



Revisjonshistorie

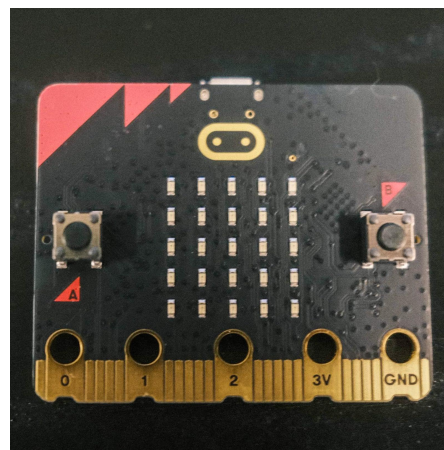
År	Forfatter
2020	Kolbjørn Austeng
2021	Kiet Tuan Hoang
2022	Kiet Tuan Hoang
2023	Kiet Tuan Hoang
2023	Tord Natlandsmyr
2024	Terje Haugland Jacobsson Tord Natlandsmyr

I Introduksjon - Kort om micro:bit

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



(a) Baksiden med sensorer



(b) Forsiden med led-lys og knapper

Figure 1: micro:bit-en brukt i mikrokontroller labben.

Under webgrensesnittet i abstraksjonsnivåstigen ligger micro:bit DAL (Device

Abstraction Layer). 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, som er ganske mye ettersom micro:bit-en bare har totalt 128 KB.

For å få en mer effektiv bruk av ressursene er det også mulig å benytte seg av C++, eksempelvis med 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 lagene henger sammen kan man derfor gå forbi DAL-en 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 M4, som er integrert inn i en Nordic Semiconductor nRF52833 SoC.

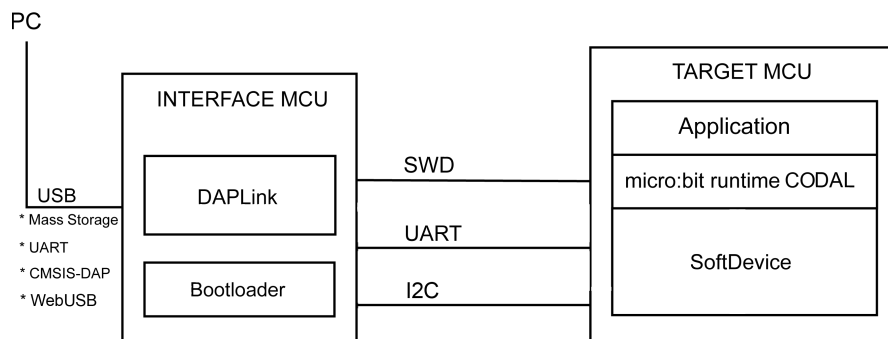


Figure 2: micro:bit-en har to mikrokontrollere.

Hvis du ser nøye etter på baksiden av micro:bit-en, vil du legge merke til at den har to chipper: en større merket med N52833, og en mindre merket med N52820. Micro:bit-en har nemlig to mikrokontrollere, også kalt MCUer (**M**icro**c**ontroller **U**nit): En som kjører programvare utelukkende for å programmere og feilsøke hovedchippet, og en som faktisk kjører koden vår. Vi skiller mellom dem med ved navnene "Target" og "Interface" (se figur 2). Når du kobler til micro:bit-en til en datamaskin, vil datamaskinen kommunisere med interface MCUen og oppdage den som USB-lagringseenhet. Programvaren [DAPLink](#) gjør det da mulig å laste over ferdigkompileerte programmer over til USB-enheten, og vil programmere target MCUen for oss. Det er denne metoden vi skal bruke til å programmere micro:bit-en i denne laben. De som er ekstra interesserte kan se [appendiks F](#) for alternative metoder for både debugging og programmering.

II Praktisk rundt filene

I denne laben får dere utlevert noen .c og .h-filer. Egne tabeller under hver oppgave lister opp alle filene som kommer med, samt litt informasjon om dere skal endre på filene eller om dere skal la dem forbli i løpet av oppgaven.

