

# Tilpassede Datasystemer

## Mikrokontrollerlab

### Kolbjørn Austreng

---



## **Revisjonshistorie**

2018	Kolbjørn Austreng
2019	Kolbjørn Austreng
2020	Kolbjørn Austreng

## i Bakgrunn

### i.1 Introduksjon til micro:bit

BBC micro:bit er et lite kort som i utgangspunktet ble utviklet for å skape interesse for programmering hos barn. For å gjøre dette mulig har man også laget et særsvennlig webgrensesnitt der man kan programmere kortet, ved å simpelthen trekke og slippe ”programmeringsklosser” der man vil. Dette er selvsagt ideelt for Datateknologi, men på Kyb ønsker vi stort sett litt mer kontroll over de lavere lagene i stacken også.

Rett under nettleserprogrammering i abstraksjonsnivåstigen ligger det vi kaller micro:bit DAL (*Device Abstraction Layer*). Denne kan programmeres via MicroPython eller JavaScript, men dette vil samtidig spise opp det meste av ressurser vi har på det lille kortet. MicroPython vil for eksempel legge beslag på omlag 14 KB av de totalt 16 KB SRAM som er tilgjengelig.

For å få en mer effektiv bruk av ressursene er det også mulig å benytte seg av C++, eksempelvis via ARM sin mbed-platform. Allikevel vil de fleste detaljene om hvordan kortet fungerer være abstrahert bort, også på dette nivået.

For å få en bedre forståelse for hvordan de lavere lagene henger sammen skal vi derfor gå forbi DALen og programmere mikrokontrollerens registre direkte. Prosessoren på kortet er en ARM Cortex M0, som er integrert inn i en Nordic Semiconductor nRF51822 SoC. Dette skal vi gjøre i C, som er det laveste abstraksjonsnivået vi kan få, uten å gå over til Thumb - som er prosessorens instruksjonssett.

### i.2 Introduksjon til labformatet

Det finnes mange gode IDEer (Integrated Development Environments) for utvikling av innnevde datasystemer. Eksempelvis Keil, Atmel Studio, IAR Workbench, eller Segger Embedded Studio. Disse har en tendens til å koste en god del om man skal gjøre seriøs utvikling, og gjemmer dessuten mange detaljer for brukeren.

I denne laben vil vi derfor bruke ARM GCC som vår verktøykjede. Denne er det man kaller åpen kilde, og er helt uten begrensninger. For å ikke drukne dere i detaljer om hva som skjer i bakgrunnen, har vi allikevel laget en Makefil for dere. Denne vil bygge kildekoden for dere, sette opp riktig minnefordeling på prosessoren, og deretter skrive koden til den.

I mappen dere har fått utdelt ligger det en gjemt undermappe ved navn “`.build_system`”. Denne inneholder det som skal til for å få koden deres til å kjøre på micro:biten. Det er ikke meningen at dere skal endre noe i denne

mappen, men om dere vil forstå hvordan koden deres henger i hop med hva som skjer på kortet, er det bare å ta en titt.

I bunnen av hver oppgavebeskrivelse vil det være hint til oppgaven. Dette kan være nyttig om dere står fast.

### i.2.1 Makefil

For å bruke Makefilen, kaller dere `make` fra et terminalvindu i samme mappe som Makefilen. Dette vil kompilere koden, og gjøre den om til en “.hex”-fil som mikrokontrolleren kan kjøre.

Når dette er gjort, kan dere laste hex-filen over i programminnet til mikrokontrolleren. Dette gjøres ved å kalle `make flash`.

I tillegg til disse målene, har denne Makefilen også to andre mål; `make erase` vil slette minnet til mikrokontrolleren, mens `make clean` vil slette ferdigkompilert kode og hex-filen fra datamaskinen.

### i.2.2 Programmeringstaktikk

For å sette ønskede registre på nRF51822en, vil vi bruke et kjent triks fra C-programmering; vi vil lage structer som dekker nøyaktig det minnet vi ønsker å fikle med - og så typecaste en peker til starten av minnet inn i structen. Dermed kan vi endre structens medlemsvariabler, og samtidig skrive til det underliggende minnet.

Dette er definisjonen på “memory mapped IO”; vi gjør endringer som i software ser ut som vanlige lese- og skriveoperasjoner i samme minnerom som resten av programmet, men i bakgrunnen er deler av dette minnet mappet til registre hos perifære enheter. Dette er i kontrast til “port mapped IO”, hvor egne instruksjoner brukes for å gjøre operasjoner i et disjunkt minneområde fra programmet.

## i.3 Datablad

Innnevde datasystemer er ganske forskjellige fra “vanlige” datasystemer, fordi de er skreddersydde for en spesifik oppgave. Ofte må de fungere med begrensede ressurser, og gjerne over lang tid kun drevet av et knappecellebatteri. Derfor må vi glemme en del generelle ting som gjelder uavhengig av plattform, og fokusere på ting som kun gjelder plattformen vi arbeider på. Det er her datablad kommer inn.

Datablad er tung, kortfattet, teknisk dokumentasjon som beskriver nøyaktig hvordan arkitekturen vi har brukes. Å finne frem i datablad er en egen kunst

i seg selv.

Gjennom dette emnet vil **nRF51 Series Reference Manual** stort sett være databladet vi skal bruke. Dette er et godt datablad som også er forholdsvis kort (datablad kan lett bli over tusen sider), så det er en god introduksjon til måten man kan jobbe med innnevde systemer om man skal skrive egen firmware.



Datablad er hellig når det kommer til programmering av innnevde datasystemer. Bruk det flittig, og les de seksjonene dere tror er relevante nøye hver gang dere står fast.

## **ii Førstegangsoppsett**

Aller først må vi sette opp de redskapene vi trenger for å kunne programmere micro:biten som vi vil. Seksjon ii.1 går gjennom hvordan dere setter opp verktøykjeden på en datamaskin, mens seksjon ii.2 dekker hvordan dere klargjør micro:biten.

### **ii.1 Software**

På Sanntidssalen vil nok dette steget være gjort for de fleste av dere, men dersom dere har lyst til å programmere micro:biten utenfor labtidene, vil denne seksjonen gi en generell oppskrift for hvordan dere kan gjøre det.

For å se om alle verktøyene er på plass før vi starter, kaller dere først `nrfjprog --version`, og deretter `mergehex --version` fra kommandolinjen. Om begge disse svarer med en versjon, og `nrfjprog` i tillegg viser en versjon for JLinkARM, kan dere hoppe over seksjon ii.1.1.

#### **ii.1.1 nrfjprog og mergehex**

For å laste ferdigkompilert kode over på micro:biten skal vi bruke Nordic sitt flasheverktøy - `nrfjprog`. Dette installerer dere slik:

1. Gå til Nordic Semi Comand Line Tools.
2. I menyen velger dere siste versjon av Linux64.
3. Når dere har lastet ned `.tar`-filen, kan dere ekstrahere alle filene i den. Navigerer deretter til mappen med de nye filene i via terminalen.
4. Kall så `sudo dpkg -i --force-overwrite JLink_*.deb` og deretter `sudo dpkg -i --force-overwrite nRF-Command-Line-Tools*.deb`

#### **ii.1.2 arm-none-eabi-gcc**

Kompilatoren vi skal bruke er GCC for ARM. På Linuxmaskiner finnes denne utvidelsen for GCC gjerne i systempakkelageret. På Ubuntu kan dere installere denne ved å kalle

```
sudo apt install gcc-arm-none-eabi
```

## ii.2 Hardware

For at micro:biten skal kunne kommunisere med `nrfjprog` over JLinkARM, må vi oppgradere firmwaret. Dette gjør vi ved å laste ned BBC micro:bit J-Link OB Firmware<sup>1</sup> fra Segger.com. Dette er en `.hex`-fil, som vi skal putte inn på micro:biten for å aktivere JLinkARM.

1. Start med micro:biten frakoblet datamaskinen.
2. Etter at dere har lastet ned `.hex`-filen, holder dere inne “Reset” på micro:biten. Dette er knappen rett ved siden av USBen.
3. Mens dere holder knappen inne, kobler dere inn USBen.
4. Det skal nå komme opp en enhet med navn “MAINTENANCE” på datamaskinen. Dere kan nå slippe knappen.
5. Trekk `.hex`-filen inn i “MAINTENANCE”.
6. Nå vil micro:biten skru seg av- og på igjen, og oppdages på nytt av datamaskinen. Koble USBen ut, og deretter inn igjen *uten* å holde inne “Reset”.
7. Når micro:biten er koblet inn igjen kaller dere `nrfjprog -f nrf51 -e` fra terminalen.
8. LED-matrisen på micro:biten skal nå slutte å lyse.

Dere har drept micro:biten. Gratulerer.

Nå er det på tide å vekke den til live igjen. Kyb style.

---

<sup>1</sup>[www.segger.com/downloads/jlink#BBC\\_microbit](http://www.segger.com/downloads/jlink#BBC_microbit)

# 1 Oppgave 1: GPIO

## 1.1 Beskrivelse

I denne laben skal vi skru på alle LEDene i matrisen når knappen “B” trykkes, og skru dem av når knappen “A” trykkes.

Denne laboppgaven vil være litt kokebok, for å introdusere noen konsepter dere skal bruke senere. Deretter vil det bli gradvis mindre håndholding. Det kan være lurt å skumme gjennom appendikset for en rask oppsummering av hvordan vi gjør bitoperasjoner i C.

LED-matrisen på micro:biten er implementert litt annerledes enn det man kanskje tror. Istedentfor å bruke en 5x5 matrise direkte, har man implementert den som en 3x9 matrise der to av cellene ikke er koblet til en diode. Dette er illustrert i figur 1. I denne matrisen er pinne 4 til pinne 12 jord, mens pinne 13 til pinne 15 er strømforsyning. Dermed, for å få diode nummer 12 til å lyse, må P6 være trukket lav, mens P14 være høy.

## 1.2 Oppgave

Ta en titt på den vedlagte filen “schematics.pdf”; dette er referansedesignet for en micro:bit. Finn ut hvordan de to knappene er koblet. Hvilke pinner på nRF51822en brukes? Vil pinnene være høye eller lave dersom knappene trykkes?

Se deretter i databladet til nRF51-serien. Hvordan ser minnekartet for mikrokontrolleren ut? Hva er baseadressen til GPIO-modulen? Bytt ut `--GPIO_BASE_ADDRESS--` i mainfilen med den faktiske baseadressen.

I mainfilen vil dere se at det er definert en struct ved navn `NRF_GPIO_REGS`. Denne structen representerer alle registrene til GPIO-modulen. Ved å *typecaste* adressen til GPIO-modulen inn i structen, kan vi så endre på structens medlemsvariabler for å skrive til registrene. Det er nettopp dette som er formålet med kodelinjen

```
#define GPIO ((NRF_GPIO_REGS*)__GPIO_BASE_ADDRESS__)
```

Når denne er definert, kan vi eksemplvis endre OUT-registret ved å kalle

```
GPIO->OUT = desired_value;
```

Dere vil også se at medlemsvariabelen `RESERVED0` er en array av type `volatile uint32_t` med 321 elementer. Dette er fordi databladet forteller oss at OUT-registeret har en offset på  $0x504$  ( $504_{16}$ ) fra GPIO-modulens baseadresse.  $504_{16}$  er det samme som  $1284_{10}$ . Altså er det 1284 byte mellom baseadressen og OUT-registeret. Siden vi bruker en ordstørrelse på 32 bit, deler vi

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

a) micro:bit LED matrise

0	2	4	19	18	17	16	15	11	P13
14	10	12	1	3	23	21			P14
22	24	20	5	6	7	8	9	13	P15

P4    P5    P6    P7    P8    P9    P10    P11    P12

b) Implementasjon i hardware

Figur 1: Utseende av- versus implementasjon av micro:bit LED matrise.

dette tallet på fire, ettersom 32 bit er 4 byte. Dermed har vi  $1284/4 = 321$ .

