



Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang

I Introduksjon - Kort om micro:bit

BBC micro:bit er et lite kort som i utgangspunktet ble utviklet for å skape interesse for programmering hos barn (se figur 1). For å gjøre dette mulig, finnes det et veldig vennlig webgrensesnitt der man kan programmere kortet med programmeringsklosser.

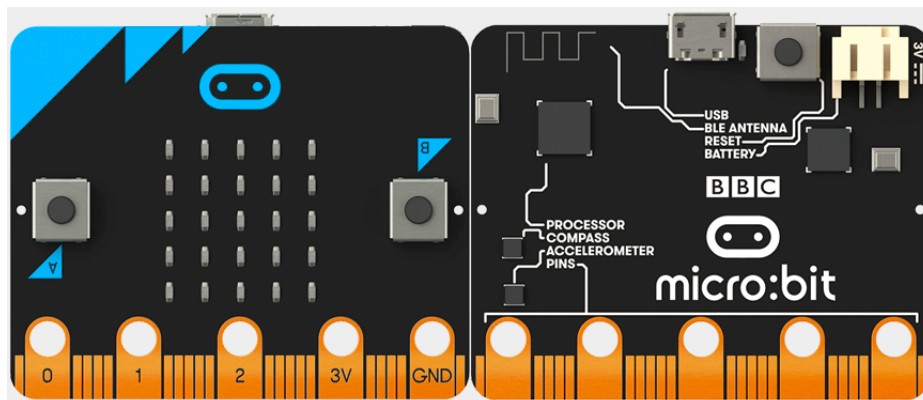


Figure 1: micro:biten brukt i mikrokontroller labben.

Under webgrensesnittet i abstraksjonsnivåstigen ligger micro:bit DAL (**D**evice **A**bstraction **L**ayer). Denne kan programmeres med MicroPython eller JavaScript. Ulempen med micro:bit DAL programmering er at mye av minnet blir brukt opp. 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 med ARM sin mbed-plattform. 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 lagene henger sammen kan man derfor gå forbi DALen og programmere mikrokontrollerens registre direkte. Dette gjøres med C, som er det laveste abstraksjonsnivået man kan få til, uten å gå over til Thumb - som er prosessorens instruksjonssett. Prosessoren på kortet er en ARM Cortex M0, som er integrert inn i en Nordic Semiconductor nRF51822 SoC.

II Introduksjon - Praktisk rundt labben

I denne labben brukes ARM GCC som verktøykjeden for programmering av micro:biten. Denne typen verktøykjede kalles åpen kilde, og er helt uten begrensninger. Dette er i motsetning til andre IDEer (Integrated Development Environments) for utvikling av tilpassede datasystemer som vanligvis koster en del.

For å gjøre denne labben litt lettere blir det lagt til en Makefile for hver oppgave. Denne vil bygge kildekoden, sette opp riktig minnefordeling på prosessoren, og deretter skrive koden til den. I tillegg blir det også utdelt en gjemt undermappe ved navn `.build_system`. Denne inneholder det som skal til for å få koden til å kjøre på micro:biten.

Det er ikke meningen at `.build_system`-mappen skal endres, men om man vil forstå hvordan koden henger sammen med hva som fysisk skjer på micro:biten, er det bare å ta en titt.

II .1 Makefile

I denne labben blir det gitt ut ferdige Makefiler. Slik som i Makefil-øvingen, kaller man `make` fra et terminalvindu i samme mappen som Makefilen for å kompilere C koden. Dette vil genere en `hex`-fil som mikrokontrolleren kan kjøre. For å faktisk overføre `hex`-filen over i programminnet til mikrokontrolleren kaller man `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.

II .2 Programmeringstaktikk

For å sette ønskede registre på nRF51822en, burde man bruke et kjent triks fra C-programmering. Dette innebærer at man lager `structs`, som dekker nøyaktig det minnet man ønsker å tukle med - for dermed å typecaste en peker til starten av minnet inn i `structen`. Dette gjør det mulig å endre `structens` medlemsvariabler, og samtidig skrive til det underliggende minnet.

Dette er definisjonen på "memory mapped IO"; man gjør endringer som i software ser ut som vanlige lese- og skriveoperasjoner i samme minnerom som resten av programmet, men i bakgrunnen peker deler av dette minnet til registre hos perifere enheter. Dette er i kontrast til "port mapped IO", hvor egne instruksjoner brukes for å gjøre operasjoner i et disjunkt minneområde fra programmet (se forøvrig forelesningene for mer om dette tema).

II .3 Datablad

Tilpassede datasystemer er forskjellige fra "vanlige" datasystemer, fordi de er skreddersydde for en spesifikk oppgave. Ofte må de fungere med begrensede ressurser, og gjerne over lang tid kun drevet av et knappecelle-batteri. Derfor må man som oftest glemme en del generelle ting som gjelder uavhengig av plattform, og fokusere på ting som kun gjelder plattformen man arbeider på. Det er her datablad kommer inn.

Datablader er essensielt dersom man vil være god på å programmere tilpassede datasystemer. For micro:biten, gjelder nRF51 Series Reference Manual. Det er viktig å bruke denne flittig, ettersom den gir en nokså kortfattet dokumentasjon som beskriver nøyaktig arkitekturen til datasystemet som blir brukt.

Sjekk ut appendiks B for en kjapp innføring i hvordan man leser og bruker databladet til micro:biten i kontekst av memory mapped IO.

III Introduksjon - Førstegangsoppsett

Før man bruker micro:biten må man laste ned redskapene som trengs for å programmere en micro:bit. subseksjon III .1 går gjennom hvordan man setter opp verktøykjeden på en datamaskin, mens subseksjon III .2 dekker hvordan man fysisk klargjør micro:biten.

III .1 Software

Denne seksjonen gir en generell oppskrift for hvordan man kan sette opp verktøykjeden som trengs for å programmere en micro:bit i Ubuntu. Om du ønsker å jobbe på Windows, anbefales det å bruke en virtuell maskin (sjekk ut appendiks C i øving 1). Spesielt interesserte kan også sjekke ut appendiks G for en annen løsning i Windows (**Ikke anbefalt! Kan fort bli veldig innfløkt.**).

III .1.1 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 kompilatoren installeres ved å kalle:

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

III .1.2 nrfjprog og mergehex

Deretter trengs et verktøy for å få flashet ferdigkompilert kode over på micro:biten. I denne labben brukes Nordic Semiconductor sitt flasheverktøy - **nrfjprog**. Men før dette installeres, er det greit å sjekke om den allerede er på datamaskinen. Dette gjøres med kommandoen:

`nrfjprog --version`, og deretter `mergehex --version` fra kommandolinjen. Om begge disse svarer med en versjon, og `nrfjprog` i tillegg viser en versjon for JLinkARM, så har datamaskinen alt det den trenger for å kunne flashe over ferdigkompilert kode. Dersom dette ikke er tilfellet, så gjør man følgende:

1. Gå til <https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download#infotabs> og velg siste versjon for Linux64.
2. Når man har lastet ned `.tar`-filen, kan man ekstrahere alle filene i den. Naviger deretter til mappen med de nye filene.
3. Kall `sudo dpkg -i --force-overwrite JLink_*.deb` og deretter `sudo dpkg -i --force-overwrite nRF-Command-Line-Tools*.deb`.