III Introduksjon - Praktisk rundt laben

I denne laben brukes ARM GCC som verktøykjeden for programmering av micro:bit-en. Denne typen verktøykjede kalles åpen kilde, og er helt uten begrensninger. Dette er i motsetning til andre IDE-er (**I**ntegrated **D**evelopment **E**nvironments) for utvikling av tilpassede datasystemer som vanligvis koster en del.

For å gjøre denne laben 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 undermappe ved navn **.build_system**. Denne inneholder det som skal til for å få koden til å kjøre på micro:bit-en.

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:bit-en, er det bare å ta en titt.

Målet med denne laben er at dere først og fremst skal ha det gøy. Siden dere får ta med micro:bit-en hjem etter faget, så er tanken at dere skal lære nok i denne laben slik at eventuelle hobbyprosjekter kan implementeres direkte på micro:bit-en. I tillegg skal dere lære dere hvordan man leser datablad. Det å kunne lese datablad er veldig viktig dersom man har lyst til å jobbe med mikrokontrollere senere, men også til eksamen.

III .1 Makefile

I denne laben blir det gitt ut ferdige **Makefiler**. Slik som i **Makefile**-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 i **.build_system**-mappen. I tillegg til **make**, har denne **Makefilen** også tre andre mål; **make debug** vil kompilere C-koden med debug-flagg for feilsøking av programmet, **make erase** vil slette minnet til mikrokontrolleren, mens **make clean** vil slette ferdigkompilert kode og **hex**-filen fra datamaskinen. For å faktisk overføre **hex**-filen over i programminnet til mikrokontrolleren går man først inn i **.build_system**-mappen, også dragger man **hex**-filen over til **MICROBIT**-mappen som representerer micro:bit-en. Denne mappen burde komme opp som en enhet på datamaskinen når man kobler micro:bit-en med USB på datamaskinene på sanntidssalen. Alternativt kan en kjøre målet **make load** for å laste over **hex**-filen til micro:bit-en. Denne metoden krever at man har laster over en **.hex**-fil via drag-and-drop-metoden først for å skrive over programmet micro:bit-en kommer med.

III .2 Programmeringstaktikk

For å sette ønskede registre på nRF52833-en, 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 `struct`-en. Dette gjør det mulig å endre `struct`-ens 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).

III .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:bit`-en, gjelder `nRF52833 Product Specification` (denne finner dere i samme mappe som denne `.pdf`-en). Det er viktig å bruke denne flittig, ettersom den gir en nokså kortfattet dokumentasjon som beskriver nøyaktig arkitekturen til datasystemet som blir brukt.

Det er lurt å sjekke ut [appendiks A](#) for en kjapp innføring i hvordan man leser og bruker databladet til `micro:bit`-en i kontekst av *memory mapped IO* før dere slenger dere løs på oppgavene.

III .4 Førstegangsoppsett

Før man bruker `micro:bit`-en må man laste ned redskapene som trengs for å programmere en `micro:bit` i Ubuntu. Det eneste som trengs før dere kan begynne egentlig er kompilatoren. Denne burde allerede være installert, men dersom den ikke er det så kan den installeres med følgende kommando:

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

Kompilatoren vi skal bruke er GCC for ARM. På Linuxmaskiner finnes denne utvidelsen for GCC gjerne i systempakkelageret allerede.

III .5 Strategier for feilsøking

Å feilsøke, også kalt å **debugge**, mikrokontrollere kan være utfordrende på grunn av mangelen på interaktive verktøy og begrensede ressurser tilgjengelig på mikrokontrollere. Selv om kompilatoren luker ut de fleste feil, er man fremdeles utsatt for logiske feil. Det finnes imidlertid to vanlige metoder som kan brukes til å feilsøke program som kjører på tilpassede datasystemer som du kan lære om i denne laben.

En av disse metodene er **Seriell debugging** som innebærer å koble en seriell kabel (se oppgave 2) mellom micro:bit-en og datamaskinen for å bruke et seriell terminalprogram for å skrive ut feilsøkingssdata fra micro:bit-en.

Den andre metoden er å bruke et debuggingsprogram som kommuniserer med Interface MCUen (se figur 2). Denne metoden blir beskrevet i appendiks F. Selv om det ikke er noe krav om å bruke noen av disse metodene for å feilsøke oppgavene, anbefaler vi å undersøke dem av egen interesse.