Ved å følge samme resonnement, hva skal da `__RESERVED1_SIZE__` være? Finn ut dette, og endre mainfilen tilsvarende.

Når dere har gjort det, kan dere fylle ut de manglende bitene av `main()`, slik at LED-matrisen lyser når vi trykker knapp B, og skrur seg av når vi trykker knapp A.

### 1.3 Hint

1. nRF51822 har en GPIO-modul, og en GPIOTE-modul (**GPIO Tasks and Events**). Sistnevnte brukes for å lage et hendelsesbasert system. Vi skal i første omgang kun bruke GPIO-modulen.
2. Om dere skriver inn GPIO-modulens baseadresse i base 16 (hexadecimal), må dere huske “0x” foran adressen. Hvis ikke vil kompilatoren tro dere mener base 10.
3. Når dere skal finne `__RESERVED1_SIZE__`, så husk at DIRCLR starter på 0x51C, som betyr at den byten slutter på 0x51F. Altstå så starter ikke RESERVED1 på 0x51C, men på 0x520.

## 2 Oppgave 2: UART

### 2.1 Beskrivelse

I denne oppgaven skal vi sette opp toveis kommunikasjon mellom datamaskinen og micro:biten. Dette skal vi gjøre ved å bruke UART, som står for **Universal Asynchronous Receiver-Transmitter**. Tradisjonelt ble signalene sent mellom to UART-moduler båret over et RS232 COM grensesnitt. Nå har det seg derimot slik at moderne datamaskiner ikke lages med en slik port. Labdatamaskinene på Sanntidssalen har en DSUB9-port, som vi kunne brukt, men i dette tilfellet er vi heldigere enn som så.

Om dere ser etter på micro:biten, vil dere se en liten chip med nummer M26M7V. Dette er faktisk en ekstra Freescale MKL26Z128VFM4 mikrokontroller. Grunnen til at denne er der, er først og fremst at den lar oss programmere nRF51822-SoCen over USB. I tillegg til dette implementerer den en USB CDC (**C**ommunications **D**evice **C**lass), som lar oss “pakke inn” UART-signaler i USB-pakker. På den måten vil datamaskinen se ut som en UART-enhet for mikrokontrolleren; og mikrokontrolleren vil se ut som en USB-enhet for datamaskinen.

### 2.2 Kort om UART på nRF51 og micro:bit

Modulen for UART som finnes på nRF51822-SoCen implementerer såkalt *full duplex* med automatisk flytkontroll. Full duplex betyr simpelthen at UARTEn er i stand til å både sende- og motta meldinger samtidig. For å tillate full duplex, trengs det naturlig nok en dedikert linje for å motta; og en dedikert linje for å sende. Flytkontrollen består av to ekstra linjer, som brukes for å avtale når en enhet kan sende, og når den må holde kjeft.

Dermed er vi totalt oppe i fire linjer: RXD (Mottakslinje), TXD (Sendelinje), CTS (Clear To Send), og RTS (Request To Send). Når alle disse linjene brukes, er det mulig å oppnå en pålitelig overføringshastighet på 1 million bit per sekund. Dette er ganske imponerende med tanke på at “vanlig” UART-hastighet ligger på 115200 bit per sekund.

Uheldigvis er ikke alt bare fryd og gammen, rett og slett fordi vi blir tvunget til å snakke gjennom Freescalechipen om vi ønsker å kunne tolke signalet som USB. Grunnen til at dette er problematisk, er at micro:biten bare kobler to UART-linjer mellom de to chipene, så vi må holde oss til UART uten flytkontroll. Dette betyr at den høyeste baudraten vi pålitelig kan sende med, er 9600 bit per sekund. Stort sett vil micro:biten kunne håndtere høyere rater enn dette, men om dere ønsker minimalt med pakketap, er det lurt å holde seg til 9600 baud. Forutsatt at dere setter pakkestørrelsen til 8 bit, og

bruker 2 stoppbit, vil dette uansett gi dere en overføringshastighet på omlag 800 bokstaver per sekund - som burde være mer enn nok.

### 2.2.1 Noe å ha i bakhodet

På nRFen er det nyttig å tenke på UARTmodulen som en tilstandsmaskin, der den vil sende så lenge den er i tilstanden STARTTX. Den vil bare stoppe å sende når den forlater denne tilstanden, altså når den går over i STOPTX. Dette er veldig godt illustrert i figur 68 i referansemanualen.

## 2.3 Oppgave

Det første vi må gjøre, er å identifisere hvor UART-pinnene er koblet. Både nRF51- og nRF52-serien er i stand til å koble hver enkelt modul til et vilkårlig sett av GPIO-pinner, så om vi designet kretsen selv, hadde dette vært opp til oss.

Slik er det altså ikke, så vi må ta en titt i “schematics.pdf”. Finn ut hvilken pinne fra nRF51822-chipen som er TGT\_RXD, og hvilken pinne som er TGT\_TXD. Disse pinnene skal vi sene konfigurere som henholdsvis input og output.

Opprett nå filene “uart.h” og “uart.c”. Headerfilen skal inneholde deklarasjonen til tre funksjoner:

```
void uart_init();
void uart_send(char letter);
char uart_read();
```

Denne headerfilen inkluderes fra mainfilen. I implementasjonsfilen (“uart.c”) skal vi igjen bruke en struct til minneoperasjoner, slik vi gjorde for GPIO:

```
#include <stdint.h>

#define UART ((NRF_UART_REG*)0x-----)

typedef struct {
    volatile uint32_t ...;
} NRF_UART_REG;
```

Dere må også legge til “uart.c” bak **SOURCES** := `main.c` i Makefilen, slik at kompilatoren kompilerer denne filen også.

### 2.3.1 void uart\_init()

Denne funksjonen skal først konfigurere de nødvendige GPIO-pinnene som input/output. Derfor må “gpio.h” inkluderes i “uart.c”. Denne filen er

allerede skrevet for dere.

Når pinnene er konfigurert i GPIO-modulen, må de brukes av UART-modulen. Dette gjøres via PSELTXD- og PSELRXD-registrene.

Om dere ser i “schematics.pdf”, vil dere se at vi ikke har noen CTS- eller RTS-koblinger fra nRF51-chipen (Clear To Send og Request To Send). Dette betyr at vi ikke er i stand til å gjøre flytkontroll i hardware. Dermed står vi i fare for å miste pakker om vi prøver å sende for fort. Velg derfor en baudrate på 9600. “Baudrate” forteller simpelthen hvor mange bit vi sender per sekund.

Av samme grunn må vi fortelle UART-modulen at vi ikke har CTS- og RTS-linjer. Sett opp de riktige registrene for dette.

Til slutt skal vi gjøre to ting. Først må vi skru på UART-modulen, som gjøres via et eget ENABLE-register. Deretter skal vi starte å ta imot meldinger, sett derfor STARTRX til 1.

### 2.3.2 `void uart_send(char letter)`

Denne funksjonen skal ta i mot én enkel bokstav, å sende den over til datamaskinen. Ta en titt på figur 68 (“UART Transmission”) i databladet til nRF51-serien for å finne ut hva dere skal gjøre. Husk å vente til sendingen er ferdig, før dere skrur av sendefunksjonaliteten.

### 2.3.3 `char uart_read()`

Denne funksjonen skal lese én bokstav fra datamaskinen og returnere den. Vi ønsker ikke at funksjonen skal blokkere, så om det ikke er en bokstav klar akkurat når den kalles, skal den returnere '`\0`'.

Husk at dere må gjøre dette i en bestemt rekkefølge for å kunne garantere at UART-modulen ikke taper informasjon. Det vil si, sett RXDRDY til 0 før dere leser RXD - og sørge også for å kun lese RXD én gang.

Dere skal ikke skru av mottakerregisteret når dere har lest meldingen.

## 2.4 Sendefunksjon

Programmér micro:biten til å sende '`A`' om knappen A trykkes, og '`B`' om B trykkes. For å motta meldingene på datamaskinen, skal vi bruke programmet `picocom`. Fra et terminalvindu, kaller dere:

```
picocom -b 9600 /dev/ttyACM0
```

Dette vil fortelle `picocom` at det skal høre etter enheten `/dev/ttyACM0`, med baudrate 9600.

Merk at den siste bokstaven bak “ACM” er et null-tegn.

## 2.5 Mottaksfunksjon

Lytt etter sendte pakker på micro:biten. Om datamaskinen har sendt en bokstav, skal micro:biten skru på LED-matrisen om den var av, og skru den av om den allerede var på. Husk at `char uart_read()` returnerer '`\0`' om ingenting var sendt.

For å sende bokstaver fra datamaskinen bruker vi igjen `picocom`. Standardoppførselen til `picocom` er å sende alle bokstaver som skrives inn i terminalen når det kjører. Bokstavene vil derimot ikke bli skrevet til skjermen, så dere vil ikke få noen visuell tilbakemelding på datamaskinen (om dere ikke manuelt sender bokstaven tilbake fra micro:biten).

## 2.6 Mer avansert IO

Nå har dere en funksjon for å sende over nøyaktig én bokstav av gangen; og en funksjon for å motta nøyaktig én bokstav av gangen. Om vi ønsker å sende en C-streng av vilkårlig lengde kunne vi laget en funksjon som dette:

```
void uart_send_str(char ** str){
    UART->STARTTX = 1;
    char * letter_ptr = *str;
    while(*letter_ptr != '\0'){
        UART->TXD = *letter_ptr;
        while(!UART->TXDRDY);
        UART->TXDRDY = 0;
        letter_ptr++;
    }
}
```

Denne funksjonen vil fungere, men den er ikke annet enn en “dum” variant av `printf()`. Den gjør nemlig nesten det samme som `printf`, men den har ingen av formateringsalternativene som gjør `printf` ettertraktet.

Vi vil heller inkludere `<stdio.h>` og bruke en heltallsvariant av `printf`, kalt `iprintf`. Nå har vi derimot ett problem: `printf` vil i utgangspunktet snakke med en strøm som heter `stdout`, som tradisjonelt sett peker til en terminal.

Mikrokontrolleren vår har ingen skjerm - og dessuten er ikke `printf` egentlig implementert ferdig for oss på denne plattformen. Dette er fordi vår variant

av `<stdio.h>` kommer fra et bibliotek kalt **newlib** - som er skrevet spesielt for innnevde datamaskiner.

Når `printf(...)` kalles, vil et annet funksjonskall til `_write_r(...)` skje i bakgrunnen. Denne funksjonen vil deretter kalle `ssize_t _write(int fd, const void * buf, size_t count)`, som foreløpig ikke gjør noe. Grunnen til at denne finnes, er at den trengs for at programmet skal kompilere, men den er i utgangspunktet tom, fordi vi gir lenkeren flagget `--specs=nosys.specs`, som dere kan se i Makefilen.

Gjennom **newlib** kan vi faktisk lage mange varianter av slike skrivenfunksjoner, om vi har et komplekst innnevde system med mange skriveenheter, eller om vi har flere tråder. Denne arkitekturen har bare én kjerne - og vi vil bare bruke UART, så vi kan fint implementere en global variant av denne skrivenfunksjonen. For å gjøre det, legger vi til følgene i mainfilen:

```
#include <stdio.h>

[...]

ssize_t _write(int fd, const void *buf, size_t count){
    char * letter = (char *) (buf);
    for(int i = 0; i < count; i++){
        uart_send(*letter);
        letter++;
    }
    return count;
}
```

Merk at returtypen til `_write` er `ssize_t`, mens `count`-variabelen er av type `size_t`. Når denne funksjonen er implementert, kan dere kalle `iprintf` fra programmet deres.

For å prøve ut programmet deres, kaller dere

```
iprintf("The chemical formula for Ketamine is C%{H%}dClNO\n\r",
→ 13, 16);
```

Om `picocom` da forteller dere den kjemiske formelen til Ketamin, så har dere greid oppgaven. (Selv om dere ikke er i stand til å lage ordenlige *subscript* i `picocom`.)