III .2 Hardware

For at micro:biten skal kunne kommunisere med `nrfjprog` over JLinkARM, må man oppgradere firmwaren. Dette gjøres ved å laste ned "BBC micro:bit J-Link OB Firmware" fra [www.segger.com/downloads/jlink#BBC microbit](http://www.segger.com/downloads/jlink#BBC_microbit). Dette er en `.hex`-fil, som må lastes inn på micro:biten for å aktivere JLinkARM. For at dette skal funke, må man foreløpig konvertere `hex`-filen til en binær fil før den flashes (se appendiks A). Når dette er gjort, så følger dere denne oppskriften:

1. Start med micro:biten frakoblet datamaskinen.
2. Hold inne "Reset" på micro:biten (knappen rett ved siden av USBen).
3. Mens man holder knappen inne, kobler man inn USBen.
4. Det skal nå komme opp en enhet med navn "MAINTENANCE" på datamaskinen. Man kan nå slippe RESET knappen.
5. Trekk `.bin`-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.

Om alt har blitt gjort riktig skal LED-matrisen på micro:biten slutte å lyse.

1 Oppgave 1 - GPIO

1 .1 Beskrivelse

I denne oppgaven skal vi skru på alle LEDene i matrisen når knappen "B" trykkes, og skru dem av når knappen "A" trykkes. Dette gjøres med GPIO-modulen. Dette er da en modul som har ansvar for generell input og output (GPIO = General Purpose Input Output).

Denne oppgaven er strukturert som en walkthrough, for å introdusere konsepter som skal brukes i senere oppgaver. Tanken er at det blir gradvis mindre håndholding. Før dere starter er det lurt å skimlese appendiks [B](#) og [C](#).

1.2 Oppgave

LED-matrisen på micro:biten består av en 5x5 matrise som har blitt implementert som en 3x9 matrise, hvor to av cellene ikke er koblet til en diode (se figur 2). 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.

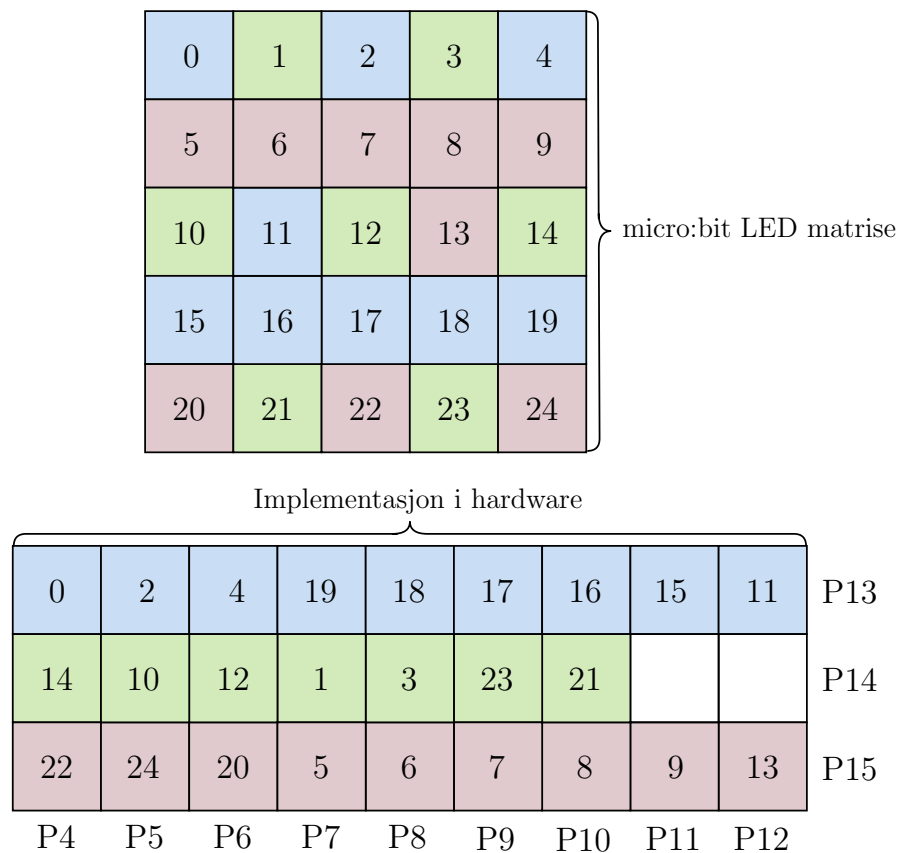


Figure 2: micro:bit LED matrise.

Til å starte oppgaven, ta en titt på den vedlagte filen ”`schematics.pdf`”. Dette er referansedesignet for en micro:bit. Finn ut hvordan de to knappene A og B er koblet.

- Hvilke pinner på nRF51822en brukes? Vil pinnene være høye eller lave dersom knappene trykkes (referer tilbake til videoforelesningene til Martin!)?

Se deretter i databladet til nRF51-serien.

- Hvordan ser minnekartet for micro:biten ut? Hva er baseadressen til GPIO-modulen? Bytt ut `__GPIO_BASE_ADDRESS__` i `main.c` med den faktiske

baseadressen.

I `main.c` vil dere se at det er definert en `struct` som heter `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 (Typisk "memory mapped IO" `struct`). Det er nettopp dette som er formålet med kodelinjen:

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

Når denne er definert, kan vi eksempelvis 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-registret har et offsett 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-registret. Siden vi bruker en ordstørrelse (word) på 32 bit, deler vi dette tallet på fire (32 bit er 4 byte). Altså, $1284/4 = 321$. I hexadesimal, tilsvarer dette `0x141`.

- Dersom man nå følger samme resonnement, hva skal `__RESERVED1_SIZE__` være? Finn ut dette, og endre `main.c` tilsvarende.

Når dere har gjort det, kan dere fylle ut de manglende bitene i `main()`, som består av å legge inn logikk slik at LED-matrisen lyser når vi trykker på knapp B, og skrur seg av når vi trykker på knapp A. Dersom dere nå kaller `make` og `make flash` i terminalen, vil dere kunne se at LED-matrisen lyse av og på, avhengig av hvilken knapp som blir trykket, om alt har blitt gjort riktig.

1.3 Hint

- `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.
- Om dere skriver inn GPIO-modulens baseadresse i base 16 (heksadesimal), må dere huske "0x" foran adressen. Hvis ikke vil kompilatoren tro dere mener base 10.
- Når dere skal finne `__RESERVED1_SIZE__`, så husk at `DIRCLR` starter på `0x51c`, som betyr at den byten slutter på `0x51f`. Altså starter ikke `RESERVED1` på `0x51c`, men på `0x520`.
- `BTN` brukes veldig ofte som en forkortelse for "button".
- Sjekk ut appendiks C for hvordan man kan manipulere bits.
- Sjekk ut appendiks B for hvordan man bruker databladet til å typecaste.

