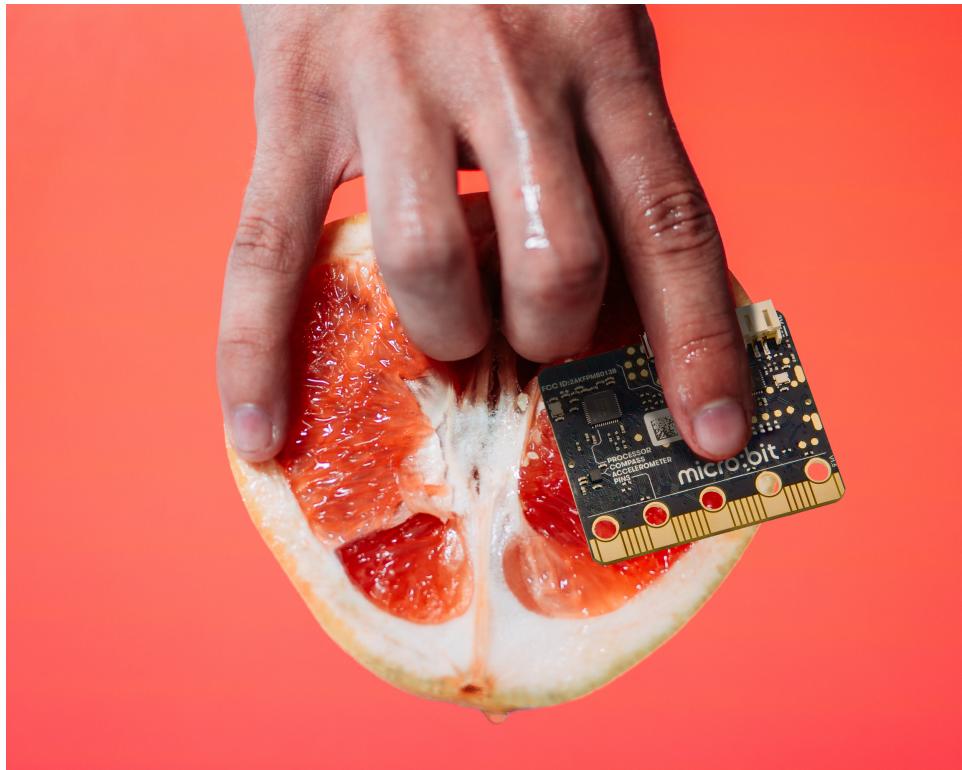


Tilpassede Datasystemer

Alternativ Mikrokontrollerlab

Kolbjørn Austreng



“Man blir jo ikke høy av å spise vanlig sopp som selges over diskene vel?”

Formål

Hvorfor gidde å skrive en helt ny oppgavetekst på grunn av litt SARS-CoV-2? Resten av Gløs har allerede tatt en mer “mañana mañana”-stilling til hele greie, og nøyser seg med å gjøre diverse laber frivillige, selv når internett-baserte løsninger byr seg frem.

I Tilpassede Datasystemer er det ganske uheldig, for jeg er ikke imponert over hva dere (ikke) har lært i DatDig, og egentlig ikke C++ heller. Denne alternative oppgaven er derfor et ærlig forsøk på å ikke la et helt kull fra Kyb bli like udugelige som Datateknologi.

Min ærlige mening er dette: Dere *må* kunne lese et datablad, og dere *må* beherske pekere. Ikke fordi dere alle er garantert til å jobbe mye med mikrokontrollere eller lavnivå kode i fremtiden - men fordi både datablad og pekere trener deres øye for detaljer. Ikke fortell meg at det ikke er nyttig i *basically alle andre sammenhenger*. Utover dette er pekerprogrammering et paradigme som jeg personlig mener kan gi dere dypere innsikt i hvordan andre former for programmering fungerer. Dette gjelder for såvidt ikke bare pekere - men de fleste former for programmering. Av samme argument vil jeg anbefale alle å utforske “funksjonell” programmering - et riktig kult paradigme hvor dere ikke har variabler eller løkker ;)

Sånn, når denne prekenen om læringsfilosofi er unnagjort, kan vi begynne å gjøre litt ting med mikrokontrollere. Dette er en kortere oppgave som burde være enklere å gjennomføre på egenhånd. Formålet med oppgaven er å putte micro:biten på internett ved å gjøre litt triksing med UARTen og litt kode som jeg har skrevet for dere for å oversette til websockets. Langs veien dekker vi mye av det alle bør kunne om mikrokontrollere¹.

¹Til og med bestemor Beatrice bør kunne dette.

1 Før vi starter

Denne oppgaveteksten forutsetter at førstegangsoppsettet fra den opprinnelig oppgaveteksten er fulgt. Det er ganske rett frem; bare last ned en hex-fil fra segger.com og gjør noen enkle greier.

For de som sitter på Windows bør det være ganske greit å følge Martin sin oppskrift (under “undervisningsmateriell” på blackboard). For dere vil nok ikke micro:biten hete “`/dev/ttyACM0`” når dere kommer til UART. Men det finner dere ut av ;)

Windows har heller ikke `picocom`, så vidt jeg vet, så da kan dere Google etter “PuTTY”, som er en tilsvarende greie for Windows.

Når det er sagt, håper jeg selvsagt å spre Linuxevangeliet til så mange av dere som mulig ☺.

Om noe skulle være uklart i oppgaveteksten, eller bare ikke gi mening, så er det bare å sende meg en epost. Dere er også MVPs hvis dere i tillegg bruker forumet på blackboard, sånn at andre kan lære av dere.

2 Minnemappet IO

De fleste moderne prosessorer har to ting til felles: de er RISC²-baserte, og de har minnemappet IO.

2.1 RISC

En RISC-basert prosessor har en enklere intern oppbygging enn en CISC³-basert prosesser, og er derfor billigere å lage med tanke på mengden silisium som skal til, i tillegg til at den trekker mindre strøm. Kostnaden av dette er at man trenger flere instruksjoner for å gjøre samme ting som på en CISC. Dette betyr også at programmene blir litt større for en RISC enn for en CISC i det generelle tilfellet - men dette er 2020 og lagringsplass er ikke særlig dyrt lengre.

2.2 IO

En prosessor som ikke er i stand til å snakke med eksterne enheter er bare en kalkulator som ingen kan bruke. Derfor trenger man alltid en eller annen måte å koble prosessoren til omverdenen på. Her er det to forskjellige metoder som rår; om man separerer minne og eksterne enheter fra hverandre, har man implementert “Port-mapped IO” (også kjent som “isolert IO”). Da trenger prosessoren egne instruksjoner som får den til å gå inn i “ekstern enhet”-modus.

Alternativet, som nå ruler, er “minnemappet IO”. Her vil forskjellige adresser på prosessoren være koblet til forskjellige ting. En vilkårlig adresse kan nå være koblet inn i RAM, sånn at den refererer til hovedminne - men den kan også være koblet til en egen enhet, som for eksempel en modul som er i stand til å skru eksterne spenninger på eller av. Det er dette vi skal se på i denne oppgaven.