## 2.7 Frivillig: `_read()`

Vi kan også implementere funksjonen `ssize_t _read(int fd, void *buf, size_t count)`, slik at vi kan bruke `scanf` fra `<stdio.h>`. Legg til denne funksjonen i mainfilen:

```

ssize_t _read(int fd, void *buf, size_t count){
    char *str = (char*)(buf);
    char letter;

    do {
        letter = uart_read();
    } while(letter == '\0');

    *str = letter;
    return 1;
}

```

Skriv deretter et kort program som spør datamaskinen etter 2 heltall. Disse skal leses inn til micro:biten, som vil gange dem sammen, og sende resultatet tilbake til datamaskinen.

## 2.8 Hint

1. Det skal være totalt 11 reserverte minneområder i UART-structen. De skal ha følgene størrelser: 3, 56, 4, 1, 7, 110, 93, 31, 1, 1, 17.
2. Det er ingen forskjell på *tasks*, *events* og vanlige registre. Når LSB er satt i et *event*-register, har en hendelse skjedd. Når LSB settes i et *task*-register, startes en oppgave.
3. `int fd i _read og _write` står for “file descriptor”. Den er der i tilfelle noen vil bruke `newlib` i forbindelse med et operativsystem. Vi trenger derimot ikke tenke på det.
4. Husk å legge til “uart.c” i Makefilen, bak `SOURCES : = main.c`.

## 3 Oppgave 3: GPIOTE og PPI

### 3.1 Beskrivelse

Nå har det seg slik at vi jobber på en ganske ressursbegrenset plattform, som stort sett er tilfellet når vi driver med innveide datasystemer. For eksempel er nRF51822 SoCen basert på en ARM Cortex M0, som bare har én kjerne. Derfor har vi ikke mulighet til å kjøre kode i sann parallel. Vi kan selvsagt bytte veldig fort mellom to eller flere *fibre*, men dette vil skape en del problemer om vi trenger nøyaktige tidsverdier.

For å løse dette problemet, har nRF51822en noe som kalles **Programmable Peripheral Interconnect**. Dette er en teknologi som lar oss direkte koble en perifer enhet til en annen, uten at vi trenger å snakke med CPUen. For å dra nytte av denne teknologien må vi innføre *oppgaver* og *hendelser* - eller *tasks* og *events*. Oppgaver og hendelser er egentlig bare registre, men brukes annerledes enn "vanlige" registre.

Om et hendelsesregister inneholder verdien 1 har en hendelse inntruffet; om registeret inneholder verdien 0, har hendelsen ikke inntruffet. Oppgaveregistre er knyttet til en gitt oppgave (derav navnet), som kan startes ved å skrive verdien 1 til det. Oppgaver kan derimot ikke stanses ved å skrive verdien 0 til samme register som startet oppgaven.

De fleste perifere enhetene som finnes på nRF51822 har noen form for oppgaver og hendelser. For å knytte oppgaver og hendelser til GPIO-pinnene, har vi en egen modul kalt **GPIOTE** - som står for **General Purpose Input Output Tasks and Events**.

I denne oppgaven skal vi bruke GPIOTE-modulen til å definere en hendelse (A-knapp trykket), og tre oppgaver (skru på eller av spenning til de tre LED-matriseforsyningene).

### 3.2 Oppgave

Først må jordingspinnene til LED-matrisen konfigureres som output, og settes til logisk lave. Dere trenger ikke konfigurere forsyningspinnene, fordi GPIOTE-modulen vil ta hånd om dette for dere. På samme måte slipper dere å konfigurere A-knappen som input.

Dere har allerede fått utlevert headerfilene "gpiote.h" og "ppi.h", men dere må selv lese kapitlene om GPIOTE og PPI for å se hvordan de skal brukes. Når dere har gjort det, skal dere gjøre følgende:

### 3.3 GPIOTE

Alle de fire GPIOTE-kanalene skal brukes. Bruk én kanal til å lytte til A-knappen. Denne kanalen skal generere en hendelse når knappen trykkes - altså når spenningen på GPIO-pinnen går fra høy til lav.

De tre resterende kanalene skal alle være konfigurert som oppgaver, og koblet til hver sin forsyningsspinne for LED-matrisen. Forsyningsspenningen skal veksle hver gang oppgaven aktiveres. Hvilken initialverdi disse GPIOTE-kanalene har, er opp til dere.

### 3.4 PPI

For å koble A-knapphendelen til forsyningsoppgavene, trenger vi tre PPI-kanaler; en for hver forsyningsspinne. Som dere så i databladet, kan hver PPI-kanal konfigureres med en peker til en hendelse, og en peker til en oppgave. Fordi vi lagrer pekerene i registre på hardware, må vi typecaste hver peker til en `uint32_t`, som demonstrert her:

```
PPI->PPI_CH[0].EEP = (uint32_t)&(GPIOOTE->IN[3]);  
PPI->PPI_CH[0].TEP = (uint32_t)&(GPIOOTE->OUT[0]);
```

Denne kodesnutten vil sette registeret `EventEndPoint` for PPI-kanal 0 til adressen av `GPIOOTE-IN[3]` - typecastet til en `uint32_t`. Tilsvarende vil den sett registeret `TaskEndPoint` for PPI-kanal 0 til adressen av `GPIOOTE->OUT[0]`, etter å ha typecastet denne til en `uint32_t`.

Denne koden kan være litt kryptisk første gang man ser den, men om man bare tar seg tid til å lage en mental modell av hvor hver peker går, så vil man se at den egentlig er rett frem.

### 3.5 Opphold CPUen

Når den ene GPIOTE-hendelsen er koblet til de tre GPIOTE-oppgavene via PPI-kanalene, så skal LED-matrisen veksle mellom å være av eller på, hver gang A-knappen trykkes - uavhengig av hva CPUen gjør. Lag en evig løkke der CPUen ikke gjør noe nyttig arbeid.

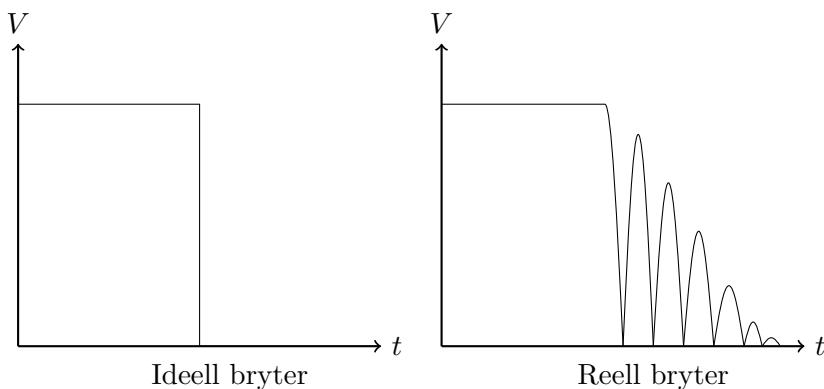
Når dere har kompilert og flashet programmet, skal LED-matrisen fungere som beskrevet. Det kan allikevel hende at matrisen ved enkelte knappetrykk blinker fort av og på, eller ikke veksler i det hele. Grunnen til dette er et fenomen kalt *inputbounce*.

#### 3.5.1 Debouncing av input

I en perfekt verden ville spenningen til A-knappen sett ut som spenningskurven til den ideelle bryteren i figur 2. I virkeligheten vil de mekaniske platene

i bryteren gjentatt slå mot-, og sprette fra hverandre. Når dette skjer, får vi spenningskurven illustrert for den reelle bryteren i figur 2. I dette tilfellet kan CPUen registrere spenningstransienten som raske knappetrykk.

Stort sett er det to grunner til at dette ikke er et problem. Først og fremst er *tactile pushbuttons* - knappene som finnes på micro:biten - mye bedre på å redusere bounce enn andre typer knapper. Den andre grunnen er at når du manuelt sjekker knappeverdien i software, vil CPUen gjerne ikke være rask nok til å merke at transienten er der. Dette er grunnen til at dere sannsynligvis ikke hadde dette problemet når dere brukte GPIO-modulen.



Figur 2: Spenningen over en ideell- og en reell bryter.

For å komme rundt dette problemet kan man gjøre *debouncing* i enten software eller hardware. I hardware ville man lagt til en RC-krets til knappen, slik at spenningen blir lavpassfiltrert før mikrokontrolleren leser den. I software kan man simpelthen vente i kort stund om man først leser en endring, slik at man hopper over transienten.

I vårt tilfelle er knappene bare trukket høye med en pullup, og det er ikke noe vi kan gjøre for å endre på det. Siden vi i denne oppgaven koblet A-knappen direkte til LED-matrisen via GPIOE og PPI, har vi heller ikke mulighet til å legge inn software debouncing; så her får vi bare leve med inputbounce.

Strengt talt, kunne vi koblet knappen fra GPIOE inn i en TIMER-instans via PPI, og deretter brukt en ny PPI-kanal til å koble en overflythendelse fra TIMER-instansen inn på GPIOE-oppgavene, men dette er mye innsats for en marginal forbedring.

### 3.6 Hint

1. Husk å aktivere hver PPI-kanal. Når de er konfigurert riktig, aktiveres de ved å skrive til **CHENSET** i PPI-instansen.
2. GPIOTE-kanalene trenger ingen eksplisitt aktivering fordi **MODE**-feltet i **CONFIG**-registeret automatisk tar hånd om pinnen for dere.

## 4 Oppgave 4: Two Wire Interface

### 4.1 Beskrivelse

En av de vanligste protokollene for seriell kommunikasjon er **I<sup>2</sup>C** (av og til formattet som “I2C”), som står for **Inter-Integrated Circuit**. Denne protokollen ble utviklet av Philips Semiconductor (i dag NXP Semiconductors), som naturligvis låste ned protokollen bak en del registrerte varemerker.

Av den grunn begynte andre produsenter å implementere protokollen under et par andre navn, hvor det mest vanlige er **TWI**, som står for **Two Wire Interface**. Av og til kan dere også komme over navnet **USI**, som står for **Universal Serial Interface** - men dette navnet er ikke like vanlig, fordi det stort sett brukes om moduler som kan operere både i **SPI**<sup>-2</sup> og **I<sup>2</sup>C**modi.

Siden 2006 kreves det ikke lenger lisens for å implementere **I<sup>2</sup>C** under navnet **I<sup>2</sup>C**, men man kommer fortsatt over navn som stort sett er synonyme.

I Nordic sitt tilfelle, implementerer nRF51serien et supersett av **I<sup>2</sup>C**; den vanlige protokollen har støtte for vanlig rate (100 kbps) og “*Fast-mode*” (400 kbps) - mens nRFen også støtter et mellommodus på 250 kbps.

Det dere skal gjøre i denne oppgaver, er å bruke **TWI**bussen til nRF51822-SoCen til å kommunisere med akselerometeret som finnes på micro:biten. Dere skal deretter bruke denne sensorinformasjonen til å lyse opp én LED i matrisen, basert på hvordan micro:biten er orientert.

### 4.2 Oppgave

Først og fremst må dere finne ut hvordan akselerometeret er koblet til nRFen. Et naturlig sted å starte er “**schematics.pdf**”. Det dere skal se etter er pinnene kalt **SDA (Serial Data)** og **SCL (Serial Clock)**.

Når dere vet hvordan ting er koblet opp, må dere finne ut av hva akselerometeret forventer skal skje på **TWI**bussen. Dette er beskrevet i databladet “**MMA8653FC.pdf**”, under seksjon 5.8. Bit dere blant annet merke i hvilken adresse akselerometeret har, og hvordan dere leser ett- og flere byte.