2 Oppgave 2 - UART

2.1 Beskrivelse

I denne oppgaven skal vi sette opp toveis kommunikasjon mellom datamaskinen og micro:Biten. Dette gjøres med **UART** (**U**niversal **A**synchronous **R**eceiver-**T**ransmitter, se gjerne videoforelesninger om dette). Tradisjonelt ble signalene mellom to UART-moduler ofte overført via et RS232-COM grensesnitt. På Sannidssalen finnes det en DSUB9-port som vi kunne brukt til dette, men i denne øvingen trenger vi ikke det.

Om dere ser litt nærmere på micro:biten, vil dere se en liten chip med nummer M2M7V. Dette er en ekstra Freescale MKL26Z128VFM4 mikrokontroller som har blitt inkludert i micro:biten. Denne 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 i gjengjeld se ut som en USB-enhet for datamaskinen.

Les kjapt appendiks [D](#) før dere begynner. Appendikset vil gi dere en kort introduksjon til UART, og litt spesifikk informasjon om begrensningene som kan oppstå ved bruk av UART i micro:biten.

2.2 Oppgave - Innføring i UART

Det første vi må gjøre er å identifisere hvor UART-pinnene faktisk er koblet. For å finne dette ut, tar dere en titt i "schematics.pdf".

- Finn ut hvilken pinne fra nRF51822-brikken som er merket TGT_RXD, og hvilken pinne som er TGT_TXD.

Disse pinnene skal vi senere 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();
```

Disse funksjonene skal brukes for å manipulere UART-modulen i micro:biten. De må derfor inkluderes fra `main.c`.

I implementasjonsfilen (`uart.c`) skal vi igjen bruke memory mapped IO, slik vi gjorde for GPIO med en `struct` til minneoperasjoner:

- Opprett en struct som dere skal typecaste til UART-modulen. Gi denne navnet `NRF_GPIO_REG`.

Som dere kanskje har merket, så har det ikke blitt inkludert en `main.c` i mappen

for denne oppgaven. Det er opp til dere å opprette denne. Om man sitter litt fast på akkurat dette, kan det være hensiktsmessig å ta inspirasjon fra `main.c` fra Oppgave 1.

2.2.1 `void uart_init()`

Målet med denne funksjonen er å initialisere de nødvendige GPIO-pinnene som input/output.

- Første steg er derfor å inkludere `gpio.h` (allerede implementert for dere) i `uart.c`
- Andre steg er å konfigurere pinnene som input eller output i GPIO-modulen.
- Når pinnene er ferdig konfigurert i GPIO-modulen, må de brukes av UART-modulen. Dette gjøres med PSELTXD- og PSELRXD-registrene.

Om dere ser i `schematics.pdf`, vil dere se at vi ikke har noen CTS- eller RTS-koblinger fra nRF51-brikken.

- Dere må derfor velge en baudrate på 9600 for å unngå pakketap på grunn av mangel på flytkontroll i hardware (sjekk ut registeret "BAUDRATE").
- I tillegg er det viktig å faktisk fortelle UART-modulen at vi ikke har CTS- eller RTS-koblinger. Sett opp de riktige registrene for dette (sjekk ut "PSELRTS" og "PSELCTS").
- Til slutt skal vi gjøre to ting. Først må vi skru på UART-modulen, som gjøres med et eget ENABLE-register. Deretter skal vi starte å ta imot meldinger, sjekk derfor ut STARTRX-registeret.

2.2.2 `void uart_send(char letter)`

Denne funksjonen skal ta i mot en enkel bokstav, for å sende den over til datamaskinen.

Sjekk ut figur 68 ("UART Transmission" i side 152) 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.

2.2.3 `char uart_read()`

Denne funksjonen skal lese en 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 ta hensyn til rekkefølge for å kunne garantere at UART-modulen ikke taper informasjon.

- I praksis kan pakketap unngås ved å sette RXDRDY til 0 før RXD blir lest.

- I tillegg er det viktig å sørge for å kun lese RXD en gang. Altså: dere skal ikke skru av mottakerregisteret når dere har lest meldingen.

2.3 Sendefunksjon

Programmer deretter micro:biten til å sende A om knappen A trykkes, og B om B trykkes.

For å motta meldingene på datamaskinen, bruker vi på Sanntidslabben programmet `picocom`. Kall dette fra et terminalvindu:

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

for å fortelle `picocom` at det skal høre etter enheten `/dev/ttyACM0`, med en baudrate på 9600 bit per sekund.

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

2.4 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. Denne logikken implementerer dere i `main.c`

For å sende bokstaver fra datamaskinen bruker vi igjen `picocom`. Standard-oppførselen 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 (gitt at dere ikke manuelt sender bokstaven tilbake fra micro:biten). Sjekk ut [appendiks E](#) dersom dere vil ha mer informasjon om `picocom`, eller om dere får feilmeldinger.

2.5 Oppgave - Mer avansert IO

Nå har dere en funksjon for å sende over nøyaktig en bokstav av gangen; og en funksjon for å motta nøyaktig en bokstav av gangen. Om vi ønsker å sende en C-streng av vilkårlig lengde må vi lage 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++;
    }
}
```

Dette er egentlig en dårlig implementasjon, ettersom den gjør nesten det samme som `printf`, uten noen av formateringsalternativene som gjør `printf` ettertraktet. Det er derfor litt lurere å inkludere `<stdio.h>` og bruke en heltallsvariant av `printf`, kalt `iprintf`. Det er derimot litt problematisk å bruke `iprintf` direkte, ettersom den i utgangspunktet snakker med `stdout`, som tradisjonelt sett peker til en terminal. Derfor bruker vi heller et bibliotek kalt `newlib` som er en variant av `<stdio.h>` for tilpassede datamaskiner.

Når `printf(...)` kalles, vil et annet funksjonskall til `_write_r(...)` utføres 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` (sjekk Makefilen).