2.2.1 Hva er en ekstern enhet?

Når vi snakker om “eksterne enheter”, så er det naturlig å tro at de ligger utenfor chipen vi programmerer. “Ekstern” refererer til “utenfor prosessorkjernen” - altså utenfor ARM Cortex-M0en i vårt tilfelle. De “eksterne” enhetene er fortsatt en del av nRF51en, så de ligger inne i samme chip på micro:biten. Dette er illustrert i figur 1 i nRF51-databladet.

²Reduced Instruction Set Computer

³Complex Instruction Set Computer

2.3 Hvordan snakker vi med eksterne enheter?

Når vi skal lese fra- eller skrive til eksterne enheter på en mikrokontroller med minnemappet IO, er fremgangsmåten alltid den samme: vi oppretter en peker til området vi ønsker å gjøre noe med - så vil mikrokontrolleren oversette adressen til riktig enhet på adressebussen.

For å få til dette, må vi vite to ting:

1. Hvordan registrene til den eksterne enheten ser ut.
2. Hvor den eksterne enheten ligger plassert i adresseområdet.

Det er her datablad kommer inn. Alle mikrokontrollere er litt forskjellige, og alle kommer med et datablad som forklarer hvordan de skal brukes. Når det kommer til mikrokontrollere er **The Church of Datasheet** religionen vi alle praktiserer.

I C-kode finnes det en veldig vanlig fremgangsmåte å gjøre disse to tingene:

1. Vi oppretter en **struct** som har samme struktur som registrene til den eksterne enheten.
2. Vi lager en **struct peker** som peker til adressen til den eksterne enheten.

Om vi gjør dette, så får vi tilgang til den eksterne enhetens registre gjennom feltene i **structen**. For all minnemappet programmering dere kommer til å gjøre gjennom C, er dette sannsynligvis måten dere kommer til å ønske å gjøre det på.

2.4 Oppgave: GPIO

Vi skal bruke en ekstern enhet hos nRF51en som styrer spenninger på eksterne pinner. Dette kalles “**General Purpose Input Output**”, og lar oss gjøre ting som å sjekke om knapper er trykket inne, eller skru av- og på LEDer.

2.4.1 Finn koblingen til knappene

Åpne “schematics.pdf” som ligger i mappen “oppgave og datablad”. Helt øverst til høyre i dette dokumentet vil dere finne oppkoblingen for knapp “A” og knapp “B” på micro:biten. Vi må vite to ting:

1. Er knappene “aktivt høye” eller “aktivt lav”? Det vil si, når knapper er trykket inne, vil mikrokontrolleren lese en høy- eller lav spenning fra knappen?

2. Hvilke pinner på mikrokontrolleren er knappene koblet til? Det er tallene som står *inne i den gule blokken til nRF51en* som gjelder. For eksempel er “TGT_NRST” koblet til pinne 19, ikke 39.

Når dere har funnet dette, åpner dere “button.c” i mappen “io/_source” og fyller inn “BUTTON_A_PIN” og “BUTTON_B_PIN”.

Fyll også inn funksjonene “button_a_pressed” og “button_b_pressed”. Disse funksjonene skal returnere 1 om den tilhørende knappen er trykket inne, og 0 hvis den ikke er det. Måten vi kan sjekke for dette på er å lese registeret som heter “IN” i GPIO-modulen, som gjøres slik:

```
/* Returns the IN register of GPIO */  
GPIO->IN;
```

For å vite hva registeret inneholder, leser vi først seksjon 14 i databladet til nRF51en, før vi tar en titt på registerforklaringen på bunnen av side 62 i samme datablad.



Dette er alltid fremgangsmåten. Hvis vi lurer på hvordan GPIO fungerer, går vi i *innholdsfortegnelsen* og ser etter den riktige seksjonen. Så leser vi den for å skjønne hva vi driver med.

Det er *nesten helt meningsløst* å bruke Ctrl+F i datablad, fordi det vil gi “falske positiver” overalt. Bruk innholdsfortegnelsen istedenfor.

Måten vi kan sjekke enkelte bit i et register på er slik:

```
/* Checks if bit number 12 in register IN is set */  
GPIO->IN & (1 << 12);  
  
/* Checks if bit 2 and 3 in register IN are set */  
GPIO->IN & (1 << 2) | (1 << 3);
```

Når dere har fylt inn “button.c”, kan dere kalle “make” og “make flash” fra mappen “1_io”. Når dere trykker på de to knappene på micro:biten skal lysene i LED-matrisen gå på- og av. Det er alltid litt *instant satisfaction* i å se at software har effekter på hardware.

3 UART

Flytt dere nå inn til mappen “2_uart/source”. I filen “uart.c” skal vi lage noe tilsvarende som den magiske structen dere fikk utlevert i “gpio.h” i oppgave 1. For å ikke drukne dere i detaljer er det meste av structen gitt allerede. Vi trenger bare å fylle inn UARTens startadresse, og “Tasks”-registrene.

All informasjonen dere trenger er forklart i seksjon 29.10 i databladet til nRF51en. For å illustrere måten dere oversetter denne informasjonen til C, vil jeg her forklare måten vi kom fram til GPIO-structen fra tidligere:

3.1 Start med å finne baseadressen

Det første vi gjør, er å bruke *innholdsfortegnelsen* til å navigere oss frem til riktig seksjon. Så finner vi registeroversikten til modulen vi er interessert i. I GPIO-tilfellet finner vi det dere ser i figur 1. Dette forklarer oss at baseadressen til GPIO-modulen er 0x50000000.

14.2 Register Overview

Table 74: Instances

Base address	Peripheral	Instance	Description
0x50000000	GPIO	GPIO	GPIO Port

Figur 1: Startadressen til GPIO-modulen.

Enkelte eksterne modulen vil ha flere forskjellige “instanser”. Et eksempel på dette er “Timer”-modulen til nRF51en. Der finnes det tre forskjellige “kopier” av samme enhet, som illustrert i figur 2. I slike tilfeller finnes det flere identiske enheter som gjør samme ting på chipen. Dette er for eksempel kjekt hvis vi ønsker flere uavhengige klokker, men er ikke noe vi trenger å tenke på her.

18.2 Register Overview

Table 140: Instances

Base address	Peripheral	Instance	Description
0x40008000	TIMER	TIMER0	Timer/Counter
0x40009000	TIMER	TIMER1	Timer/Counter
0x4000A000	TIMER	TIMER2	Timer/Counter

Figur 2: Startadressen(e) til Timermodulen.

Når vi først har funnet startadressen, oversettes dette ganske direkte inn i C slik:

```
#define GPIO ((NRF_GPIO_REG*)0x50000000)
```

Måten å lese dette på er slik: "Vi sier at *GPIO* er en peker til adresse *0x50000000*, og pekeren er av type *NRF_GPIO_REG*".

Når vi senere døreferer denne pekeren og bruker et av feltene i structen ved å skrive for eksempel *GPIO->IN*, vil vi som sagt få tilgang til modulens "IN"-register. Dette er derimot bare første steg; vi må også definere hvordan "NRF_GPIO_REG" ser ut.