---

<sup>2</sup>Serial Peripheral Interface



Ikke grav dere ned i detaljer; få et overblikk først. Om dere ikke vet hva en “start condition” er, så er det nok å vite at nRFen lager en for dere. Om dere allikevel vil vite litt mer om I<sup>2</sup>C, så er appendiks B stedet å lese om protokollen.

#### 4.3 void twi\_init()

Det første som må gjøres er å lage filene “twi.h” og “twi.c”. I headerfilen deklarerer dere funksjonen `void twi_init()`. Den tilhørende implementasjonen skal aktiveres TWI-modulen på nRFen med de riktige signallinjene og 100 kbps overføringshastighet. Legg merke til at signallinjene først må konfigureres av GPIO-modulen. Dette er beskrevet i referansemanualen, under seksjon 28.

Som før, må dere først lage en struct som mapper ut adresseområdet til TWI-modulen, slik:

```
#define TWIO ((NRF_TWI_REG*)0x40003000)

typedef struct {
    volatile uint32_t STARTRX;
    [...]
} NRF_TWI_REG;
```

Denne structen er litt større enn det UART-structen fra tidligere er. Riktige verdier for de reserverte områdene er oppgitt i hintene, men prøv først å finne dem selv.

#### 4.4 void twi\_multi\_read(...)

Som dere kan se i appendiks B og i databladet til akselerometeret, er det mulig å lese- eller skrive flere byte av gangen, uten å måtte gi fra seg TWI-bussen. Dette er også mer effektivt, fordi man slipper å sende slavens adresse på nytt for hver byte. Måten I<sup>2</sup>C-enheter gjør dette på, er at de automatisk hopper til et nytt internt register hver gang du har lest fra- eller skrevet til et. Dette er beskrevet i databladet.

Istedentfor å lage funksjoner som leser- og skriver ett og ett resister, skal vi lage mer generelle funksjoner som kan lese- og skrive *n* registre av gangen. Vi kan da simpelthen sette *n* lik 1 om vi trenger kun ett.

Start med å legge denne deklarasjonen i “twi.h”:

```

void twi_multi_read(
    uint8_t slave_address,
    uint8_t start_register,
    int registers_to_read,
    uint8_t * data_buffer
);

```

Siden vi bruker typen `uint8_t`, må vi nå inkludere `<stdint.h>` i “`twi.h`”.

Med denne koden ønsker vi at det følgende skal skje: Først skal vi kunne adressere en vilkårlig slave, med adresse `slave_address`. Deretter skal vi fortelle slaven hvilket register vi ønsker å begynne å lese fra; nemlig `start_register`. Når dette er gjort, skal vi lese så mange byte vi trenger (`registers_to_read`), og putte dem i et arrayet `data_buffer`. Til slutt avslutter vi kommunikasjonen og forlater TWI-bussen. Seksjon 28.6 i nRF51-databladet forklarer veldig godt hvordan dette skal gjøres. Her har dere en oppsummering av hva som skal til:

1. Sett ADDRESS-registeret til `slave_address`.
2. Start en **skriveoperasjon**.
3. Overfør `start_register` til TWI-bussen.
4. Når dere har fått ACK tilbake fra slaven (som betyr at en TXDSENT-hendelse er blitt generert), starter dere en **leseoperasjon** uten å stoppe bussen. Dette kalles en *repeated start sequence*.
5. Les TWI-bussen (`registers_to_read - 1`) ganger. Dette er fordi dere må sende en NACK til slaven den siste gangen dere leser en byte. Hver verdi dere leser, dytter dere inn i `data_buffer`.
6. Til slutt kjører dere STOP-oppgaven, før dere leser busser for siste gang. Dette vil gjøre at nRFen genererer en NACK istedenfor en ACK, slik at slaven ikke sender ut flere byte på bussen.



Hendelen TXDSENT settes ikke automatisk til 0 når dere dytter en ny byte inn i TXD. Det samme gjelder også for RXDREADY-registeret. Derfor er det nødvendig å gjøre dette manuelt, slik:

```
TWI0->TXDSENT = 0;  
TWI0->TXD = start_register;  
while(!TWI0->TXDSENT);  
  
[...]  
  
TWI0->RXDREADY = 0;  
TWI0->STARTRX = 1;
```

#### 4.4.1 Les WHO\_AM\_I

For å prøve ut `void twi_multi_read(...)` skal vi forsøke å lese akselerometerets enhets-ID. Denne ligger lagret i et register kalt WHO\_AM\_I. Adressen til dette registeret finner dere i seksjon 6 i databladet til akselerometeret.

For å lage en buffer å dytte iden inn i, har vi flere alternativer. Siden vi kun er ute etter én byte, holder det å simpelthen ta pekeren til en `uint8_t`. For å gjøre det mer generelt til senere, skal vi bruke `malloc()`, som gjør det samme som `new` gjør i C++. For å få tilgang til `malloc()`, må dere inkludere `<stdlib.h>`. For å allokkere minne, gjør dere slik:

```
uint8_t * data_buffer;  
data_buffer = (uint8_t *)malloc(8 * sizeof(uint8_t));  
  
[...]  
  
free(data_buffer);
```

Dette vil allokkere minne til et dynamisk array av 8 `uint8_t`. Tilsvarende C++ sin `delete`-operator, bruker vi `free()` i C for å frigi dynamisk allokkert minne tilbake til heapen.

For å vite at TWI-bussen fungerer, kan dere sammenligne det dere får tilbake med akselerometerets fabrikk-ID; nemlig `0x5A`, eller 90 i base 10. Om dette er det dere får, kan dere for eksempel skru på LED-matrisen, eller sende en melding over UART for å signalisere at ting virker som det skal.



Hver gang dere kaller `make flash`, tilbakestilles bare nRFen på micro:biten. Akselerometeret og magnetometeret lever helt sine egne liv. Det betyr at om dere sender en vanskapt TWI-meldig, kan dere sette disse i en tilstand hvor de ikke svarer på korrekte TWI-meldinger senere. Om koden deres ser riktig ut, men ting fortsatt ikke fungerer, kan dette være årsaken. Løsningen er rett og slett å ta strømmen og prøve på nytt.

#### 4.5 `void twi_multi_write(...)`

Om dere syntes forrige deloppgave var litt vrien, er det helt forståelig. Heldigvis er det nå veldig enkelt å skrive tilsvarende funksjon for å skrive  $n$  byte til en vilkårlig slave. Faktisk er dette enklere enn å lese, fordi vi ikke må tenke på NACK-grensetilfellet fra leseoperasjonen. I “twi.h” legger dere til denne deklarasjonen:

```
void twi_multi_write(  
    uint8_t slave_address,  
    uint8_t start_register,  
    int registers_to_write,  
    uint8_t * data_buffer  
) ;
```

Sekvensen vi trenger å implementere er beskrevet i seksjon 28.5 i nRF51-databladet. Sekvensen dere skal implementere, kan oppsummeres slik:

1. Sett ADDRESS-registeret.
2. Start en skriveoperasjon.
3. Skriv `registers_to_write` antall byte til bussen.
4. Kall STOP-oppgaven.

Som før, må vi huske å manuelt sette TXDSENT til 0 etter at hendelsen er blitt aktivert:

[...]

```
TWI0->TXDSENT = 0;  
TWI0->TXD = data_buffer[n];  
while(!TWI0->TXDSENT);
```

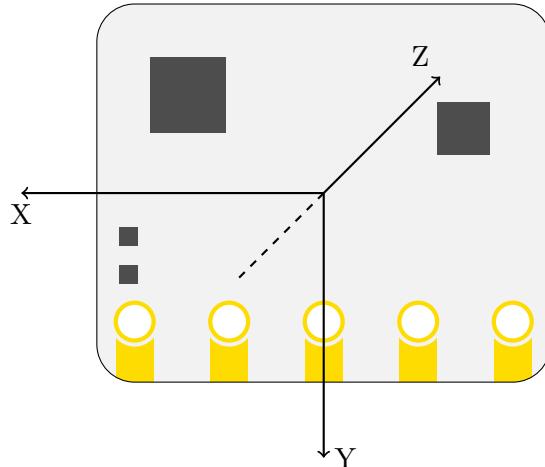
[...]

## 4.6 Les akselerometeret

Når dere har skrevet både `twi_multi_read` og `twi_multi_write`, kan disse brukes til å lese akselerometerets X-, Y-, og Z-registre. For å slippe å kalle for mange TWI-spesifikke greier direkte fra main, har dere allerede fått utlevert en *wrapper* for akselerometeret, som bruker TWI-funksjonene dere allerede har skrevet. Denne heter “`accel.h`” og deklarerer funksjonene `void accel_init()` og `void accel_read_x_y_z(int * data_buffer)`.

Funksjonen `accel_init()` vil skru akselerometeret på, og sette oppdateringsraten til 200 Hz. Denne funksjonen krever at TWI-bussen er initialisert i forkant.

Funksjonen `accel_read_x_y_z(int * data_buffer)` tar inn en peker til et array av typen `int`. Dette arrayet må minst være av størrelse 3, hvis ikke risikerer dere uforutsigbar oppførsel. Om arrayet er større enn 3 har ingenting å si. Når funksjonen er kjørt ferdig, vil de tre første elementene i dette arrayet være henholdsvis X-, Y-, og Z-komponentene til akselerasjonen akselerometeret måler. Koordinatsystemet dere får komponentene i, er illustrert i figur 3.



Figur 3: Høyrehåndskoordinatsystemet `accel_read_x_y_z` bruker.

### 4.6.1 Fyll inn konstanter i `accel.c`

Før dere kan kan bruke funksjonene i “`accel.h`”, må dere fylle ut tre verdier i implementasjonsfilen “`accel.c`”:

- `ACCEL_ADDR` skal være adressen til akselerometeret, som dere fant i databladet “MMA8653FC.pdf” under seksjon 5.8.

- ACCEL\_DATA\_REG skal være adressen til det mest signifikante bytet av akselerometerets X-komponent. Dette kan dere finne under seksjon 6 i databladet.
- ACCEL\_CTRL\_REG\_1 skal være adressen til registeret som kontrollerer dataraten, og om akselerometeret er på eller ei. Dette kan også finnes under seksjon 6 i databladet.

#### 4.6.2 Skriv verdier til skjerm via UART

Når dere har fylt inn konstantene i “accel.c”, er det fritt frem for å bruke funksjonene filen definerer. For å gjøre det lett å skrive ting til skjermen, har dere også fått utlevert “utility.h” med tilhørende implementasjonsfil. Denne filen gir dere funksjonen `utility_print`, som etterligner funksjonaliteten til `printf`, uten å trekke inn for mye annet sòl som `<stdio.h>` rasker med seg.

Dere velger selv om dere vil bruke `utility_print`, men her har dere iallfall et eksempel på hvordan funksjonen kan brukes:

```
int * data_buffer = (int *)malloc(3 * sizeof(int));
accel_read_x_y_z(data_buffer);

int x_acc = data_buffer[0];
int y_acc = data_buffer[1];
int z_acc = data_buffer[2];

utility_print(&uart_send, "X: %6d Y: %6d Z: %6d\n\r", x_acc,
→ y_acc, z_acc);
```

Dette er også dokumentert i filen “utility.h”.