Med `newlib` kan vi lage mange varianter av slike skrivefunksjoner om vi har et komplekst system med mange skriveenheter eller om vi har flere tråder. Denne arkitekturen har bare en kjerne og vi vil bare bruke UART, så vi kan fint implementere en global variant av denne skrive funksjonen. For å gjøre det, legger vi til følgende i `main.c`:

```
#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 eksempelvis skrive:

```
iprintf("The average grade in TTK%d was in %d and %d: %c\n\r",4235
,2019,2018,'C')
```

Om `picocom` da forteller dere gjennomsnittskaracteren i tilpassede datasystemer i 2019 og 2018, så har dere fullført oppgaven.

2.6 Oppgave - `_read()` (Frivelig)

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.

2.7 Hint

- På nRFen er det nyttig å tenke på UART-modulen 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 **STOPX** (sjekk figur 68, side 152, i referansemanualen).
- Det skal være totalt 11 reserverte minneområder i UART-structen. De skal ha følgende størrelser: 3, 56, 4, 1, 7, 110, 93, 31, 1, 1, 17.
- Det er ingen forskjell på tasks, events og vanlige registre. Når **LSB** er satt i et event-register, har en hendelse skjedd, mens når **LSB** settes i et task-register, startes en oppgave.
- `int` `fd` i `_read` og `_write` står for "file descriptor". Den er der i tilfelle noen vil bruke `newlib` i forbindelse med et operativsystem. I denne oppgaven lar vi denne være som den er.
- Husk å legge til "uart.c" i Makefilen, bak `SOURCES := main.c`.

3 Oppgave 3: GPIOTE og PPI

3.1 Beskrivelse

Akkurat nå jobber vi på en datamaskin som er basert på en ARM Cortex M0, som bare har en kjerne. Vi har derfor ikke mulighet til å kjøre kode i sann parallellisering. En mulighet er å bytte veldig fort mellom to eller flere fibre, men dette kan være problematisk om man trenger nøyaktige tidsverdier.

For å løse dette problemet, har nRF51822-en noe som kalles **PPI** ("Programmable Peripheral Interconnect"). Dette er en teknologi som lar oss direkte koble en periferienhet til en annen, uten at vi trenger å kommunisere først med CPUen. For å dra nytte av denne teknologien, må vi innføre oppgaver og hendelser (tasks og events). Disse er egentlig bare registre, men brukes litt annerledes enn vanlig registre. Om et hendelsesregister inneholder verdien 1 - så har en hendelse inntruffet. Om den derimot inneholder 0, så har ikke hendelsen inntruffet. Oppgaveregistrene er knyttet til gitte oppgaver, som startes ved å skrive verdien 1 til det. Det som er litt spesielt, er at oppgaven ikke kan stanses ved å skrive verdien 0 til samme register som startet oppgaven.

De fleste periferienhetene som finnes på nRF51822 har noen form for oppgaver og hendelser. For å knytte disse til **GPIO**-pinnene, har vi en egen modul kalt **GPiOTE** (**G**eneral **P**urpose **I**nput **O**utput **T**asks and **E**vents). 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 - Grunnleggende **GPiOTE** og **PPI**

Først må jordingspinnene til LED-matrisen konfigureres som output, og settes til logisk lav. Dere trenger ikke å 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 `gpote.h` og `ppi.h`, men dere må selv lese kapitlene om **GPiOTE** og **PPI** for å se hvordan de skal brukes. Når dette er gjort, skal dere gjøre følgende:

3.2.1 **GPiOTE**

Alle de fire **GPiOTE**-kanalene skal brukes.

- Bruk en kanal til å lytte til **A**-knappen. Denne kanalen skal genere 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 forsyningspinne for LED-matrisen. Forsyningsspenningen skal veksle hver gang oppgaven aktiveres. Hvilken initialverdi disse **GPiOTE**-kanalene har er opp til dere.

3.2.2 **PPI**

For å koble **A**-knapphendelsen til forsyningsoppgavene, trenger vi tre **PPI**-kanaler; en for hver forsyningspinne. Som dere ser 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)&(GPIO->IN[3]);  
PPI->PPI_CH[0].TEP = (uint32_t)&(GPIO->OUT[0]);
```

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

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

- Sett de ulike PPI-registrene til riktige verdier.

3.2.3 Opphold CPU

Når den ene `GPIO`-hendelsen er koblet til de tre `GPIO`-oppgavene gjennom PPI-kanalene, skal LED-matrisen veksle mellom å være av eller på hver gang A-knappen trykkes - uavhengig av hva CPUen gjør. Test dette ut ved å lage en evig løkke hvor CPUen ikke gjør noe nyttig arbeid.

Når dere har kompilert og flashet programmet over til micro:biten, 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 "input bounce".

Idealistisk sett, ville spenningen til A-knappen sett ut som en spenningskurven til en ideell bryter (se figur 3). I virkeligheten vil de mekaniske platene i bryteren gjentatt slå mot-, og sprette fra hverandre. Når dette skjer, får vi spenningskurven for den reelle bryteren i figur 3. I dette tilfellet kan CPUen registrere spenningstransienten som raske knappetrykk.

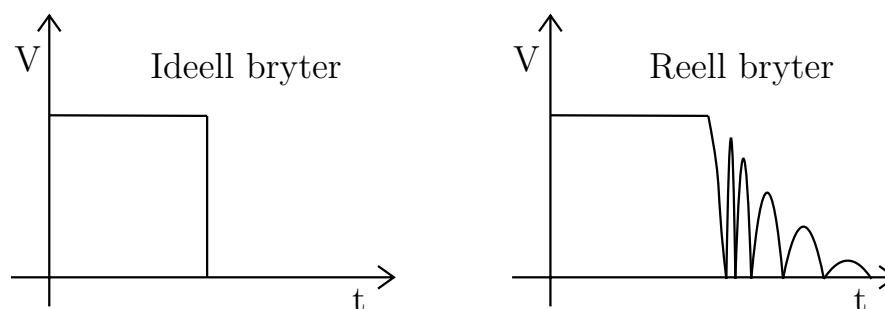


Figure 3: Spenningen over en ideell- og en reell bryter.

Stort sett er det to grunner til at dette ikke er et problem:

1. Vi har tactile pushbuttons på micro:bit-en. Disse er mye bedre på å redusere bounce enn andre typer knapper.
2. I tillegg, dersom man manuelt sjekker knappeverdien i software, vil CPUen som oftest ikke være rask nok til å merke at transienten er der. Dette er

grunnen til at dere sannsynligvis ikke hadde dette problemet da dere brukte GPIO-modulen.

For å komme rundt "input bounce" 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 lavpass-filtrert før micro:biten kan lese den. I software kan man simpelthen vente i en 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 egentlig gjøre for å endre på det. Siden vi i denne oppgaven koblet A-knappen direkte til LED-matrisen med GPIOTE og PPI, har vi heller ikke muligheten til å legge inn software debouncing uten å bryte den evige løkka vi allerede har implementert. Strengt tatt kunne vi koblet knappen fra GPIOTE inn i en TIMER-instans gjennom PPI, og deretter brukt en ny PPI-kanal til å koble en overflythendelse fra TIMER-instansen inn på GPIOTE-oppgavene, men dette er mye mer innsats for en marginal forbedring.

3.3 Hint

- Husk å aktivere hver PPI-kanal. Når de er konfigurert riktig, aktiveres de ved å skrive til CHENSET i PPI-instansen (husk at vi bare bruker 3 PPI-kanaler).
- GPIOTE-kanalene trenger ingen eksplisitt aktivering fordi MODE-feltet i CONFIG-registret 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 I2C (også formatert som I²C), som står for **I**nter-**I**ntegrated **C**ircuit. Denne protokollen ble utviklet av Philips Semiconductor (i dag NXP Semiconductors), som låste ned protokollen bak en del registrerte varemerker.