3.2 Definer registerutseende

Under modulens startadresse vil vi se hvilke registre den har, se figur 3:

Table 75: Register Overview

Register	Offset	Description
Registers		
<i>OUT</i>	0x504	Write GPIO port
<i>OUTSET</i>	0x508	Set individual bits in GPIO port
<i>OUTCLR</i>	0x50C	Clear individual bits in GPIO port
<i>IN</i>	0x510	Read GPIO port
<i>DIR</i>	0x514	Direction of GPIO pins
<i>DIRSET</i>	0x518	DIR set register
<i>DIRCLR</i>	0x51C	DIR clear register
<i>PIN_CNF[0]</i>	0x700	Configuration of GPIO pins
<i>PIN_CNF[1]</i>	0x704	Configuration of GPIO pins
<i>PIN_CNF[2]</i>	0x708	Configuration of GPIO pins
<i>PIN_CNF[3]</i>	0x70C	Configuration of GPIO pins

Figur 3: Registrene i GPIO-modulen.

Det vi trenger å se på er

1. Navnet til registeret vi skal mappe.
2. *Offsetet* mellom dette registeret og det som kom før.

Når det står at registeret "OUT" har et offset på 0x504, betyr det at dette registeret ligger $504_{16} = 1284_{10}$ antall byte unna forrige register. Det ligger ikke noe register før "OUT" i GPIO-modulen, så dette betyr at det er 1284_{10} byte mellom startadressen til modulen og "OUT". I C blir dette slik:

```
typedef struct{
    volatile uint32_t RESERVED0[321];
    volatile uint32_t OUT;
    ...
} NRF_GPIO_REG;
```

Grunnen til at vi skriver 321 og ikke 1284 er at hver register er 32 bit stort - altså 4 byte. Adressen er oppgitt i byte, mens hvert register er et "word". Måten vi vet at hver adresse svarer til ett byte er at ARM (de som har laget prosessorkjernen) har designet det slik og fortalt oss om det.

Fordi hvert register tar 4 byte, vet vi at registeret "OUT" vil okkupere offsetene 0x504, 0x505, 0x506, og 0x507. Den neste ledige adressen etter "OUT" er derfor 0x508. Dette er samme offset som registeret "OUTSET" har, som betyr at det ikke er noen tomrom mellom "OUT" og "OUTSET".

Siden registeret "OUTSET" kommer rett etter "OUT" oversettes dette til C slik:

```
typedef struct{
    volatile uint32_t RESERVED0[321];
    volatile uint32_t OUT;
    volatile uint32_t OUTSET;
    ...
} NRF_GPIO_REG;
```

Slik fortsetter vi nedover listen, helt til vi kommer til registeret "DIRCLR". Dette registeret starter på adresse 0x51C, som betyr at det okkuperer de fire adressene 0x51C, 0x51D, 0x51E, og 0x51F. Altså er den neste ledige adressen 0x520. Registeret PIN_CNF[0] starter derimot ikke på denne adressen, men på 0x700, som betyr at det er et tomrom mellom de to registrene.

For hvert "tomrom" vi finner i minnekartet, putter vi inn et nytt "RESERVED"-felt. Størrelsen på dette finner vi ved å ta differansen mellom startadressen til "PIN_CNF[0]" og *neste ledige* adresse etter "DIRCLR":

$$\begin{aligned} 700_{16} - 520_{16} &= 1792_{10} - 1312_{10} = 480 \text{ byte} \\ 480 \text{ byte} &= 120 \text{ word} \end{aligned}$$

Ergo blir den tilsvarende C-koden slik:

```
typedef struct{
    ...
    volatile uint32_t DIRCLR;
    volatile uint32_t RESERVED1[120];
    volatile uint32_t PIN_CNF[32];
} NRF_GPIO_REG;
```

Protip: Dere kan åpne python i en terminal og skrive slike uttrykk direkte. Python er en grei kalkulator:

```
>>> (0x700 - 0x520) / 4
120.0
```

3.2.1 Formålet med “volatile”

Når vi deklarerer noe som “**volatile**”, forteller det kompilatoren at variabelen som er **volatile** kan endre seg uten at programmet vårt har gjort noe. For å gi et enkelt eksempel:

```
int a = 6;

while(a != 7){
    /* Do nothing */
}
```

Hvis vi antar at dette kjører på én tråd, og at ingen andre har referanser til **a**, så vil dette være ekvivalent med en evig løkke uten noen variabel:

```
while(1){
    /* Do nothing */
}
```

Kompilatoren står da fritt til å “skrive om koden litt”, fordi det ikke endrer oppførselen til programmet. Om vi derimot deklarerer **a** som **volatile**, vil det fortelle kompilatoren at hardwaret vi kjører på kanskje kommer til å endre denne variablene, og at vi derfor *må* beholde den, og lese den på nytter hver gang vi bruker verdien dens. Dette er definitivt ønskelig i systemer med minnemappet IO.

3.3 Initialiser UARTen

Når dere har fylt inn startadressen og registerkartet til UARTen skal vi skrive ferdig “**uart_init**”. Denne funksjonen skal konfigurere UARTen til å operere med en hastighet på 9600 “Baud”⁴, i tillegg til at UARTen interne logikk skal kobles til de riktige pinnene på nRF51-chipen.

3.3.1 Pinnekonfigurasjon

Ta først en titt i “schematics.pdf” for å se hvordan nRF51en er koblet til “TGT_RXD” og “TGT_TXD”. Igjen er det tallene inne i den gule blokken som er pinnene på nRF51en. Når dere har funnet disse, definerer dere “PIN_TX” og “PIN_RX” i “**uart.c**”.

Når dere har funnet de riktige pinnene, skal vi se på registeroversikten til UARTen i seksjon 29.10 igjen. Vi er på utkikk etter registrene “PSELRTS”, “PSELTxD”, “PSELCTS”, og “PSELrxD”. Her står “PSEL” for “Pin Select”. I databladet kan dere trykke på registernavnet for å bli tatt til oversikten for det registeret - som forklarer dere hva dere skal putte inn i det.

⁴Baud er det samme som symbol per sekund

Pinnene TXD og RXD skal naturlig nok settes til pinnene vi fant tidligere. CTS og RTS, som henholdsvis står for “Clear to send” og “Request to send”, skal ikke brukes. For å si at en pinne ikke skal brukes skriver vi en magisk “Disconnected”-verdi til PSELxxx-registeret. Denne verdien står i databladet.

3.3.2 Baudrate

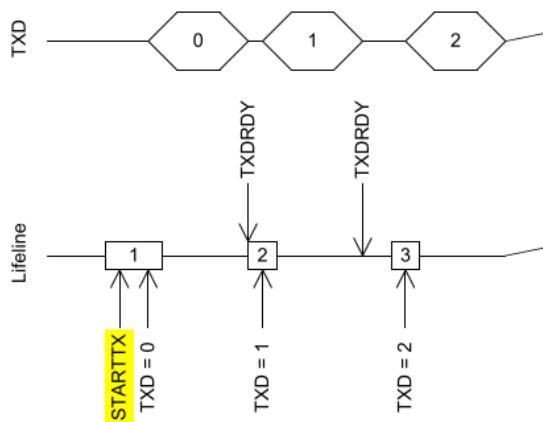
Baudraten skal som sagt settes til 9600 Baud. Igjen tar vi en titt i registeroversikten til UARTen, og finner registeret kalt “BAUDRATE”. Under forklaringen til dette registeret står det hvilken verdi som skal settes.