#### 4.7 Lys opp riktig diode

Når dere greier å lese fra akselerometeret, er det duket for å ha det litt gøy med LED-matrisen til micro:biten. På grunn av den litt artige måten matrisen er koblet opp på, er det litt styr å skru på én og én diode. For å skjerme dere for å måtte skrive grisete kode selv, har dere fått utlevert filen “ubit\_led\_matrix.h”, som dere er mer enn velkomne til å benytte dere av. Et forslag til hvordan denne brukes, ser slik ut:

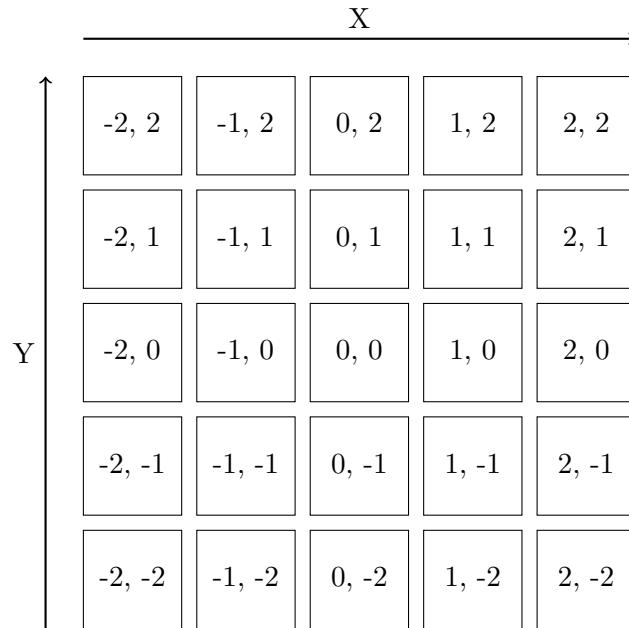
```
ubit_led_matrix_init();

[...]

// Calculate X-LED and Y-LED from accelerometer reading
```

```
ubit_led_matrix_light_only_at(x_led, y_led);
```

Koordinatsystemet `ubit_led_matrix_light_only_at(int x, int y)` bruker, er illustrert i figur 4. Hvordan dere regner ut hvilke koordinater som skal



Figur 4: Koordinatsystemet `ubit_led_matrix_light_only_at` bruker.

lyse opp gitt akselerasjon i x- og y-retning, er opp til dere. Om dere vil gjøre det enkelt, har dere en ganske rett frem metode her:

```
int x_accel = data_buffer[0];
int y_accel = data_buffer[1];

int x_dot = x_accel / 50;
int y_dot = - y_accel / 50;

ubit_led_matrix_light_only_at(x_dot, y_dot);
```

#### 4.8 Frivillig: Les magnetometeret

Med `void twi_multi_read(...)` og `void twi_multi_write(...)` er det en smal sak å lese fra magnetometeret også. Dette gjøres stort sett på samme måte som for akselerometeret, og er beskrevet i databladet “MAG3110.pdf”. Hva dere gjør med det, er opp til dere - men å bruke det som kompass er jo det åpenbare alternativet.

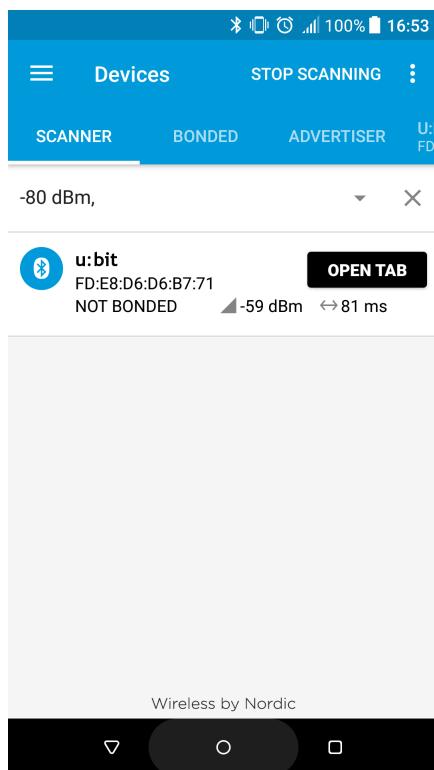
## 4.9 Hint

1. Verdier for de 15 reserverte områdene: 1, 2, 1, 56, 4, 1, 4, 49, 63, 110, 14, 1, 2, 1, 24.
2. nRF51822en har to instanser av TWI-modulen, som betyr at dere kan ha to forskjellige TWI-linjer oppå en gang. Vi trenger bare den ene.
3. Husk å manuelt sette RXDREADY og TXDSENT til 0 etter at dere sjekker disse registrene.
4. Det er egentlig unødvendig å bruke dynamisk minne i denne oppgaven. Siden vi vet ved *compile time* hvor stor plass vi trenger, kan vi simpelthen opprette et vanlig array. Vi bruker dynamisk minne bare for å eksponere dere til C-ekvivalenten av `new ;`)

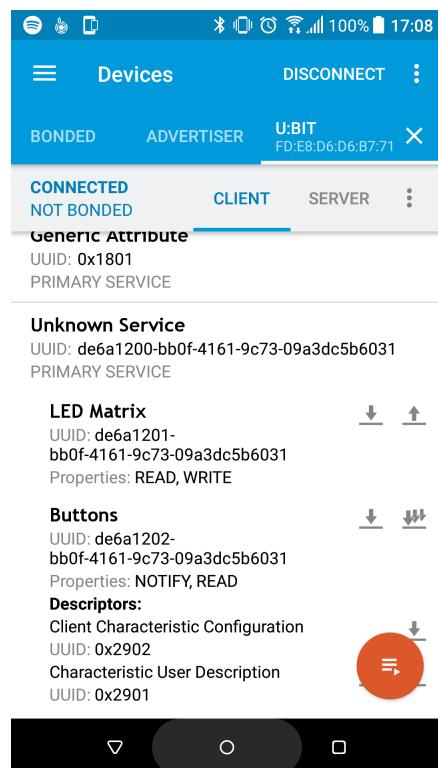
## 5 Oppgave 5: BLE GAP

### 5.1 Beskrivelse

BLE står for Bluetooth Low Energy, og er versjon 4.2 av Bluetoothspesifikasjonen. I appendiks C kan dere lese litt om Bluetooth 4.2, og forskjellene fra Bluetooth Classic. Denne oppgaven, og oppgaven om BLE GATT, er derimot ganske selvinnkapslet, så det er ikke noe krav å ha omfattende kjennskap til spesifikasjonen. Det er forøvrig ikke sunt å ha omfattende kjennskap til spesifikasjonen heller - med tanke på at siste utgave av kjerne-spesifikasjonen alene (bluetooth.com) er på rolige ~3000 sider.



Figur 5: Skanning i nRF Connect.



Figur 6: Serviceeksempler.

#### 5.1.1 Testing

For å koble til micro:biten over Bluetooth skal vi bruke "nRF Connect", som er en app dere kan laste ned fra Google Play eller Appstore. Det er utlevert et eksempelprogram, kalt `example.hex` som dere kan flashe til micro:biten for å teste. Det gjøres ved å kalle:

```
nrfjprog -f nrf51 --chiperase --program example.hex --reset
```

Når dere åpner nRF Connect vil telefonen deres starte å skanne etter *advertising packets*, som sett i figur 5. For å begrense antall enheter som vises kan dere sette RSSI-filteret, som bestemmer minimal signalstyrke før en enhet vises. I figur 5 er filteret satt til -80 dBm. På Sanntidssalen vil det være mange andre enheter i nærheten, så en verdi nærmere -60 dBm er kanskje bedre.

Når dere kobler til micro:biten vil dere se noe som minner om figure 6. Navnene “LED Matrix” og “Buttons” vil ikke vises i første omgang, men dere kan selv sette dem ved å trykke lenge på *characteristics* og redigere navnet. Uansett er navnet på karakteristikken kun kosmetisk, og påvirker ikke hvordan micro:biten oppfører seg.

Ved å laste opp et byte forskjellig fra 0x00 til karakteristikken “LED Matrix” vil LED-matrisen skru seg på. Ved å laste opp bytet 0x00 vil matrisen skru seg av igjen.

Karakteristikken kalt “Buttons” inneholder informasjon om tilstanden til knappene på micro:biten. Dere kan lese denne verdien én gang ved å trykke nedlastingsikonet; eller dere kan *subscribe* til karakteristikken ved å trykke “notifikasjonsikonet” (heltrukken linje med tre piler som peker ned). Når notifikasjoner er på, vil telefonen automatisk få beskjed når dere trykker eller slipper knappene på micro:biten.

### 5.1.2 Heads up

Et ærlig ord fra forfatteren av oppgaveteksten: Av og til kan dokumentasjonen rundt Bluetooth Low Energy være litt “Meh”. Det er helt normalt å føle at det burde være en enklere måte å lage trådløse applikasjoner på. Det viser seg derimot dessverre at radioteknologi er ganske komplisert, så bare husk å puste med magen når ting ikke ser ut til å fungere som det skal.

Do enjoy :)

## 5.2 Om GAP

Selv portvokteren i Bluetooth er kalt *Generic Access Profile*, eller GAP. Dette er delen av Bluetooth som håndterer kringkasting, oppdagelse av enheter, oppsetting av koblinger, og etablering av sikkerhetsnivåer.

Selv om GAP er involvert i å etablere koblinger, er GAP i seg selv koblingsløst, og egentlig ganske simpelt. Essensen er denne: Når du kringkaster, vil en BLE-enhet sende ut *advertising packets* på 31 byte til alle som hører på, med et intervall som kan ligge mellom 20 millisekunder og 10.24 sekunder.

I tillegg til dette kan enhver enhet som mottar en *advertising packet* forespørre mer informasjon i form av en *scan request*. Når enheten som kringkaster mottar en *scan request* kan den sende tilbake en *scan response packet*, som også er på 31 byte. Enheten er ikke pålagt å svare på *scan requests*.

GAP skjer utelukkende på kanalene 37, 38, og 39, som er reservert for kringkasting. Dette er illustrert i figur 13 under appendiks C. Grunnen til at disse kanelene ble valgt for kringkasting er at de redusere mengden interferens fra WiFi, som påvirker kanalene 0-9, 11-20, og 23-32 mest.

For å gjøre ting mer spennende, finnes det selvsagt ikke bare én måte å kringkaste på. Nyansene mellom måtene man kan sende ut *advertising packets* på er ikke viktige for oss, men de er listet her for spesielt interesserte:

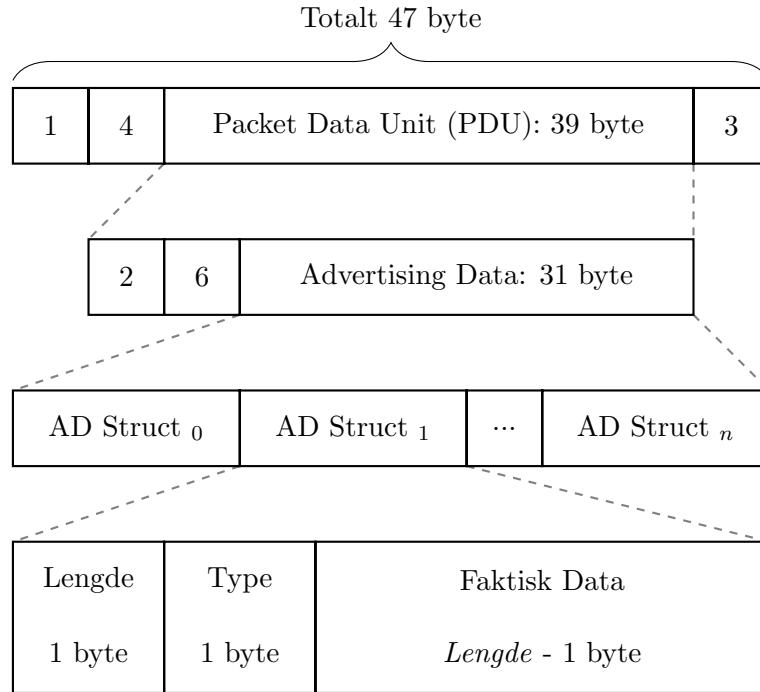
- ADV\_IND: “Connectable undirected advertising”. Dette er den vanlige måten å sende over GAP på. Enhver annen enhet kan be om en *scan response* eller en tilkobling.
- ADV\_DIRECT\_IND: “Connectable directed advertising”. Dette brukes for å være en enhet i *central mode* om en tilkobling. Denne pakken vil også kringkastes, men andre enheter som skanner vil ignorere den hvis forespurt adresse ikke samsvarer med deres egen. Enheten som sender denne pakken vil også ignorere alle *scan requests*.
- ADV\_SCAN\_IND: “Scannable undirected advertising”. Kringkasteren vil ignorere et hvert forsøk på å etablere en kobling, men kan velge å svare på *scan requests*.
- ADV\_NONCONN\_IND: “Non-connectable undirected advertising”. Dette er modien brukt av “Bluetooth Beacons”. Informasjon sendes ut i hver *advertising packet*, men kringkasteren vil ignorere alle forsøk på å etablere en kobling, og vil heller ikke svare på *scan requests*.