Av den grunn, begynte andre produsenter å implementere protokollen under andre navn, hvor den mest brukte er TWI (**T**wo **W**ire **I**nterface). Siden 2006 kreves det ikke lenger lisens for å implementere I2C under navnet I2C, men man kommer fortsatt over navn som stort sett er synonyme (se forøvrig forelesningsvideoer for mer informasjon om I2C).

I Nordic sitt tilfelle, implementerer nRF51serien et supersett av I2C; den vanlige protokollen har støtte for vanlig rate på 100 kbps og "Fast-mode" (400 kbps). I tillegg støtter også nRFen også et mellommodus på 250 kbps.

Det dere skal gjøre i denne oppgaven, 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 en LED i matrisen basert på hvordan micro:biten er orientert.

For de mest interesserte, så finner dere et lite appendiks om I2C i [appendiks F](#).

4.2 Oppgave - Grunnleggende TWI-kommunikasjon

Først og fremst må dere finne ut hvordan akselerometeret er koblet til nRFen. Et naturlig sted å starte er "schematic.pdf". Det dere skal se etter er pinnene kalt SDA (Serial Data) og SCL (Serial Clock). Dette er de to linjene som utgjør TWI-bussen.

Etter dette, må man finne ut hva akselerometeret forventer skal skje på TWI-bussen. Dette er beskrevet i databladet "MMA8653FC.pdf", under seksjon 5.8. Det viktige er at dere vet hvilken adresse akselerometeret har, og hvordan man leser ett- og flere byte. Et lite hint er å ikke grave dere ned i detaljer; prøv å få et overblikk først. For eksempel; 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 I2C, så kan dere referere til [appendiks F](#) for å lese mer om protokollen, eventuelt se forelesningene som omhandler temaet.

4.2.1 `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 aktivere 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:

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

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

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

4.2.2 void twi_multi_read()

Som dere kan se i appendiks F og i databladet til akselerometeret, er det mulig å lese- eller skrive flere byte om 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 I2C-enheter gjør dette på, er at de automatisk hopper til et nytt internt register hver gang du har lest fra- eller skrevet til et (sjekk databladet!).

Istedenfor å lage funksjoner som leser- og skriver ett og ett register, 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:

1. Først skal vi kunne adressere en vilkårlig slave, med adresse `slave_address`.
2. Deretter skal vi fortelle slaven hvilket register vi ønsker å begynne å lese fra. I dette tilfellet er det `start_register`.
3. Når dette er gjort, skal vi lese så mange byte vi trenger (`registers_to_read`), og putte dem i arrayet `data_buffer`.
4. 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. 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.
4. 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`.
5. Til slutt kjører dere STOP-oppgaven, før dere leser busser for siste gang. Dette vil gjøre at nRFen generer en NACK istedenfor en ACK, slik at slaven ikke sender ut flere byte på bussen.

Hendelsen 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:

```
TWIO->TXDSENT = 0;
TWIO->TXD = start_register;
while(!TWIO->TXDSENT);
```

```
[...]
```

```
TWIO->RXDREADY = 0;
TWIO->STARTRX = 1;
```

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.

Siden vi er kun ute etter en byte, holder det å ta pekeren til `uint8_t` og bruke det som en buffer for å dytte ID-en inn i. Men for å gjøre det mer generelt til senere, skal vi bruke `malloc()` (fra biblioteket `<stdlib.h>`), som har samme funksjon som `new` i C++. For å allokere minne kan dere gjøre noe slikt:

```
uint8_t * data_buffer;
data_buffer = (uint8_t *)malloc(8 * sizeof(uint8_t));
```

```
[...]
```

```
free(data_buffer);
```

Dette vil allokere minne til et dynamisk array av 8 `uint8_t`, hvor `free` i C brukes for å frigi dynamisk allokert minne tilbake til heapen.

For å teste om TWI-bussen fungerer, kan dere sammenligne det dere får tilbake med akselerometerets fabrikk-ID; `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.

4.2.3 `void twi_multi_write()`

Å skrive `n` byte til en vilkårlig slave er mye lettere enn å lese. Grunnen til dette er at når vi skriver data, så trenger vi ikke å ta hensyn til NACK-grensetilfellet fra leseoperasjonen.

Til og starte med legger dere til denne deklarasjonen i `twi.h`:

```
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. Kort fortalt skal dere:

1. Sette `ADDRESS`-registeret.
2. Starte en skriveoperasjon
3. Skrive `registers_to_write` antall byte til bussen
4. Kalle `STOP`-oppgaven.

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

```
[...]

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

[...]
```

4.3 Oppgave - Lesing av 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 TWI-spesifikke funksjoner 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 deklarer 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 man uforutsigbar oppførsel. Det har derimot ikke noe å si om arrayet er større enn 3. Når funksjonen er kjørt ferdig, vil de tre første elementene i dette arrayet være henholdsvis `x`-, `y`-, og `z`-komponentene til akselerasjonen som akselerometeret måler. Se figur 4 for å se koordinatsystemet dere får komponentene i.

4.3.1 Fyll inn konstanter i `accel.c`

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

- ACCEL_ADDR er adressen til akselerometeret, som dere fant i databladet "MMA8653.pdf" under seksjon 5.8.
- ACCEL_DATA_REG er adressen til det mest signifikante bytet av akselerometerets x-komponent. Dette kan dere finne under seksjon 6 i databladet.
- ACCEL_CTRL_REG_1 er adressen til registeret som kontrollerer dataratene, og om akselerometeret er på eller ei. Dette kan også finnes under seksjon 6 i databladet.

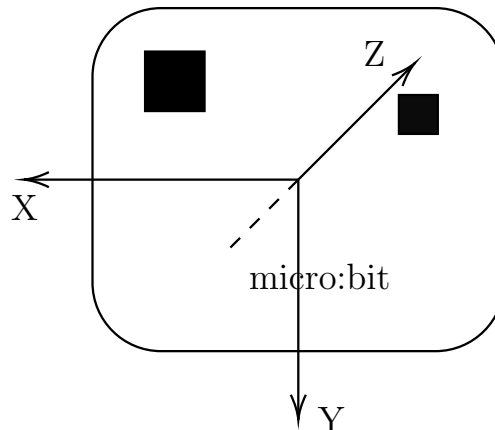


Figure 4: Høyrehåndskoordinatsystemet `accel_read_x_y_z` bruker.

4.4 Oppgave - Skriv verdier til skjerm med UART

Når dere har fylt inn konstantene i `accel.c`, er det fritt frem for å bruke funksjonene filen definerer. For å gjøre det lettere å skrive 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 som `<stdio.h>` bringer med seg.

Det er opp til dere om dere vil bruke `utility_print`, men her er et eksempel på hvordan funksjonen kan brukes (se `utility.h`):

```
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);
```

4.5 Oppgave - Lys opp riktig diode

Når dere greier å lese fra akselerometeret, er det på tide å ha det litt gøy med LED-matrisen til micro:biten. På grunn av måten matrisen er koblet opp på, er det litt komplisert å skru på en og en diode. For å gjøre det litt enkelt, 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 5. Hvordan dere regner ut hvilke koordinater som skal lyse opp gitt akselerasjonen i x- og y-retning, er opp til dere. Et eksempel på hvordan dette kan gjøres kan dere se 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);
```

4.6 Oppgave 4 - Lesing av magnetometeret (Frivilig)

Med `void twi_multi_read(...)` og `void twi_multi_write(...)` er det ganske lett å lese fra magnetometeret også. Dette gjøres på samme måte som for akselerometeret, og er beskrevet i databladet "[MAG3110.pdf](#)". Noe som er kult å implementere er å prøve å bruke magnetometeret som et kompass.