3.4 Implementer sendefunksjonalitet

Til slutt skal vi implementere funksjonen “uart_send”. Denne skal ta inn én byte, og sende den ut på UARTens linjer. For å vite hva vi skal gjøre i C, tar vi en titt på seksjon 29.4 i databladet. Spesielt legger vi litt ekstra innsats i å forstå figur 68 i samme seksjon. Disse “lifeline”-figurene er veldig nyttige, men de kan være litt forvirrende i starten. La oss derfor gå gjennom dette sammen:

3.4.1 How to read a datasheet

Første setning i seksjon 29.4 i nRF51-databladet er “*A UART transmission sequence is started by triggering the STARTTX task*”. Dette kjenner vi igjen i figur 68 i databladet. Her gjengitt i figur 4.

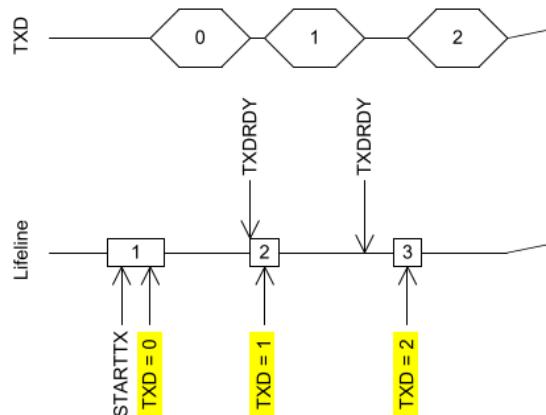


Figur 4: Oppstart av sending på UART.

Hver gang Nordic snakker om “staring a task”, vil det oversettes til å skrive 1 til det tilhørende “task”-registeret, som forklart i seksjon 10.1.3 i nRF51-databladet. I vårt tilfelle starter vi derfor slik:

```
void uart_send(uint8_t byte){
    UART->STARTTX = 1;
    ...
}
```

So far, so good. Neste setning i databladet sier “*Bytes are transmitted by writing to the TxD register*”. Dette er også noe vi kjenner igjen i figur 68 i databladet, gjengitt her som figur 5. Her står “0”, “1”, og “2” for nullte, første, og andre byte vi sender. Vi trenger bare å bry oss om ett byte.

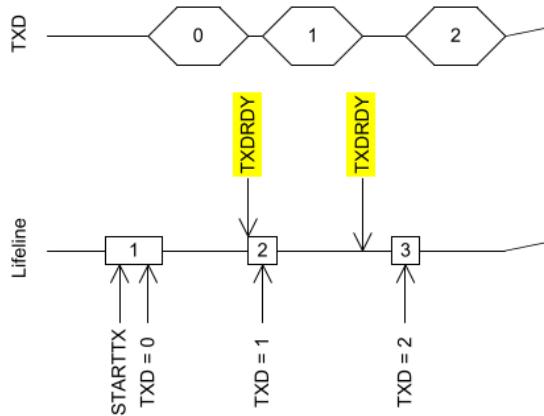


Figur 5: Sending av byte over UART.

I C vil dette oversette til å simpelthen skrive til “TxD”:

```
void uart_send(uint8_t byte){
    UART->STARTTX = 1;
    UART->TxD = byte;
}
```

Dette går absolutt strålende, så vi fortsetter med neste setning; “*When a byte has been successfully transmitted the UART will generate a TXDRDY event [...]”*. Nok en gang kjenner vi igjen dette i figur 68 i databladet, gjengitt her som figur 6.



Figur 6: TXDRDY skjer når et byte er sendt.

Hver gang Nordic snakker om en “event”, er det bare at hardware har satt et 1-tall inn i et spesielt register - som forklart i seksjon 10.1.4 i databladet. Vi ønsker å vente til bytet vårt er sendt, så dette oversettes til C-kode slik:

```

void uart_send(uint8_t byte){
    UART->STARTTX = 1;
    UART->TXD = byte;

    while(!UART->TXDRDY);
    UART->TXDRDY = 0;
}

```

Forresten, hvis dere ikke allerede har sett det før, så er et semikolon etter en **while** eller **for** det samme som en tom løkkekropp.



Legg merke til at vi manuelt setter TXDRDY til null igjen etter at hardware satte den høy. Dette er nødvendig, som forklart i seksjon 10.1.4 i databladet.

Skru alltid “events” av igjen etter at dere har ventet til de går høye.

Riktig så smooth. Nesten i mål. Neste setning er “*A UART transmission sequence is stopped immediately by triggering the STOPTX task*”. Easy, vi vet allerede hvordan vi aktiverer “tasks”, så C-koden blir slik:

```
void uart_send(uint8_t byte){  
    UART->STARTTX = 1;  
    UART->TXD = byte;  
  
    while(!UART->TXDRDY);  
    UART->TXDRDY = 0;  
  
    UART->TXDSTOP = 1;  
}
```

Protip: Det er lurt å legge inn litt *whitespace* mellom ting som er logisk samhørende. Det er selvsagt litt smak og behag inn i bildet, men mitt argument for plasseringen av mine mellomrom er slik:

1. Først starer vi en overføring.
2. Deretter venter vi til den er ferdig.
3. Til slutt går vi ut av overføringsmodus.

Dere velger selv om dere kjøper mitt argument for at dette gjør koden mer oversiktlig eller ei.

Ok, la oss starte på neste setning. Den sier “*If flow control is enabled a transmis...*” Nei! Stopp! *Flow control ain't enabled son!*

Vi skrudde eksplisitt av CTS og RTS tidligere, så denne setningen er ikke relevant. Dette er også grunnen til at vi har nektet blankt å ofre noe som helst oppmerksomhet til CTS-linjen i figur 68 i databladet. Mye av kunsten bak å lese et datablad er å lese de seksjonene du trenger *veldig* grundig, men på den annen side må du også kunne identifisere informasjon som ikke er viktig for deg, sånn at du kan aktivt gå inn for å ignorere den.

Ideelt sett bør du være i stand til å gi uviktig informasjon mindreverdighetskomplekser bare ved å ikke engang se på den.

3.5 Kjør programmet

Når “uart.c” er ferdig utfyldt, navigerer dere til mappen “2_uart”, hvor dere kaller “make” og “make flash”. Så åpner dere en ny terminal og kaller “picocom -b 9600 /dev/ttyACM0”.

Når picocom er oppe og kjører, kan dere trykke på de to knappene på micro:biten. Da skal bokstavene “A” og “B” komme opp i terminalen.

For å avslutte picocom er det Ctrl+A, etterfulgt av Ctrl+X.

3.6 Funker ikke picocom?

Hvis picocom klager på at dere ikke har tilstrekkelige rettigheter, kan det være fordi dere ikke har tillatelse til å lytte til “/dev/ttyACM0”, dette bør løse seg ved å legge til “sudo” foran picocom.