### 5.3 Pakkeformat

Som sagt består hver *advertising packet* og hver *scan response packet* av 31 byte. Hver pakke er arrangert i et format kalt “AD Structs” - eller “Advertising Data Structures”, som introduserer litt overhead. Dette gjør at vi har mindre data å gå på. Stort sett bruker man ikke GAP til å gjøre noen betydelig dataoverføring, så dette pleier ikke være et problem.

For å være helt korrekte, kan en kringkaster faktisk sende opp til 47 byte hvert kringkastingsintervall, men mye av dette har vi ikke kontroll over. Dette er illustrert i figur 7. Feltene som kun inneholder tall indikerer data som GAP vil sette inn, og som ikke er av interesse for oss. Vi har altså kun

de 31 byte i “Advertising Data” å leke med - og disse må struktureres i et format som spiser ytterligere 2 byte for hvert felt vi spesifiserer.



Figur 7: Innholdet i hver sending over GAP.

Selv om vi ikke fullt får brukt de 31 byte vi har tilgjengelig, må vi fortsatt spesifisere hvor mange av dem vi bruker, og hvordan de er satt opp. Koden som definerer hver kringkastingspakke i eksempelprogrammet er gjengitt i kodesnutt 1.

```
static uint8_t adv_data[] = {
    6, BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME,
    'u', ':', 'b', 'i', 't',
    3, BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_COMPLETE,
    0x0D, 0xF0
};
uint8_t adv_data_length = 11;
```

Kodesnutt 1: Datafelter “Advertising Data” fra example.hex.

Legg merke til at vi ikke trenger å fylle ut alle de 31 byte. Dette er bevisst en del av Bluetooth 4.2 spesifikasjonen, og lar oss sende enn 47

byte over lufta om vi ikke har noe spennende å si. Dette reduserer naturlig nok strømforbruket til mikrokontrolleren.

## 5.4 Oppgave

Dere har fått utdelt noen C-filer som dere kan bygge på. Om dere er en gjeng masochister som ønsker å gjøre ting helt selv, står dere fritt til å forkaste denne koden og begi dere ut på eventyr på egen hånd. Om dere verdsetter deres egen livskvalitet, så er det bare å følge med.

I `main` vil dere se et kall til `bluetooth_init()`. Dette er en wrapper rundt noen lavnivåfunksjoner som først starter Nordic sin S130 SoftDevice, for deretter å sette opp BLE stakken med noen fornuftige startparametre. Dere trenger ikke tenke for mye på dette; deres oppgave er å implementere funksjonen `bluetooth_gap_advertising_start()`, som er ansvarlig for å sette opp AD Structene (se figur 7), og så skru på kringkasting.

Første steg er naturligvis å åpne dokumentasjonen til SoftDevicen; som ligger under `softdevice/s130_nrf51_2.0.1_API/html/index.html`. Åpne denne i en nettleser. Derfra navigerer dere til `Modules`, og så `Generic Access Profile (GAP)`, så til slutt `Functions`.

Dere vil nå se en litt avskreckende ansamling funksjoner som alle er prefikset med en makro kalt “`SVCALL`”. Om dere føler dere fryktløse og virile, kan dere gyve løs på seksjon 5.4.2 med en gang. Hvis ikke, forklarer seksjon 5.4.1 hvordan dere skal tolke dokumentasjonen.

### 5.4.1 Hva i alle dager er dette for noe?

Det meste av dokumentasjonen til Nordic sine SoftDevicer kommer i form av spesielt formaterte kommentarer i headerfiler. Disse kan brukes til å automatisk generere nettsider ved hjelp av et hendig lite program kalt Doxygen. Det er en slik nettside dere nå ser på.

Doxygen gjør det enkelt å garantere at dokumentasjonen alltid er oppdatert, fordi den genereres direkte fra kommentarer i kodebasen. Den eneste ulempen er at den genererte dokumentasjonen kan bli litt kryptisk hvis man bruker mange magiske C-triks, som makroer. Det er dette som har skjedd her, men fortvil ikke - la oss disseker dette sammen:

I figur 8 ser dere et utsnitt av dokumentasjonen til GAP. Hver linje starter med makroen `SVCALL`, som står for “Supervisor Call”. Dette er en implementasjonsdetalj, og ikke noe vi har kontroll over.

<b>SVCALL (SD_BLE_GAP_DISCONNECT</b> , uint32_t, sd_ble_gap_disconnect(uint16_t conn_handle, uint8_t hci_status_code))
Disconnect (GAP Link Termination). <a href="#">More...</a>
<b>SVCALL (SD_BLE_GAP_TX_POWER_SET</b> , uint32_t, sd_ble_gap_tx_power_set(int8_t tx_power))
Set the radio's transmit power. <a href="#">More...</a>
<b>SVCALL (SD_BLE_GAP_APPEARANCE_SET</b> , uint32_t, sd_ble_gap_appearance_set(uint16_t appearance))
Set GAP Appearance value. <a href="#">More...</a>
<b>SVCALL (SD_BLE_GAP_APPEARANCE_GET</b> , uint32_t, sd_ble_gap_appearance_get(uint16_t *p_appearance))
Get GAP Appearance value. <a href="#">More...</a>
<b>SVCALL (SD_BLE_GAP_PPCP_SET</b> , uint32_t, sd_ble_gap_ppcp_set( <b>ble_gap_conn_params_t</b> const *p_conn_params))
Set GAP Peripheral Preferred Connection Parameters. <a href="#">More...</a>
<b>SVCALL (SD_BLE_GAP_PPCP_GET</b> , uint32_t, sd_ble_gap_ppcp_get( <b>ble_gap_conn_params_t</b> *p_conn_params))
Get GAP Peripheral Preferred Connection Parameters. <a href="#">More...</a>

Figur 8: Utsnitt av SoftDevicens GAP-dokumentasjon.

Det faktiske funksjonsnavnet som er tilgjengelig for oss ligger etter denne makroen. For eksempel, i linja **SVCALL(SD\_BLE\_GAP\_TX\_POWER\_SET, uint32, sd\_ble\_gap\_tx\_power\_set(uint8\_t tx\_power))** er funksjonen vi skal bruke kalt

```
sd_ble_gap_tx_power_set(int8_t tx_power);
```

Under hver linje med SVCALL er det en link kalt [More...](#), som vil ta dere til forklaringen til den aktuelle funksjonen.

Dere vil legge merke til at mange funksjoner tar inn pekere til egne structer, fremfor å ta inn flere primitive typer. Dette er igjen grunnet impletasjonsdetaljer. I mange tilfeller hvor en funksjon tar inn en peker til en struct, må structen være i live etter at funksjonen er blitt kalt. I slike tilfeller må dere enten allokkere minnet til structen fra heapen med `malloc`, eller deklarere den som `static`.



Om det ikke var klart: I enkelte tilfeller må minnet til en variabel være veldefinert etter et funksjonskall, da er `static` deres venn.

Til slutt er det verd å påpeke en del av kodekonvensjonene som brukes i SoftDevicens API:

1. Modifikatoren `const` er brukt flittig for å signalisere til kompilatoren at en variabel ikke skal endres. Dette er en god kutyme- spesielt når man skriver kode som lett kan gå i stykker, for eksempel kode til Bluetooth.
2. Pekervariabler prefikses med `p_`, og i de få tilfellene en peker til en peker er brukt, er prefikset `pp_` brukt. Dette er en god konvensjon som kan anbefales, og er en rask måte å luke ut en egen spesielt irriterende familie bugs.

- I dokumentasjonen til hver funksjon er parametrene enten markert [in], [out], eller [in,out]. Dette forteller dere om funksjonen vil modifisere argumentet eller ikke. Parametre som er markert med [in] vil kun ta en referanse, men ikke endre på dataen som blir pekt til. Parametre markert som [out] eller [in,out] vil bli endret av funksjonskallet.

#### 5.4.2 Definer kringkastingsdata

Det første dere skal gjøre er å sette opp dataen dere skal sende i hver *advertising packet*. I filen `bluetooth.c` ser dere en ufullstendig funksjon kalt `bluetooth_gap_advertise_start()`. Deres oppgave er å fylle ut `static uint8_t adv_data[]` med én eller flere *AD Structs*, som illustrert i figur 7. Hvilke *AD Structs* dere putter inn er opp til dere.

Funksjonen dere er mest interessert i heter `sd_ble_gap_adv_data_set`. Dere kan droppe å definere en *scan response packet* om dere vil, og isåfall setter dere `p_sr_data` til `NULL`, og `srdlen` til 0.

For en komplett liste over hvilke *AD Structs* dere kan definere, kan dere sjekke ut [Generic Access Profile \(GAP\) → Defines → GAP Advertising and Scan Response Data format](#).



Om dere ønsker å sette både *complete local name*, og *short local name*, så må *short local name* være et kontinuerlig subsett av *complete local name* og starte fra starten av *complete local name*. Dette er spesifisert i Bluetooth Core Specification versjon 7.

#### 5.4.3 Start kringkastingen

Etter at dere har definert de *AD Structene* dere ønsker, er neste steg å starte sendingen. Sjekk ut funksjonen `sd_ble_gap_adv_start`. Legg merke til at denne funksjonen tar en peker til en `ble_gap_params_t` struct.

For å finne ut hva feltene i `ble_gap_params_t` skal være, følg lenken fra dokumentasjonen til `sd_ble_gap_adv_start`. Kun feltene `type` og `interval` er av interesse for oss. Resten av feltene kan nulles ut, noe vi kan gjøre med funksjonen `memset`:

```
ble_gap_params_t adv_params;
memset(&adv_params, 0, sizeof(adv_params));
// Then set only the interesting fields
```

Funksjonen `memset` vil starte på adressen til `&adv_params` og overskrive `sizeof(adv_params)` antall byte med 0. For komplett dokumentasjon av `memset` kan dere kalle `man 3 memset` fra terminalen. Forøvrig lever `memset` i `<string.h>`.

Dere kan fritt velge måten å kringkaste på, men dere er mest sannsynlig interessert i `BLE_GAP_ADV_TYPE_ADV_IND`. Dere finner en komplette liste over kringkastingsmodiene under `Generic Access Profile (GAP) → Defines → GAP Advertising types`.

#### 5.4.4 Print for å sjekke koden deres

Alle funksjoner i APIet til SoftDevicen vil returnere 0 (`NRF_SUCCESS`) når et funksjonskall er vellykket. For å se hva som blir returnert kan dere for eksempel sende returverdien over UART til datamaskinen. Dere har fått utlevert funksjonen `ubit_uart_print`, som fungerer på samme måte som `printf`, med unntaket av at `ubit_uart_print` kun aksepterer heltall. Et eksempel på hvordan denne funksjonen brukes ser slik ut:

```
ubit_uart_init(); // Call once  
ubit_uart_print("My favorite number is %d\n\n", 2);
```

For å lese UARTEn fra datamaskinen bruker dere som før `picocom`:

```
picocom -b 9600 /dev/ttyACM0
```

#### 5.4.5 Sjekk med telefonen

Hvis programmet kompilerer og kjører med korrekte returverdier, kan dere endelig bruke telefonen for å sjekke om det er noe av interesse på luften. Åpne `nRF Connect` og start skanningen. Dere skal nå se navnet til micro:biten deres, samt alle andre *AD Structs* dere har definert.