4.7 Hint

- Verdier for de 15 reserverte områdene: 1, 2, 1, 56, 4, 1, 4, 49, 63, 110, 14, 1, 2, 1, 24.
- nRF51822en har to instanser av TWI-modulen, som betyr at dere kan ha to forskjellige TWI-linjer oppe på en gang. Vi trenger bare en av dem i labben.
- Husk å manuelt sette `RXREADY` og `TXDSENT` til 0 etter at dere sjekker disse registrene.
- Det er egentlig unødvendig å bruke dynamisk minne i denne oppgaven. Siden

vi vet ved kompileringstiden hvor stor plass vi trenger, kan vi simpelthen opprette en vanlig liste. Vi bruker dynamisk minne bare for å eksponere dere til C-ekvivalenten av `new`.

- 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 feilaktig TWI-melding, kan dere sette disse i en tilstand hvor 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,

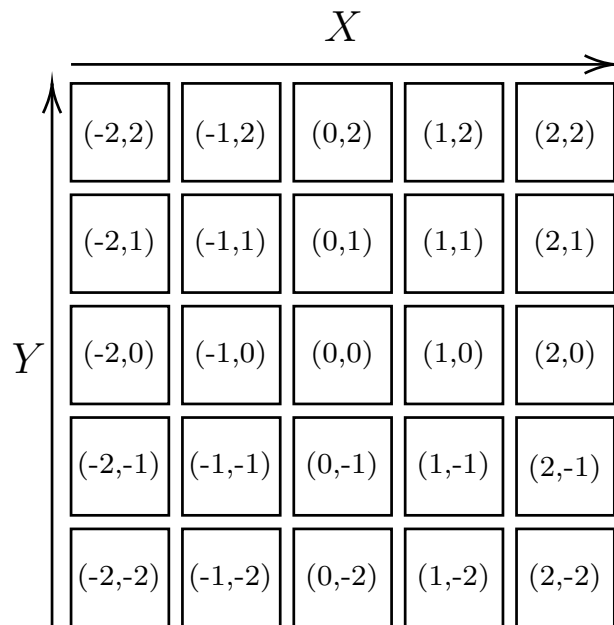


Figure 5: Koordinatsystemet `ubit_led_matrix_light_only_at` bruker.

A Appendiks - Filkonvertering fra .hex til .bin

Foreløpig så er det en bug som oppstår når man prøver å oppdatere firmwaren til micro:biten med BBC micro:bit J-Link OB Firmware. Dette er fordi den nye DAPLink 0234 bootloader som man finner i micro:biten ikke er kompatibel med eldre firmware.

En måte å bypasse dette på, er å konvertere .hex-filen til en .bin-fil. Dette kan man gjøre via <http://www.keil.com/download/docs/7.asp>.

En annen måte å konvertere .hex-filer til .bin på, er ved å bruke hex2bin.py fra Intel Hex Python-biblioteket (sjekk ut <https://python-intelhex.readthedocs.io>). Gitt at man har installert Intel Hex Python-biblioteket, kan man konvertere .hex-filen med følgende kommando:

```
python hex2bin.py /path/to/filename.hex filename.bin
```

hvor /path/to er adressen til .hex-filen man vil konvertere, og filename.hex er filen som skal konverteres.

B Appendiks - Grunnleggende databladkunnskaper

For å bruke micro:bit, eller en hvilken som helst mikrokontroller, er det viktig å kunne mestre bruken av datablader. Mer spesifikt, så er det veldig viktig å forstå hvordan man bruker det som kalles memory mapped IO. I praksis, betyr memory mapped IO at man typecaster adressen til en modul inn i en struct. Grunnen til at man bruker memory mapped IO, er at det gjør det mulig å skrive direkte til registrene i mikrokontrolleren ved å bare endre på structens medlemsvariabler. Se forøvrig pensumlitteratur og forelesninger for mer informasjon om memory mapped IO og hvordan en forholder seg til det i c-programmering.

B.1 Memory Mapped IO informasjon fra datablad

Det første man trenger for å kunne typecaste adressen til en modul inn i en struct, er å finne adressen til modulen. I GPIO-tilfellet, er baseadressen 0x50000000 (se figur 6).

Base address	Peripheral	Instance	Description
0x50000000	GPIO	GPIO	GPIO Port

Figure 6: Startadressen til GPIO-modulen (side 57 fra nRF51_RM_v3.0.1.pdf).

Enkelte eksterne moduler vil ha flere forskjellige "instanser". Et eksempel på dette er Timer-modulen til nRF51en. Der finnes det tre forskjellige kopier av samme enhet (se figur 7). Dette er veldig nyttig dersom man ønsker for eksempel flere uavhengige klokker (ikke relevant for denne labben).

Når man først har baseadressen, oversettes dette ganske direkte inn i C slik:

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

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

Figure 7: Startadressen(e) til Timer-modulen (side 100 fra [nRF51_RM_v3.0.1.pdf](#)).

Denne kodesnutten tilsvarer å definere GPIO som en peker til adresse 0x5000000, hvor pekeren er av typen `NRF_GPIO_REG`.

Neste steg er å definere hvordan `NRF_GPIO_REG` ser ut. Strukturen til `NRF_GPIO_REG` finner man som oftest rett under baseadressen (se figur 8).

Base address	Peripheral	Instance	Description
0x50000000	GPIO	GPIO	GPIO Port

Table 75: Register Overview

Register	Offset	Description
Registers		
<code>OUT</code>	0x504	Write GPIO port
<code>OUTSET</code>	0x508	Set individual bits in GPIO port
<code>OUTCLR</code>	0x50C	Clear individual bits in GPIO port
<code>IN</code>	0x510	Read GPIO port
<code>DIR</code>	0x514	Direction of GPIO pins
<code>DIRSET</code>	0x518	DIR set register
<code>DIRCLR</code>	0x51C	DIR clear register
<code>PIN_CNF[0]</code>	0x700	Configuration of GPIO pins
<code>PIN_CNF[1]</code>	0x704	Configuration of GPIO pins
<code>PIN_CNF[2]</code>	0x708	Configuration of GPIO pins
<code>PIN_CNF[3]</code>	0x70C	Configuration of GPIO pins
<code>PIN_CNF[4]</code>	0x710	Configuration of GPIO pins

Figure 8: Registrerne i GPIO-modulen (side 57 fra [nRF51_RM_v3.0.1.pdf](#)).