Hvis picocom sier noe som “*FATAL: [...] No such file or directory*”, så er ikke micro:biten koblet til “/dev/ttyACM0”. No worries, dette har skjedd før, og det *sannsynligvis* ingenting galt med micro:biten deres. Den er bare litt sjener. Her er løsningen:

1. Koble først ut micro:biten.
2. Åpne en ny terminal, hvor dere kaller “dmesg --follow”.
3. Koble i micro:biten.
4. Dere skal nå komme opp en melding om en ny USB-enhet. Se figur 7:

```
[363662.406088] usb 2-2: new full-speed USB device number 13 using xhci_hcd
[363662.547360] usb 2-2: New USB device found, idVendor=1366, idProduct=0105, bcdDevice= 1.00
[363662.547363] usb 2-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[363662.547364] usb 2-2: Product: J-Link
[363662.547366] usb 2-2: Manufacturer: SEGGER
[363662.547367] usb 2-2: SerialNumber: 000789817999
[363662.550669] cdc_acm 2-2:1.0: ttyACM0: USB ACM device
```

Figur 7: Denne micro:biten fikk navnet “ttyACM0”.

5. Ta navnet som micro:biten fikk av operativsystemet, her “ttyACM0”, og prøv det etter “/dev/” i picocom.

3.7 Ekstra

Dere har også fått utlevert en komplett implementasjon av “uart_receive” som dere kan leke med hvis dere vil sende ting til micro:biten fra datamaskinen. Denne funksjonen tar inn en peker til lagringsplass for bytet den potensielt mottar fra datamaskinen, så et eksempel på bruk ser slik ut:

```
uint8_t received_byte;

if(uart_receive(&received_byte)){
    /* received_byte contains valid data now */

    if(received_byte >= 'a' && received_byte <= 'z'){
        received_byte += 'A' - 'a';
    }

    uart_send(received_byte);
    uart_send('\n');
    uart_send('\r');
}

else{
    /* received_byte is undefined here */
}
```

4 Two Wire Interface

På micro:biten har vi også en “ordentlig ekstern” chip; en LGM303AGR. Denne kombinerer et akselerometer og et magnetometer. Ved hjelp av denne chipen har vi plutselig tilgang til informasjon om orienteringen til micro:biten. LSMen har naturlig nok sitt eget datablad, som også ligger utlevert.

De to chipene kan snakke med hverandre over en protokoll kalt “Two Wire Interface” (TWI) eller “Inter-Integrated Circuit”. (I^2C). Bakgrunnsstoff som hvordan denne protokollen fungerer kan finnes i et eget appendiks i den opprinnelige oppgaveteksten.

For å ikke gjøre denne oppgaven for omfattende, har dere fått utlevert en komplett implementasjon av selve TWI-driveren. Dere trenger kun å lese litt i databladet til LSM303AGR'en for å finne noen adresser som dere skal skrive til- og lese fra.

4.1 Hvordan gjør LSM303AGR TWI?

Vi starter med å lese gjennom seksjon 6.1 i databladet “lsm303agr.pdf”. Igjen er kunsten å kun lese det vi trenger å vite. Det meste her er “vanlig I^2C ”, med ett eneste unntak.

Når man leser flere registre over TWI/ I^2C , er det vanlig at man starter på en eller annen vilkårlig adresse, og at hver ekstra leseoperasjon gir deg det neste påfølgende registeret. På denne chipen må vi endre litt på adressene vi prøver å lese for å oppnå denne funksjonaliteten. Dette er forklart i seksjon 6.1.1, i tredje avsnitt, i siste setning. Om dere bare leser én eneste setning i dette databladet, så sorg for at dette er setningen dere leser.

4.2 Finn korrekte slaveadresse

Siden LSMen har både et akselerometer og et magnetometer internt, har den også to forskjellige slaveadresser. Vi er ute etter adressen til akselerometer i denne oppgaven, som kan finnes i seksjon 6.1 i databladet. Noter ned denne adressen i heksadesimal.

Grunnen til at vi ønsker adressen i heksadesimal istedenfor i binærrepresentasjon er at ting som “0b00011000” ikke er en del av C-standarden. Det vil fungere hvis dere kompilerer med GCC, fordi GCC implementerer dette som en “compiler extension” - men det er ikke garantert å fungere andre steder.

Heksadesimal er dessuten mer oversiktlig enn binær.

4.3 Finn registeradressene til akselerometeret

Når dere har funnet akselerometeret sin adresse, skal dere også finne adressene til akselerometerets interne registre. Gitt hvor mange ganger dere har lest ”adresse” i denne oppgaveteksten begynner dere forhåpentligvis å sette pris på at pekere kan være ganske så nyttige :)

Dere er ute etter adressene for CTRL_REG1_A til CTRL_REG6_A. Dette bør være ganske rett frem å finne i seksjon 7 i LSM303AGR-databladet. Suffikset ”_A” står forøvrig for ”accelerometer”, så hvis dere ser et register som slutter på ”_M” er det bare å ignorere det.

Til slutt må dere finne adressen til akselerometermålingene. Dette starter på et register som heter ”OUT_X_L_A”, for ”X low byte”. Det er derimot her dere må trikse litt med adresser for å få til automatisk inkrementering ved leseoperasjoner. Les seksjon 6.1.1 om igjen hvis dere ikke tok den.

4.4 Substituer og kjør

Når dere har funnet det dere trenger, kan dere navigere til mappen ”3_twi/source” og putte inn de riktige verdiene i ”accelerometer.c”.

Deretter kaller dere som vanlig ”make” og ”make flash” fra mappen ”3_twi”.

Når dere nå åpner picocom som før, skal dere se verdier fra akselerometeret i terminalen.

5 Interrupts

Denne seksjonen inneholder ikke noen oppgave - den bare forklarer litt om hvordan *interrupts* fungerer på ARM-baserte prosessorer, og hvordan vi kan bruke interrupts på nRF51en. Hvis dette ikke virker interessant, så er det bare å hoppe videre til neste seksjon.

5.1 Hva er en interrupt?

En interrupt kalles av og til i ARM-sirkler en “exception”, og med god grunn. Det er noe som tar deg ut av vanlig programflyt ved å rett og slett bare avbryte det du holdt på med i utgangspunktet.

Dette kan være veldig kjekt hvis dere trenger å reagere på noe som skjer sporadisk, eller hvis noe skjer periodisk, men dere ønsker å gjøre andre ting “i mellomtiden”.

Det finnes flere grunner til å bruke interrupts, men det finnes også mange til å unngå dem. Gode grunner til å bruke interrupts inkluderer:

- Dere bruker “SysTick” på ARM-prosessoren til å implementere tidsskritt for en *scheduler* eller et operativsystem.
- Dere ønsker at mikrokontrolleren er i “sleep” så mye som mulig, men at den skal reagere når ting skjer.
- Dere starter en prosess som tar lang til å fullføre, og det ville vært ineffektivt å *busy wait*.