Dere har nå gjort dere fortjent til litt ros; å skrive Bluetooth Low Energy firmware er ikke helt trivielt ;)

Om dere nå prøver å koble til micro:biten, blir dere møtt med ganske anstiklimatisk oppførsel. Når GAP for en tilkoblingsforespørsel vil den stoppe å kringkaste, men fordi vi ikke har implementert selve tilkoblingsfunksjonen enda, vil micro:biten simpelthen skru av kringkasting og gjøre ingenting. Til slutt vil nRF Connect time ut tilkoblingsforsøket.

Til slutt kan dere legge merke til at nRF Connect vil rapportere et litt tregere kringkastingsintervall enn det dere spesifiserte i koden. Dette er fordi spesifikasjonen til Bluetooth Low Energy legger til en tilfeldig forsinkelse i hvert intervall, for å redusere sannsynligheten for pakkekollisjoner på båndet.

## 6 Oppgave 6: BLE GATT

### 6.1 Beskrivelse

Selv om GAP er stilig, er det GATT som er arbeidshesten i Bluetooth Low Energy. GATT står for *Generic Attribute Profile*, og tar for seg det meste av dataoverføring i en vanlig BLE-applikasjon. Siden Bluetooth 4.1 ble det riktig nok mulig å bruke et lavere lag, nemlig L2CAP (*Logical Link Controller and Adaption Protocol*), direkte om man ønsker høyere dataoverføringsrater - men det ser fortsatt ut til at Bluetooth Low Energy stort sett er synonymt med GATT på ett eller annet vis i de fleste tilfeller.

Om dere nå er forvirret over rollene til GAP og GATT, så er det helt normalt. Essensen er at GAP tar seg av kringkasting, tilkoblinger, og sikkerhet. GATT er derimot en *profil*, som bruker *protokollen* ATT (*Attribute Protocol*) for å tilby datautveksling over et klient/server-grensesnitt.

### 6.2 Protokoller og Profiler

Det kan være greit å ha et høynivå overblikk over hva Bluetoothspesifikasjonen mener når den nevner protokoller og profiler. Noen dyp forståelse er derimot ikke nødvendig, så fortvil ikke om forskjellen forblir litt frynsete.

#### 6.2.1 Protokoller

I Bluetooth er en *protokoll* et lag som implementerer lavnivå detaljer som hvordan pakker formateres, hvor de sendes, enkoding, multipleksing, og hvilket grensesnitt som brukes for å sende datapakker mellom enheter.

ATT (*Attribute Protocol*) er et eksempel på en slik protokoll. ATT er i utgangspunktet veldig enkelt; den er simpelthen en tilstandsløs maskin som tillater datautveksling via såkalte attributter. ATT kan operere som enten en server, som en klient til en annen server, eller som begge på en gang. Hvilke modi ATT opererer i er også uavhengig av om Bluetooth-enheten er en master eller slave.

Innad i ATT finner man et sett attributter. Hver attributt er arrangert på et eget format bestående av fire komponenter:

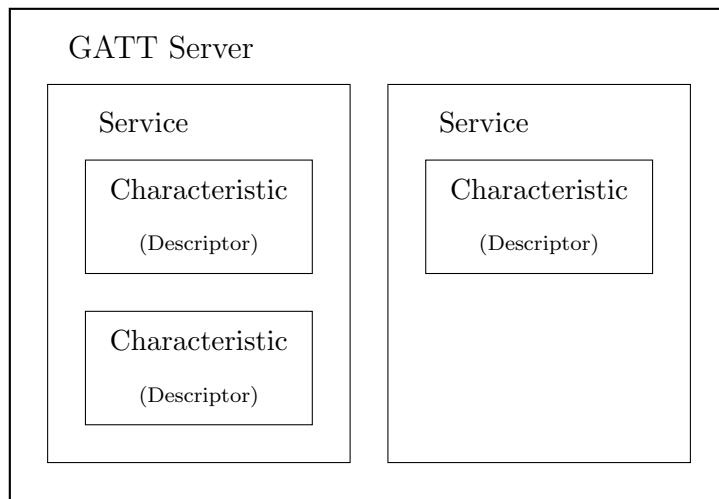
1. Et attributthåndtak, bestående av en 16-bits verdi.
2. Et 128-bits unikt tall, kjent som en UUID (Universally Unique Identifier).
3. Et sett av skrive- og leserettigheter.
4. Attributtens faktiske verdi.

Hver gang en klient ønsker å lese fra- eller skrive en verdi til en attributt i ATT, sendes det en forespørsel til ATT-serveren. Serveren må da behandle hele forespørselen før den er i stand til å motta andre forespørsler fra andre klienter.

For hver ATT-forespørsel bruker klienten et 16-bits attributthåndtak for å identifisere hvilken attributt den prøver å få tilgang til. UUIDen til en attributt beskriver hvilken type data attributten består av. Når en klient leser en attributt, vil ATT-serveren gjøre så lite feilsjekking som mulig; det er klienten som er ansvarlig for å tolke lest data riktig. Dette er i kontrast til en skriveoperasjon, hvor serveren står fritt til å forkaste data som ikke møter forventet format.

### 6.2.2 Profiler

En *profil* er en oppskrift på hvordan man bruker én eller flere protokoller for å oppnå et bestemt mål. Et eksempel på en profil som benytter seg av ATT-protokollen er GATT - eller Generic Attribute Profile.



Figur 9: Oppbygningen av en GATT-server.

GATT bygger sterkt på ATT, men introduserer selvsagt mer terminologi og noen nye konsepter. Den viktigste forskjellen å merke seg er at GATT introduserer mer veldefinert dataorganisering enn ATT. I ATT har man et utall attributter, som rådes over av en server - men stort mer enn det vet man ikke. I GATT er all data organisert i et hierarki av såkalte *services*. En *service* er igjen en samling av *characteristics* (karakteristikker), og et definerert sett med forhold til andre *services*. En karakteristikk er konseptuelt bare en gruppering med relaterte stykker med data. Dette er illustrert i figur 9.

En grei objektorientert analogi er å se på servicer som klasser, og karakteristikker som medlemsvariabler.



Bluetoothspesifikasjonen er veldig generell fordi den skal støtte så mange applikasjoner som mulig. Dette betyr at det blir mange forkortelser og detaljer.  
Pust med magen og slapp av.

### 6.3 Oppgave

## A Bitoperasjoner i C

Når vi arbeider med mikrokontrollere er vi stort sett ikke veldig langt unna “metallet”, i den forstand at vi hele tiden loker rundt med registre og individuelle bit for å oppnå det vi vil. C er et godt egnet språk for mikrokontrollere, nettopp fordi det ikke gjemmer bort tilgang til slike plattformspefikke detaljer.

C har seks forskjellige bitoperatorer:

- & Bitvis og (AND)
- | Bitvis eller (OR)
- ^ Bitvis eksklusiv eller (XOR)
- ~ Ens komplement (Flipp alle bit)
- << Venstreskift
- >> Høyreskift

Den beste måten å lære seg disse på, er nok å tegne opp noen byte og gjøre operasjonene for hånd med penn og papir et par ganger. Her har dere noen eksempler:

```
// The prefix Ob means ``number in binary''
uint8_t a = 0b10101010;
uint8_t b = 0b11110000;
uint8_t c;

c = a | b;           // c is now 1111 1010
c = a & b;           // c is now 1010 0000
c = b >> 2;          // c is now 0011 1100
c = a ^ b;           // c is now 0101 1010
c = ~b;              // c is now 0000 1111
```

Som de andre operatorene, er det mulig å kombinere en bitvis operasjon og et likhetstegn for å modifisere et tall direkte:

```
uint8_t a = 0b10101010;

a <= 4;             // a is now 1010 0000
a >= 4;             // a is now 0000 1010
a |= (a << 4);     // a is now 1010 1010
a |= (a >> 1);     // a is now 1111 1111
a &= ~(a << 4);    // a is now 0000 1111
```



Som dere kanskje ser, er vestreskift det samme som å multiplisere med 2, og høyreskift det samme som å dividere på 2.

Ikke finn på å bytte ut dobling og halvering med bitshifting i “vanlig” kode. Dette gjør ting bare mindre lesbart, og er uansett en optimalisering kompilatoren stort sett gjør for dere.



I koden over brukte vi “0b” for å skrive binære tall. Dette er egentlig *ikke* en del av C-standarden (men C++14). Det er en såkalt “compiler extension” som GCC implementerer.

Vennligst unngå å bruke “0b” siden dette er kompilatorspesifikk oppførsel (og “0x” er mer lesbart uansett). Det er kun brukt her som et pedagogisk hjelpemiddel.

I C bruker vi tall som boolske verdier, der vi tolker 0 som `false` og alt annet som `true`. Det betyr at vi kan isolere et eneste bit, og så teste for sannhet på vanlig vis om vi foreksempel ønsker å vite om en knapp er trykket inne:

```
// GPIO->IN is a register of 32 bits, and button A is held if
// the 17th bit is zero (zero-indexed)

int ubit_button_press_a(){
    return (!(GPIO->IN & (1 << 17)));
}

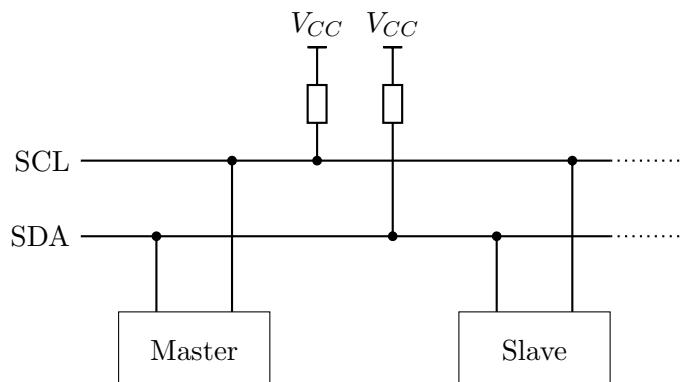
// (1 << 17) gets us bit number 17, counting from 0
// & isolates the 17th bit in GPIO->IN, because we do an AND
// operation with a single bit masking.
// We finally negate the answer, to return true if the bit
// was not set.
```

## B Kort om I<sup>2</sup>C (a.k.a. TWI)

I<sup>2</sup>C (eller bare I2C) er en av de vanligste bussprotokollene for innvevde data-systemer der ute. Det finnes mange andre, slik som SPI, CAN, Ethernet, RS-232/422/485, 1-wire, etc.

Den store fordelen ved I<sup>2</sup>C er derimot at den er ekstremt simpel, billig å implementere, og støttes av nesten alt. Den er en ganske treg protokoll, med maks overføringshastighet på 400 kbps, men dette er mer enn nok for et par sensorer koblet til en mikrokontroller.

Oppkoblingen av en I<sup>2</sup>C-buss ser dere illustrert i figur 10. Dette er en såkalt *open-drain* buss, fordi enheter koblet til bussen bare kan trekke de to signallijnene lave. Linjene trekkes høye når bussen ikke er i bruk av et sett med pullupmotstander.



Figur 10: Oppkobling av en I<sup>2</sup>C-buss.

Protokollen støtter flere enn én master på samme buss, og enheter kan legges til eller fjernes uten å påvirke de andre enhetene som deler bussen. Faktisk kan man i teorien legge til vilkårlig mange enheter - men vanlig I<sup>2</sup>C støtter kun 112 unike adresser<sup>3</sup>. Om man av en eller annen grunn trenger flere enn dette, blir det nødvendig å gjøre litt triksing, eller bruke en annen bussprotokoll.