Informasjonen som vi trenger for å kunne bruke `NRF_GPIO_REG` sine registre finner man under "Register" og "Offset". "Register" beskriver navnet til registrerne som finnes i modulen, mens "Offset" beskriver offsetet mellom et register, og det registeret som kom før. Eksempelvis vil man for GPIO-modulen kunne se at registeret "OUT" har et offset på 0x504. Dette betyr at registeret ligger $504_{16} = 1284_{10}$ byte unna forrige register. Siden det ikke ligger noe register før OUT i GPIO-modulen, betyr dette at det er 1284_{10} byte mellom baseadressen til modulen og OUT. I C kan man definere `NRF_GPIO_REG`-structen 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 hvert element i et array av typen `uint32_t` er 32 bit stort - altså 4 bytes - noe som tilsvarer registerstørrelsen i prosessoren. Registerstørrelsen i en prosessor er platform-spesifikk, og i dette tilfellet for ARMs (de som har laget prosessorkjernen) Cortex M0-arkitektur. 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 noe tomrom mellom OUT og OUTSET. Dette oversettes direkte til C på denne måten:

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

Slik fortsetter man nedover listen, helt til man kommer til registeret DIRCLR (husk at disse registernavnene er spesifikt til GPIO-modulen! Andre moduler har andre registre.) Dette registeret starter på adresse 0x51C, som betyr at det okkuperer de fire adressene 0x51C, 0x51D, 0x51E og 0x51F. Den neste ledige adressen er 0x520. Registeret PIN_CNF[0] starter derimot ikke på denne adressen. Lik tomrommet på starten, er det standard å legge inn RESERVED for hvert tomrom i modulen. Størrelsen på dette tomrommet finner man ved å ta differansen mellom startadressen til PIN_CNF[0] og neste ledige adresse etter DIRCLR:

$$700_{16} - 520_{16} = 1792_{10} - 1312_{10} = 480 \text{ byte} = 120 \text{ word} \quad (1)$$

I C, bruker man denne informasjonen på denne måten:

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

Merk at i motsetning til tomrommet på starten, så har dette tomrommet fått navnet RESERVED1. Det er standard å inkrementere tallet etter RESERVED for hvert tomrom.

Dersom man nå har definert ferdig NRF_GPIO_REG, så er man egentlig i mål. Da kan man direkte få tilgang til modulens registre ved å derefere pekeren. Eksempelvis, dersom man har lyst til å aksessere GPIO sitt IN-register, kan man simpelthen bare skrive GPIO->IN.

Husk at dette eksempelet baserer seg på databladet for nRF51 Series Reference Manual, som også skal brukes som standard datablad for micro:biten. Ulike datablader kan ha ulik design, men mye av informasjonen er det samme.

B.2 Hint

- Python kan brukes til å regne ut offsetet mellom to registre. Da kan man direkte skrive inn $(0x700 - 0x520) / 4$. Dette vil resultere i 120.0.

C Appendiks - Bitoperasjoner i C

C er et godt egnet språk for mikrokontrollere fordi den ikke gjemmer bort tilgang til plattformspesifikke detaljer. Dette resulterer i at brukeren kan tukle med spesifikke registre og individuelle bits på mikrokontrollerne. I C har man seks forskjellige bitoperasjoner:

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

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

```
// The prefix 0b 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
```

Koden over brukte "0b" for å beskrive binære tall. Dette er egentlig ikke en del av C-standarden (men C++14). Det er en "compiler extension" som er spesifikt til GCC. Derfor:

Vennligst unngå å bruke 0b, siden dette er kompilatorspesifikk oppførsel. Heller bruk "0x"!

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
```

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 for eksempel 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.
```

Et annet eksempel, som kan være litt nyttig for denne labben finner dere i kodesnutten under:

```
/* 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);
```

D Appendiks - Kort om UART

Modulen for UART som finnes på nRF51822-SoCen implementerer noe som kalles full duplex med automatisk flytkontroll. Full duplex betyr at UART-en er i stand til å både sende- og motta meldinger samtidig. Dette blir implementert med en dedikert linje for å motta data, og en dedikert linje for å sende data. Flytkontrollen består av to ekstra linjer, som brukes for å avtale når en enhet kan sende, og når den ikke kan sende.

Kort summert har vi totalt fire linjer: RXD (mottakslinje), TXD (sendelinje), CTS (**C**lear **T**o **S**end) og RTS (**R**equ^est **T**o **S**end). Når alle disse linjene brukes, er det mulig å oppnå en pålitelig overføringshastighet på 1 million bit per sekund. Dette er relativt bra med tanke på at "vanlig" UART-hastighet ligger på 115200 bit per sekund.

Uheldigvis er det litt mer tungvint med micro:biten. Grunnen til dette er at vi blir tvunget til å kommunisere gjennom Freescale-brikken om vi ønsker å kunne tolke signalet som USB. Dette fører til at micro:Biten bare kobler to UART-linjer mellom de to brikkene. Dette resulterer i at vi må holde oss til UART uten flytkontroll. Den høyeste baudraten (bit per sekund) vi pålitelig kan sende med blir derfor redusert til 9600, dersom vi ønsker minimalt med pakketap. Forutsett at vi setter pakkestørrelsen til 8 bit, og bare bruker 2 stoppebit, tilsvarer dette en overføringshastighet på omlag 800 bokstaver per sekund - som burde være mer enn nok i denne oppgaven.

E Appendiks - Kort om picocom

For å debugge eller kommunisere med mikrokontrollere, er det kjekt å bruke `picocom`. Dette er et simpelt program, som åpner, konfigurerer og styrer en seriell port (en `tty`-enhet) og dens innstillinger. Dette gjør `picocom` ved å koble seg til terminalen som man er i. For å starte `picocom`, kaller man:

```
picocom -b baudrate /dev/ttyNAME
```

hvor `baudrate` er overføringsraten til den serielle porten, og `ttyNAME` er navnet på `tty`-enheten.

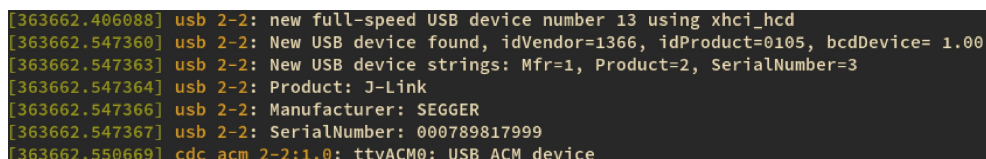
E.1 Vanlige feil ved bruk av picocom

Kanskje den vanligste feilen som kan oppstå ved bruk av `picocom`, er når den klager på manglende rettigheter. Dette kan skje om dere ikke har tillatelse til å lytte til `"/dev/ttyACM0"`. **Dette løses ved å legge til: "sudo" foran picocom.**

En annen vanlig feil som kan oppstå, er når `micro:biten` ikke er koblet til `"/dev/ttyACM0"`. Da vil `picocom` si `"FATAL: [...] No such file or directory"`.