Veldig dårlige grunner til å bruke interrupts på inkluderer:

- “Interrupts er kult da”.
- Det var det første du prøvde, uten å tenke etter om det er andre måter å gjøre en oppgave på.
- “Interrupts er jo alltid mer effektivt enn polling”.

Spesielt det siste er en sketchy grunn; hver gang en interrupt skjer, må prosessoren lagre hele den nåværende tilstanden, og så gå for å gjøre ett eller annet, og så komme tilbake og gjenopprette tilstanden den hadde før interrupten. Dette kalles en “context switch”, og kan gjerne klusse mer med kjøretiden deres enn dere forventer.

I tillegg til dette introduserer nødvendigvis interrupts mer ting å tenke på. Dere kan ikke lengre tenke på programmet deres som en sekvensiell greie som alltid kjører på en forutsigbar måte; en interrupt kan skje når som helst.

Dette er på ingen måte skremselspropaganda for å si at dere skal holde dere unna interrupts, det er bare nok et eksempel på at *det at dere kan ikke nødvendigvis betyr at dere bør*.

5.2 Hvordan fungerer interrupts på ARM?

ARM-baserte prosessorer tillater nøstede interrupts med forskjellige prioriteter. Det vil si at hvis en kritisk interrupt skjer, så vil den kunne *interrupte* en annen interrupt. Hvis dette skjer, så sier man at man har “*pre-emptive interrupting*”, som rett og slett betyr at de to rutinene ikke trenger å samarbeide for å bli enige om hvem som skal kjøre først.

For å styre dette, har ARM-prosessorer en innebygd enhet som heter “Nested Vector Interrupt Controller”, eller NVIC. Denne bestemmer hvilke interrupts som er skrudd på, og hvilken prioritet de kommer til å kjøre med.

Måten dere snakker med NVICen på er gjennom et forhåndsdefinert minnemappe som ARM kaller “System Control Space”, eller SCS. På vår nRF51 er dette minnemappen til området `0xe000e000 - 0xe000efff`. Måten vi vet dette på er at vi vet at nRF51 har en ARM Cortex-M0-kjerne, og en ARM Cortex-M0 har en ARMv6-M “kjernekjerne” (dette er embedded, selvsagt blir det litt clusterfuck etter hvert). Vi har ganske greit bare tatt en titt i databladet til ARMv6-Men, som forteller oss alt dette.

NVICen i seg selv starter på adresse `0xe000e100`.

Når så en interrupt inntreffer, vil ARM-kjernen bruke den interruptens “interrupt number”, og slå opp i en tabell med funksjonspekere for å se hva som skal skje. Denne tabellen kalles “Interrupt Vector Table”, og ligger alltid plassert på adresse `0x00000004`, uten unntak, for alle ARM-baserte prosessorer.

Grunnen til at tabellen alltid ligger her, er at det forenkler oppstarten av en ARM-prosessor litt. Når en ARM-prosessor starter, vil den første starte en rutine kalt “`Reset_Handler`” (som senere kaller vår “main”).

Ved å plassere denne i starten av tabellen, kan vi så simpelthen lese adresse `0x00` inn i registeret “main stack pointer”, og adresse `0x04` inn i registeret “program counter”. Og det er pretty much det vi trenger å gjøre - etter det antar vi at `startupkoden` tar over.

Hvis alt dette høres innfløkt ut, så er det bare å puste med magen. Det blir nok folk av dere, selv om dere ikke vet hvordan interrupts fungerer på

det laveste nivået. Uansett, la oss fortsette; startupkode.

5.3 Startupkode

Før “main” kjører er det en hel del ting som må skje. Dere kan ta en titt på filen “`gcc_startup_nrf51.S`” som ligger under mappen “`source/build_system`”. Dette er litt assembly som er ansvarlig for å laste inn “vårt program” når det faktisk skal kjøre.

Blant annet setter denne startupkoden opp vektortabellen for oss. Om dere blar litt ned, vil dere se noe som heter “`__isr_vector`”, som ser omtrentlig slik ut:

```
__isr_vector:  
    .long    __StackTop           /* Top of Stack */  
    .long    Reset_Handler  
    .long    NMI_Handler  
    .long    HardFault_Handler  
    .long    0                   /*Reserved */  
    .long    SVC_Handler  
    .long    0                   /*Reserved */  
    .long    0                   /*Reserved */  
    .long    PendSV_Handler  
    .long    SysTick_Handler  
  
/* External Interrupts */  
    .long    POWER_CLOCK_IRQHandler  
    .long    RADIO_IRQHandler  
    .long    UART0_IRQHandler  
    .long    SPI0_TWIO_IRQHandler  
    .long    SPI1_TWI1_IRQHandler  
    .long    0                   /*Reserved */  
    .long    GPIOTE_IRQHandler  
    .long    ADC_IRQHandler  
    .long    TIMER0_IRQHandler  
    .long    TIMER1_IRQHandler  
    .long    TIMER2_IRQHandler  
    ...
```

Her kjenner vi igjen “main stack pointer” og “Reset_Handler”, som er de første tingene som skrives inn i RAM for oss. De 15 første interruptene, fra “Reset_Handler” til “SysTick_Handler” er “interne” til ARM-kjernen. De defineres av prosessoren.

Etter disse kommer interrupts definert av Nordic, i det de designet nRF51en. Ingen av disse kan endres, men vi kan skru dem på eller av. Vi kan også definere våre egne intuerruptrutiner, slik at vi kan reagere på disse hendelsene.

De fleste symbolene i startupkoden er “weakly declared”, som betyr at selv om Nordic har laget en implementasjon av “TIMER0_IRQHandler”, står vi fritt til å omdefinere den i vår kode. Om dere tar en titt i filen “server_link.c” i mappen “4_internet/source/”, så vil dere se at det er nettopp dette vi har gjort.

Om vi gjør dette vil denne kalles dersom TIMER0 genererer en interrupt. Neste steg er dermed å skru på interrupts for denne modulen.

5.4 Skru på interrupts

Når vi først har definert en interruptrutine, må vi aktivere tilhørende interruptflagg for at noe skal skje. Når vi bruker “eksterne” enheter må dette gjøres to steder:

1. Først må vi skru på interrupten i NVICen, sånn at vi reagerer på den når den inntreffer. Dette gjøres ved å skrive til NVIC_ISER (“Interrupt Set Enable Register”), som i vårt tilfelle ligger på adresse 0xe000e100.
2. Deretter må vi fortelle TIMER0-modulen at den faktisk skal generere interrupts. Dette gjør vi ved å skrive til dens INTENSET-register.

Alle disse magiske adresse kommer alltid fra datablad. For ARMv6-M-arkitekturen, så er det databladet å finne her:

https://static.docs.arm.com/ddi0419/d/DDI0419D_armv6m_arm.pdf

For de spesielt interesserte, er det seksjon B3.4.2 de magiske adressene har sitt opphav.

6 Koble micro:biten på internett

Woot woot! Tiden er inne til gjøre “internet of things” - som egentlig er ordentlig lett. Litt rart at dette er blitt et buzzword egentlig.

Litt som “digital tvilling”. Mener du “adaptiv estimator”?