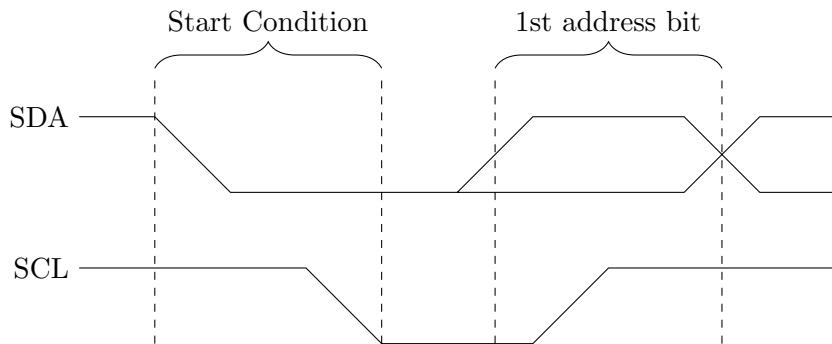
### Start condition

Når bussen ikke er i bruk, er både SCL og SDA høye. En overføring startes ved at en master (av potensielt mange) genererer en *start condition* på busslinjene. En *start condition* betyr simpelthen at masteren trekker SDA lav, etterfulgt av SCL.

---

<sup>3</sup>7-bits adresse gir 128 mulige varianter, men enkelte adresser er reservert. I<sup>2</sup>C har også et utvidet 10-bits adressesystem, som øker adresseområdet til litt over tusen enheter.

Når begge linjene er trukket lave, vil masteren sette første databit på SDA, før SCL går høy for å signalisere til slavene på bussen at SDA kan leses. Etter dette er overføringen i gang. En start condition er illustrert i figur 11.



Figur 11: Start condition, og første adressebit, på I<sup>2</sup>C-buss.

## Overføring

For hvert bit som overføres, må det være riktig på SDA i det SCL går fra lav til høy. Hver slave taster SDA-linjen for hver stigende flanke på SCL-linjen. Begge linjene styres i utgangspunktet av masteren, men hvis en slave ikke er i stand til å motta flere bit, kan den tvinge masteren til å avvente videre sending ved å trekke SCL lav. Dette kalles *clock stretching*.

Hver byte overført over I<sup>2</sup>C må bekreftes av en mottaker. Man sier gjerne at mottakeren sender en “ACK” (for acknowledgement) tilbake. Dette gjøres ved at masteren slipper SDA-linjen etter det åttende bitet den har sendt. Deretter vil masteren generere en ekstra klokkepuls på SCL; og nå er det opp til mottakeren å trekke SDA lav for å signalisere at den har mottatt bytet. Hvis masteren ikke merker at SDA trekkes lav, vil den avbryte sendingen.

## Adressering

Som sagt kan en I<sup>2</sup>C-buss være vert for mange enheter. For å signalisere hvilken enhet masteren ønsker å snakke med, vil hver enhet ha en unik adresse på bussen. For sensorer (slik som akselerometeret og magnetometeret på micro:biten) vil adressen stort sett være forhåndsbestemt fra fabrikanten uten mulighet til å endres.

Etter at en start condition er blitt generert, vil det første bytet masteren sender bestå av 7 adressebit, og ett retningsbit. Retningsbiten forteller om

masteren ønsker å skrive til-, eller lese fra en mottaker. Retningsbitet vil være 1 for en leseoperasjon, og 0 for en skriveoperasjon.

Så fremst 10-bits adressering ikke brukes, vil alle byte som følger etter dette første bytet, være data.

## Arbitrering

Hvis det finnes flere masterer på en buss, kan det hende at to masterer griper etter I<sup>2</sup>C-linjene samtidig. For å bli enige om hvem som får lov til å sende først, brukes mottakeradressen som meglingsmiddel.

I<sup>2</sup>C er en såkalt CSMA/CD-protokoll (**C**arrier **S**ense **M**ultiple **A**ccess with **C**ollision **D**etection), som bare betyr at hver I<sup>2</sup>C-master vil sample busstilstanden og sammenligne med det den selv ønsket å putte på linjene.

Hvis to masterer begynte en overføring samtidig, og en av dem ønsket å sende et høyt bit, mens den andre ønsket å sende et lavt bit - vil masteren som sende det lave bitet trekke SDA lav. Dette vil masteren som ønsket å sende et høyt bit merke, og dermed avslutte sendingen. Fordi adressebytet sendes først, vil den masteren som adresserer den laveste adressen vinne (så lenge de ikke snakker til samme adresse).

## Trivia

Andre protokoller, som for eksempel CAN (Controller Area Network) støtter også forskjellige meldingsprioriteter, noe som er implementert ved at den meldingen som har lavest ID alltid får sende først. Dette er en såkalt CSMA/CD+AMP-protokoll, som står for **C**arrier **S**ense **M**ultiple **A**ccess with **C**ollision **D**etection and **A**rbitration by **M**essage **P**riority.

## C Kort om BLE

Bluetooth Low Energy, eller Bluetooth 4.2 er en versjon av Bluetooth som er skreddersydd for applikasjoner som skal trekke lite strøm, som navnet hinter til. Første versjon av BLE (Bluetooth 4.0) så dagens lys i 2010, og siden da har BLE blitt tatt i bruk overraskende fort av forskjellige industrier - og skapt en grei mengde IoT-relaterte buzzwords. En av hovedpådriverene bak dette er smarttelefoner, og BLE har nytt godt av produsenter som Samsung eller Apple. I tillegg til dette er BLE også attraktivt fordi standarden lar en tilpasse protokollen til sine egne formål, i motsetning til Bluetooth Classic, som kun definerte et snevert sett av use cases.

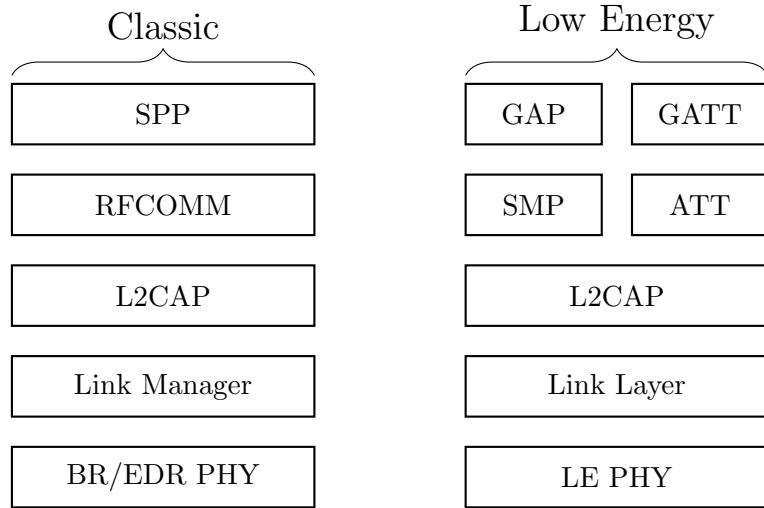
Som et resultat av at BLE trekker langt mindre strøm enn Bluetooth Classic, er også den mulige dataraten redusert. Det er fint mulig å komfortabelt høre på musikk over Bluetooth Classic, men dette er omtrent umulig å få til over en BLE-kobling.

Selv om moduleringsraten til en BLE-radioenhet er 1 Mbps, introduserer protokollen i seg selv en betydelig mengde overhead som begrenser mulig senderate. I vårt tilfelle kan en nRF51822 sende opp til 6 datapakker hvert *connection interval* - der et *connection interval* simpelthen er tiden for én dataoverføring mellom to enheter. Hver datapakke kan inneholde opp til 20 brukerdefinerte byte. Den raskeste man kan sette et *connection interval* til å være er 7.5 ms, så vi har dermed en teoretisk maksgrense på

$$\frac{6 \cdot 20 \cdot 8 \text{ bit}}{7.5 \text{ ms}} = 128 \text{ kbit/s}$$

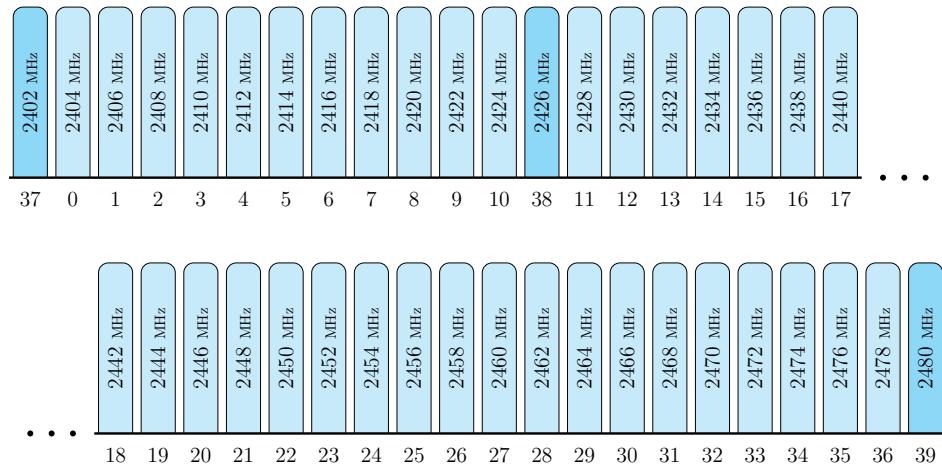
I tillegg til dette vil Bluetooth Low Energy alltid prøve å sende tapte pakker på nytt, kollisjoner kan skje på båndet, radioenheten kan være opptatt, og så videre. Dette betyr at vi i praksis kan forvente omlag 40-80 kbps. Dette er ganske uimponerende med tanke på ren dataoverføringsevne, men så er det heller ikke det BLE er spekket for å gjøre - når det kommer til lavt strømtrekk er dette veien å gå.

Fra et teknisk synspunkt har Bluetooth-stakken endret seg mye fra Bluetooth Classic til Bluetooth Low Energy; så mye at det egentlig er snakk om to forskjellige protokoller som begge har "Bluetooth" i navnet. De har som sagt forskjellige fokusområder, og er heller ikke i stand til å snakke direkte med hverandre. Det er ikke viktig i dette faget, men for de spesielt interesserte, er forskjellene i Bluetooth-stakkene illustrert i figur 12. "BR/EDR" står for "Basic Rate/Enhanced Data Rate", og er betegnelsen vi bruker for Bluetooth Classic.



Figur 12: Stakkforskjeller mellom Bluetooth Classic og BLE.

Mens Bluetooth Classic deler 2.4 GHz ISM-båndet inn i 79 kanaler, nøyser Bluetooth Low energy seg med å dele det samme båndet inn i 40 kanaler, som illustrert i figur 13. Kanalene 37, 38, og 39 er reservert for kringkasting over GAP, mens GATT står fritt til å hoppe mellom de andre kanalene.



Figur 13: Kanalspekteret for Bluetooth Low Energy.

Når det kommer til nettverkstopologier er BLE ganske liberal i hva den tillater. Det er mulig å simpelthen kringkaste til alle som vil høre på, over én eller flere av kringkastingskanalene. Dette tillater en enhet å sende informasjon til flere mottakere samtidig. Dette begrenser derimot en enhet til å

kun sende; for toveis kommunikasjon må man etablere en kobling.

Alle koblinger i Bluetooth Low Energy involverer én *central*, og én *peripheral*. Det er ikke mulig å sende til flere mottakere samtidig når man bruker en tilkobling. For å komme rundt dette, har man siden Bluetooth 4.1 kunnet konfigurere én fysisk enhet til å anta enhver kombinasjon av *central* og *peripheral* man ønsker. Det er dermed mulig å ha en enhet som fungerer både som en *central* og *peripheral* på en gang. Hver av disse *rolleinstansene* står da fritt til å gjøre koblinger med andre enheter rundt seg.

Et sentral konsept i Bluetooth Low Energy er GATT, som står for Generic Attribute Profile. Denne delen av standarden definerer en klient-serverarkitektur, som arrangerer attributter i såkalte *karakteristikker* som til sammen implementerer en eller flere *services*. Neste all informasjonsoverføring av betydning foregår over GATT, og dette er en av de viktigste grunnene til at BLE er en såpass allsidig og anvendelig protokoll som den er.