For å løse dette, så må man gjøre følgende:

1. Koble først ut `micro:biten`
2. Åpne en ny terminal, hvor dere kaller `"dmesg --follow"`.
3. Koble i `micro:biten`
4. Det skal nå komme opp en melding om en ny USB-enhet (se figur 9).



```
[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
```

Figure 9: Output fra terminalen.

5. Ta nå navnet som micro:biten ble tildelt av operativsystemet (i dette eksemplet har micro:biten fått navnet "ttyACMO") og prøv det etter "/dev/" i picocom.

F Appendiks - Kort om I²C

I²C (eller bare I2C) er en av de vanligste bussprotokollene for tilpassede datasystemer. Den store fordel ved I2C framfor alternativer som SPI, CAN, Ethernet, RS-232/422/485 og 1-wire er at I2C er ekstremt enkel, billig å implementere, og støttes av nesten alle tilpassede datasystemer. Ulempen er at den er en treg protokoll, med maks overføringshastighet på 400 kbps. Dette er derimot ikke en altfor stor begrensing, ettersom dette er mer enn nok for et par sensorer koblet til en mikrokontroller.

Figur 10 viser oppkoblingen av en I2C-buss. Denne måten å koble enhetene på kalles "open-drain buss" - fordi enheter koblet til bussen må trekke de to signallinjene lave for å aktivere dem. Linjene trekkes høye når bussen ikke er i bruk av et sett med pullupmostander. Protokollen støtter flere enn en master på samme buss, og enheter kan legges til vilkårlige mange enheter - men vanlig I2C støtter kun 112 unike adresser.

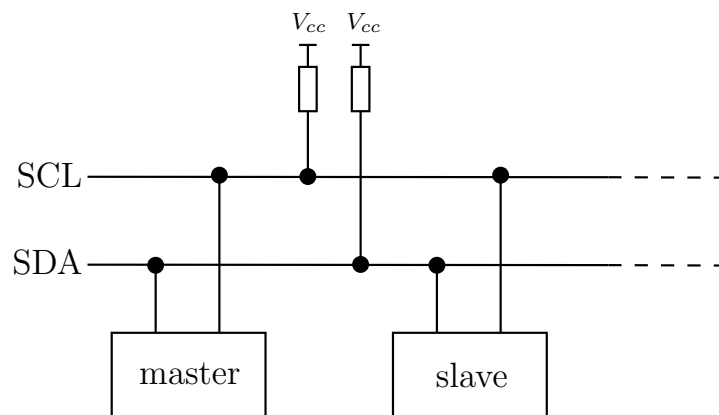


Figure 10: Oppkobling av en I2C-buss

Start condition

SCL og SDA er høye når bussen ikke er i bruk. En overføring startes ved at en master genererer en start condition på busslinjene. Start conditionet består av at masteren trekker SDA lav, etterfulgt av SCL. Når begge disse linjene er 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.

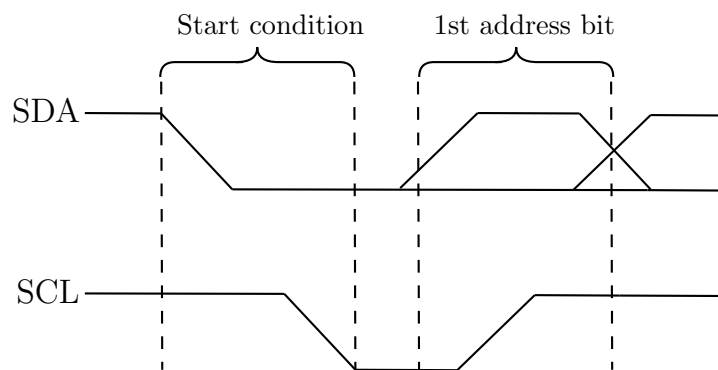


Figure 11: Start condition, og første adressebit, på I2C-buss.

Overføring

For at hvert bit skal overføres korrekt, må det være riktig bit på SDA i det SCL går fra lav til høy. Hver slave taster SDA-linjen for hver stigende anke 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 (denne type oppførsel kalles "clock stretching").

Hver byte overført over I2C må bekreftes av en mottaker. Man sier gjerne at mottakeren sender en "ACK" (Acknowledgement) tilbake. Dette gjøres ved at masteren slipper SDA-linjen etter det åttende bit-et 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.

Addressering

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 har blitt generert, vil det første bytet masteren sender bestå av 2 adressebit, og ett retningsbit. Retningsbitet 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 fremt 10-bits adressering ikke brukes, vil alle byte som følger etter dette første bytet, være data.

Arbitrering

Dersom det finnes flere mastere på en buss, kan det hende at to mastere griper etter I2C-linjene samtidig. For å bli enige om hvem som får lov til å sende først, brukes mottakeradressen som meglingsmiddel.

I2C er en CSMA/CD+AMP protokoll (**C**arrier **S**ense **M**ultiple **A**ccess with **C**ollision

Detection and Arbitration by Message Priority). Dette betyr at hver I2C-master vil sample busstiltanden og sammenligne med det den selv ønsket å putte på linjene. Hvis to mastere 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 sender det lave bitet trekke SDA lav. Dette vil masteren som ønsket å sende et høyt bit merke, og dermed avslutte sendingen. Fordi adressebyttet sendes først, vil den masteren som adresserer den laveste adressen vinne (så lenge de ikke referer 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. Denne type protokoll har navnet CSMA/CD+AMP (Carrier Sense Multiple Access with Collision Detection and Arbitration by Message Priority)

G Appendiks - micro:bit i Windows

Det er fullt mulig å utvikle kode for micro:bit på Windows¹. Dersom man velger å programmere micro:bit i Windows så må man erstatte subseksjon III .1 i oppgaveteksten med følgende steg:

1. Gå til <https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download#infotabs> og installer "nRF command line tools"
 - Mulig dere også må gå inn her <https://www.segger.com/downloads/jlink/>, trykke på "click for downloads" under "J-Link Software and Documentation pack" og velge riktig versjon her for installasjon.
2. Gå til <https://www.segger.com/downloads/jlink/>, finn fram til "BBC micro:bit"-seksjonen og "click for downloads". Last ned "BBC micro:bit J-link OB Firmware". Refer til appendiks A om hvordan man konverterer .hex til .bin. Dette er nødvendig, fordi de nyere micro:bitene ikke støtter eldre firmware.
3. Følg oppskriften her for å installere GCC arm toolchain på Windows: https://www.jann.cc/2013/10/10/embedded_development_with_open_source_tools_on_windows.html#install-the-gcc-arm-embedded-toolchain:
 - Installere Minimalist GNU for Windows herfra: <https://sourceforge.net/projects/mingw/>
 - Installere GNU Arm Embedded Toolchain herfra: <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads>

¹Merk at denne løsningen er noe mer innfløkt enn å bruke Ubuntu, og er dermed ikke anbefalt.

Det skal nå være mulig å programmere micro:biten ved å kjøre samme kommandoer som er oppgitt i labøvingen gjennom mintty.exe-terminalen. Det vil si: en laster opp kode ved å kjøre **make flash** i katalogen som inneholder kildekoden for prosjektet du jobber med.