Ok, polemikk satt til side, vi skal trikse frem litt IoT fra micro:biten ved å bruke et program som oversetter UART til TCP for oss. På en større mikrokontroller som faktisk er designet for å være på internett vil dere gjerne ha en Ethernetmodul eller støtte for WiFi.

Før noen spør: Jada, du kan kjøre WiFi på 2.4 GHz - og ja, micro:biten har en antenn for 2.4 GHz. Radiomodulen til nRF51en er derimot designet for å kjøre Bluetooth Low Energy og noen andre 802.15.4-protokoller, men dessverre ikke 802.11, så det skal nok godt gjøres å implementere en WiFi-stack på akkurat denne chipen.

6.1 Get you some Elixir

Programmet som oversetter UART til TCP er skrevet i Elixir, så last det ned om dere ikke allerede har det: <https://elixir-lang.org/install.html>

Protip: Elixir er et av de språkene som er veldig bra egnet til å regelrett gruse heisprosjektet i Sanntidsprogrammering til neste år. Bare et tips ;)

6.2 Finn din hash

Når dere har lastet ned Elixir, skal dere endre litt på “`4_internet/source`”. I linje 15 i “`main.c`” oppretter vi en forbindelse med oversetterprogrammet. Endre på denne linjen sånn at den har din hash, som du kan finne i slutten av denne oppgaveteksten.

Hvis jeg for eksempel heter “Kolbjørn Austreng”, skal min linje se slik ut:

```
server_link_init("985f0e", &m_state);
```

Som basically alt annet i C, så er dette *case sensitive*.

Når dette er gjort kaller dere “`make`” og “`make flash`”.

6.3 Start oversetteren

Ok, ett steg igjen!

Åpne en terminal i mappen “4_internet/hammer”. Her kaller dere først “mix deps.get”, og så “iex -S mix”. (Stor “S”).

Hvis dere får masse utskrifter som “Server denied ID ??????, double check your C code”, så er det en grei indikasjon på at det er på tide å sjekke C-koden din. Hvis dere ikke får noe slikt, kan dere nå besøke lubemeup.sexty.

Forhåpentligvis vil dere se et rumpetroll svømme rundt og spise plankton for å overleve. Dere kan styre det med å tilte micro:biten i forskjellige retninger. Et eksempel ser dere i figur 8.



Kolbjørn (47)

Figur 8: The beginning of Kolbjørn.



Om ingenting skjer, kan det være fordi micro:biten ikke heter “ttyACM0” på deres system. Åpne “hammer/lib/hammer_uart.ex” og endre navnet i linje 38 til det den er kalt hos dere isåfall.

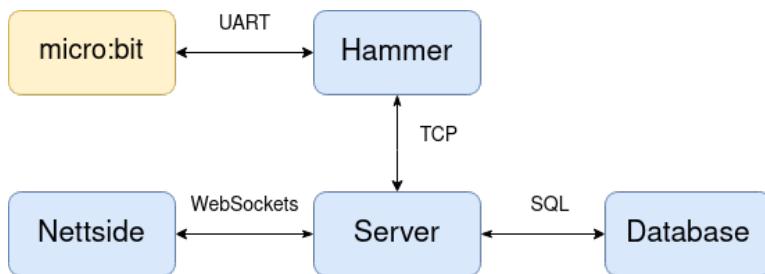
Heads up: Null innsats er lagt i å gjøre denne nettsiden skjermresponsiv, så jeg vet den ikke funker på telefoner. Jeg mistenker også at den kanskje ser litt dårlig ut på retinaskjermer, som på mac - men jeg har en negativ mengde Appleprodukter selv, så jeg får ikke testet `_(`)_/``.



Det bør være ganske åpenbart at det er rumpetroll som svømmer rundt.

De er tross alt svarte på hvit bakgrunn og ikke motsatt, HR.

For de som er interessert, kan dere se en skjematisk fremstilling av hva som foregår bak kulissene i figur 9:



Figur 9: Bestanddelene i dette prosjektet.

1. micro:biten snakker med datamaskinen over UART.
2. Datamaskinen oppretter en forbindelse med lubemeup.sexty.
3. Serveren sjekker hashen som micro:biten sendte opp mot en database.
4. Respons sendes tilbake til datamaskinen som micro:biten kommuniserer med.
5. Om hashen var godkjent, vil micro:biten sine akselerometermålinger bli sendt til serveren periodisk med 100 ms mellomrom.
6. Serveren bruker dette som rumpetrollets akselerasjon, og beregner hastighet og posisjon basert på dette.
7. Serveren sjekker også om rumpetrollet er i nærheten av et plankton, isåfall vil rumpetrollet spise planktonet.
8. En nettsleser besøker lubemeup.sexty, og oppretter i samme slengen en forbindelse med serveren over WebSockets.

9. Hver gang det skjer noe på serveren, vil nettleseren motta posisjonen til rumpetrollene som er “pålogget”, og tegne dem.
10. Når et rumpetroll kobler fra, vil poengsummen lagres til neste gang det kommer tilbake.

Dette er ikke så grusomt vanskelig å implementere heller - dere trengte jo bare å skrive litt C-kode for å gjøre alt dette, så det bør være ganske åpenbart hvor lett IoT egentlig er.

A Hasher

Navn	Hash
Aaberge, Marte Kristine	ffc047
Allum, Simen Stensrød	f35361
Alver, Morten Omholt	fd8126
Alvheim, Hanne-Grete	785b95
Andersen, Thomas	2757c6
Andreassen, Ida Marie	e42c88
Austevoll, Simon Gjengedal	706af7
Austreng, Kolbjørn	985f0e
Bakkene, Torjus	80d589
Berg, Vemund	58a354
Bjerkehagen, Daniel Berge	14d280
Bjørlo, Aurora Sletnes	792cc2
Bjørnsen, Tjerand	875f92
Blom, Kristian	93f4f7
Blomseth, Erlend	92ff31
Borgen, Kristian Novsett	a022d3
Brandis, Andreas Von	e3c7ca
Brecke, Ruben Bjarnesen	8335b0
Breirem, Lars Eik	c45766
Brendeløkken, Julianne	dfcf8a
Brenne, Håvard	c430db
Bruaset, Endre	de1f68
Budd, Philip Daniel	0c9b9a
Carlsen, Ørjan Iversen	bed64e
Christensen, Gustav Tolstrup	c5fcfc
Digerud, Ulrik Holtgaard	6aedfe
Digre, Sigurd Synnesønn	52cb5b
Dogger, Eivind	44d902

