    //basert på arrayet som mottas

132    switch (input)
    {
134        case 'A':
            lcd_printline(0, 0, A_array);
136            break;

138        case 'B':
            lcd_printline(1, 0, B_array);
140            break;

142        //Setter portene høy basert på
        //innholdet i C_array
144        case 'C':
            if (0x0f & C_array[0])
146                PORTB |= (0x0f & C_array[0]) << 4;
            else
148                PORTB &= ~(0x10);
            if (0x0f & C_array[1])
150                PORTB |= (0x0f & C_array[1]) << 5;
            else
152                PORTB &= ~(0x20);
            if (0x0f & C_array[2])
154                PORTB |= (0x0f & C_array[2]) << 6;
            else
156                PORTB &= ~(0x40);
            if (0x0f & C_array[3])
158                PORTB |= (0x0f & C_array[3]) << 7;
            else
160                PORTB &= ~(0x80);
            break;
162
164        case 'D':

            if (D_array[0] == '1')

```

```

166     {
167         //Dersom pwm-flagget ikke er høy -> sett høy.
168         //Gjør det slik at denne kodesnutten bare kjøres
169         //en gang.
170         if (pwm_on == 0)
171         {
172             TCCR2A |= (1 << COM2A0);
173             pwm_on = 1;
174         }
175         //Da hvert siffer av frekvensen er
176         //lagret i hver sin posisjon i arrayet
177         //må de subtraheres med 48 (eller '0')
178         //for så å multipliseres med nødvendig verdi
179         freq = (D_array[1] - '0') * 1000;
180         freq += (D_array[2] - '0') * 100;
181         freq += (D_array[3] - '0') * 10;
182         freq += (D_array[4] - '0');
183         //Regner ut klokketiks med F_CPU_/64.
184         add = 115200 / freq;
185     }
186     else
187     {
188         if (pwm_on)
189         {
190             //Toggler av OC2A (Output Compare 2A)
191             TCCR2A &= ~(1 << COM2A0);
192             pwm_on = 0;
193         }
194         break;
195     default:
196         break;
197 }
198
199 //Legger til datapakkene som skal sendes fra uK til sine respektive pakker
200 //og deretter legger de sammen til et array som sendes ved UART.
201 //Denne metoden gjør ISR-transmit liten og enkelt implementert.
202 void uart_send() {
203     //Appending data for transmit
204     unsigned char lf = 0, sep_cntr = 0;
205
206     //Legger inn pakke A til sending'
207     //Start conversion for ADC
208     //Dette fører til en ADC-ISR-rutine
209     ADCSRA ^= (1 << ADSC);
210
211     A_to_send[0] = 'A';
212     A_to_send[1] = (voltage / 1000 + 0x30);
213     A_to_send[2] = ((voltage % 1000) / 100 + 0x30);
214     A_to_send[3] = ((voltage % 100) / 10 + 0x30);
215     A_to_send[4] = (voltage % 10 + 0x30);
216     A_to_send[5] = 'm';
217     A_to_send[6] = 'V';
218     A_to_send[7] = '\n';
219
220     //Legger inn pakke C til sending

```

```

222 //Sjekker om bryteren er trykket ned
C_to_send[0] = 'C';
224 for (unsigned char i = 1; i < 5; i++)
{
226     if ((PINB & (1 << (i - 1))) == 0)
        C_to_send[i] = '1';
228     else
        C_to_send[i] = '0';
230 }
C_to_send[5] = '\n';
232

234 //Legger inn pakke D til sending
char address_val = 0x30;
236 if (PINC & 0x01)
    address_val += 8;
238 if (PINC & 0x02)
    address_val += 4;
240 if (PINC & 0x04)
    address_val += 2;
242 if (PINC & 0x08)
    address_val += 1;
244
246 if (address_val >= 57)
    address_val += 7;

248 D_to_send[0] = 'D';
D_to_send[1] = address_val;
250 D_to_send[2] = '\n';

252 //Legger alt sammen i et monster - array. Big time
for (unsigned char i = 0; i < 50; i++)
254 {
    if (lf == 0)
256     {
        send_array[i] = A_to_send[sep_cntr];
258         if (A_to_send[sep_cntr] == '\n')
            {
260             //Dersom linefeed - inkrementer lf, nullstill sep_cntr og hoppe til neste
iterasjon
                lf = 1;
262                 sep_cntr = 0;
                continue;
264             }
        }
266     if (lf == 1)
        {
268         send_array[i] = B_to_send[sep_cntr];
        if (B_to_send[sep_cntr] == '\n')
270             {
                //Dersom linefeed - inkrementer lf, nullstill sep_cntr og hoppe til neste
iterasjon
                lf = 2;
272                 sep_cntr = 0;
                continue;
274             }
        }
    }
}

```

```

276     }
277     if (lf == 2)
278     {
279         send_array[i] = C_to_send[sep_cntr];
280         if (C_to_send[sep_cntr] == '\n')
281         {
282             //Dersom linefeed – inkremitter lf, nullstill sep_cntr og hoppe til neste
iterasjon
283             lf = 3;
284             sep_cntr = 0;
285             continue;
286         }
287     }
288
289     if (lf == 3)
290     {
291         send_array[i] = D_to_send[sep_cntr];
292         if (D_to_send[sep_cntr] == '\n')
293         {
294             //Dersom linefeed – inkremitter lf, nullstill sep_cntr og hoppe til neste
iterasjon
295             lf = 4;
296             sep_cntr = 0;
297             continue;
298         }
299     }
300     sep_cntr++;
301 }
302 //Begynner transmit.
303 //Dette vil føre til at TXC-flagget blir høy,
304 //og en TXC ISR vil gjennomføres.
305 UDR0 = send_array[0];
306 }

```

main.c