Dokken, Mari Hestetun	79b0d9
Dybdal, Camilla	04f6d8
Eide, Lotte	4beb06
Eikså, Kristoffer	a7bee8
Ellingsen, Ludvig Brannsether	54766f
Eriksen, Kjetil Holst	7e3b57
Erlandsen, Magne Johannes	9ef5bf
Fanebust, Andreas	72dad0
Fevang, Anne Berit	0666b7
Flatlandsmo, Embla	55d29b
Fleisje, Ingvild Christoffersen	aeba29
Fløtaker, Simen Piene	8805d9
Forgaard, Theodor Johannes Line	3edbfd
Fosso, Hallvard Laupsa	da0c17
Framnes, Tuva Augustin	b0774b
Frich, Johan Fredrik Skaali	5f108b
Fuglestrand, Elias	a3cca9
Furevik, Erik Rugaard	d6ead6
Fyrand, Hanne Lindbäck	234040
Føre, Martin	80e2b6
Gangstad, Simen	896094
Ghindaoanu, Nadine Adelina	8078fb
Gjerden, Marie Skogstrand	e42cdc
Gjesdal, Maria	cd17f5
Grindvik, Mathias Hatle	187977
Grønningsæter, Ola Solli	32c902
Haaland, Jørgen	79d3f8
Haga, Gry Veronika	fe99d6
Hagehei, Tobias Jacobsen	e0cfbc
Hamland, Truls	3afe5
Hansen, Hannah	d506d8

Hansen, Henrik Adrian	30b665
Haraldstad, Vegard	75d7af
Haugen, Helene Engebakken	37a81d
Haugen, Ingvild Rustad	6533d4
Haugstad, Fredrik August Bjærum	816568
Haugstvedt, Emil Johannesen	8b89ed
Haukanes, Frida Marie	b6e99a
Haver, Mina Helena Rørvik	b26f9f
Heir, Martin Borge	194bfc
Herleiksplass, Karoline Seljevoll	ab85d7
Herø, Kristin Helno	83903b
Hestnes, Henrik Albin Larsson	4124a9
Hovdar, Magne Angvik	296cf4
Hungnes, Jenny Brandal	3e61c8
Hunshamar, Asgeir	6a1ae9
Husa, Svein Jostein	98ce98
Husby, Oliver Kristiansen	8ba343
Høgstedt, Espen Berntzen	137be5
Jernsletten, Jakob Horn	b46235
Johanraj, Jananni Jiaa	ab8ae9
Jonassen, Pauline Mørch	d5d00a
Jordheim, Henrik	9ab73b
Kenworthy, Victoria Taklo	22defb
Kippersund, Yngve	a2a4c0
Kirkerud, Ola Johan Olimb	d659bc
Kolbeinsen, Magnus Isdal	71ac1c
Koushan, Shervin	8ddbc8
Kraft, Martin Andersen	7c8a75
Kunst, Valeriy	a4ca70
Kwizera, Fred Intwaza	c3acc1
Lampe, Kristian	667cd2

Lange, Henrik Kåfjord	48cb1f
Lauvrak, Jon Magnus Mathisen	2e0ea3
Lerfald, Marcus Lundeby	352420
Lie, Erling Syversveen	e8077e
Lie, Camilla	018a8c
Lille, Hanna Berggrav	213482
Linnerud, Mari	bb2b79
Loftesnes, Hanne Karine	cf09e0
Lu, Rose	7a3c9d
Lunde, Trym Overrein	8fde66
Løkken, Sandra Garder	9ce2e4
Lønvik, Helene Tørlen	5c297e
Løvlie, Bendik	0151d8
Mæhlum, Magnus	009827
Markusson, Maja Simons	943c20
Martinsen, Kristoffer	1fd2f2
Mathisen, Mats	765211
Medina, Mikael Andreas	be0544
Melheim, Håvard	c133e5
Misic, Aurora	55a42d
Moe, Erik Daniel Haukås	e21dfb
Mogstad, Eskil Aaning	fcf051
Myrling, Hanna Løe	d2511d
Næstby, Aksel	c59f2b
Naqibi, Siawash	5f0f94
Nilsen, Kristian	ba32e9
Nordstrøm-Hauge, Iselin Johanna	6099fa
Norland, Ingrid Holter	d3af01
Nyegaarden, Nina Valberg	938080
Nyhus, Thomas	72621e
Nylænder, Karoline Malene	2c0302

Nysæther, Torje Steinsland	7568d3
Nystein, Ola Netland	c58d95
Olslund, Jørgen Skaar	da0aaa
Ould-Saada, Eskil Berg	a6426a
Paasche, Mathias Thoresen	fb8464
Radi, Selsebil Alhoda	f12d2c
Reed, Nils	c48888
Reierstad, Håvard Pettersen	46cd44
Robertson, Max Menno	b35c88
Robstad, Erik Vincent Røgenes	48e460
Rognlien, Sondre Holth	bfa386
Rosland, Åshild Berg	418917
Ruud Olsen, Marte Nordbotten	1b8af0
Rygg, Hanne Opseth	1e9fdc
Rønquist, Kristian Joseph Alia	e36e11
Røstad, Simen Sigurdsen	45ad8e
Sande, Espen Aune	29889a
Schnell, Andrea	1f674e
Seeberg, Audun Svinø	5fdc5d
Semb, Helene	21d63c
Simonsen, Ragnhild Kvist	4d770a
Skålheim, Gunnar	a55400
Skare, Kristoffer	4f7c89
Skavlid, Jon	735825
Skipenes, Pavel Germanjuk	a354a4
Skøien, Thomas Borge	589231
Sletta, Øystein Stavnes	655cdf
Sletteberg, Jon Andreas	e830ec
Sletten, Christian Moe	b45801
Smedshaug, Even Åge	781116
Solbakken, Sander Bredvei	bca139

Solbø, Øystein	cb037e
Solheim, Amalie	f16a1b
Solheim, Robert Sætre	135855
Steinsland, Kristoffer	10c975
Steinsmo, Andreas Enodd	baf158
Stenså, Olav Austrheim	9ff13e
Storli, Henrik Senderud	caf4f8
Storrø, Anders	157ddf
Strand, Jørgen Staurvik	85ca2c
Strøm, Christopher	dd6ca7
Sundal, Aksel	205e3b
Sunde, Helene Fønstelien	77d0d0
Sundquist, Ulrik	583778
Susort, Silje	d7715e
Svendsen, Daniel Dreyer	49e5dc
Tamas, Calin Vlad	11c397
Tenold, Sina Øilo	ea881b
Torgersen, Marte Konstanse	ba1f9f
Tran, Celine Uyen Tram	39cf1f
Tronstad, Phillip Larsen	1090ee
Trovik, Sofie	dc6190
Tufte, Andreas Gudahl	a42bc7
Tvinnereim, Geir Ola Lillestøl	8ee9c3
Vaagen, Fredrik	8f6169
Vaaler, Aksel	92affb
Vangen, Aleksander Blekken	35b6f8
Vassbotn, Molly	3f16d6
Vatne, Jan Erlend	7337a8
Veggeland, Oskar Gjesdal	403257
Veglo, Guro Drange	ab0d2b
Verlo, Asbjørn Ringnes	fb6f6e

Vislie, Håkon	01538a
Voll, Laila Oftedal	01165e
Vormdal, Marcus Steffensen	d1c4b5
Wammer, Ole Andreas	cd85f0
Wardenær, Lasse Tjernæs	efd3f2
Westre, Andreas	178073
Wøllo, Vebjørn	86d73b
Ødven, Petter Knutsen	d3ad45
Øverby, Marie	fde2d9
Øversveen, Sindre Bukaasen	d267b7
Øygard, Birgit Salomonsen	0ff93c