1 Oppgave 1 - GPIO

1.1 Beskrivelse

I denne oppgaven skal vi skru på alle LED-ene i matrisen når knappen B trykkes, og skru dem av når knappen A trykkes. Dette gjøres med GPIO-modulene. Dette er moduler som har ansvar for generell input og output (GPIO = **G**eneral **P**urpose **I**nput **O**utput).

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 A og B.

I denne oppgaven, så trenger dere bare å endre på `main.c`. De spesielt interesserte kan se på mappen `.build_system` og `Makefile`. Sistnevnte kan endres på om dere velger å lage flere `.h` eller `.c`-filer for at det skal bli ryddigere.

Filer	Skal denne filen endres?
<code>1_gpio/main.c</code>	ja
<code>1_gpio/.build_system</code>	helst ikke
<code>1_gpio/Makefile</code>	helst ikke

1.2 Oppgave

LED-matrisen på micro:bit-en består av en 5x5 matrise som har blitt implementert som en 5x5 matrise (se figur 3). I denne matrisen er det egentlig ikke et system (i tidligere micro:bit så har rad 1-5 alltid vært etter hverandre med hensyn på pins. Det same gjaldt også kolonnene), hvor pinne 28, 11, 31, 5, og 30 tilsvarer jord, mens pinne 21, 22, 15, 24, og 19 tilsvarer strømforsyning. Dermed, for å få diode nummer 12 til å lyse, må P31 være trukket lav, mens P15 være høy.

Til å starte oppgaven, ta en titt på den vedlagte filen `schematics.pdf` (i samme mappe som denne `.pdf`-en). Dette er referansedesignet for en micro:bit V2. Finn først ut hvordan de to knappene A og B er koblet.

- Hvilke pinner på nRF52833-en brukes? Vil pinnene være høye eller lave dersom knappene trykkes (referer tilbake til videoforelesningene til Martin!)?

Se deretter i databladet til nRF52833-serien (`nrf52833 Product Specification.pdf`).

Implementasjon i hardware (LED matrise)

0	1	2	3	4	P21 (ROW 1)
5	6	7	8	9	P22 (ROW 2)
10	11	12	13	14	P15 (ROW 3)
15	16	17	18	19	P24 (ROW 4)
20	21	22	23	24	P19 (ROW 5)
P28	P11	P31	P5	P30	

Figure 3: micro:bit LED matrise. Alt som er i grønt kan styres med GPIO-instans nummer 2, mens alt som er i blått kan styres med GPIO-instans nummer 1.

- Hvordan ser minnekartet for micro:bit-en ut? Hva er baseadressen til GPIO-modulene? Bytt ut `__GPIO_BASE_ADDRESS0__` og `__GPIO_BASE_ADDRESS1__` i `main.c` med de faktiske baseadressene.

I `main.c` vil dere se at det er definert to `structs` som heter `NRF_GPIO_REGS0` og `NRF_GPIO_REGS1`. Disse `struct`-ene representerer alle registrene til GPIO-modulene. Ved å typecaste adressene til GPIO-modulene inn i `struct`-ene, kan vi så endre på `struct`-ens medlemsvariabler for så å skrive direkte til registrene (Typisk *memory mapped IO struct*). Det er nettopp dette som er formålet med kodelinjen:

```
#define GPIO0 ((NRF_GPIO_REGS0*)__GPIO_BASE_ADDRESS0__)
#define GPIO1 ((NRF_GPIO_REGS1*)__GPIO_BASE_ADDRESS1__)
```

Når disse er definert, kan vi eksempelvis endre OUT-registret ved å kalle:

```
GPIO0->OUT = desired_value;
GPIO1->OUT = desired_value;
```

Dere vil også se at medlemsvariabelen `RESERVD0` i `GPIO0` er en array av type `volatile uint32_t` med 321 elementer. Dette er fordi databladet forteller oss at OUT-registret i modulen `GPIO0` har et offset på `0x504` (504_{16}) fra 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

- `nRF52833` har to `GPIO`-moduler. Det er viktig at dere setter riktig port avhengig av hvilken pin. Om det står `P1.05` så betyr dette at den tilhører `GPIO1`, mens pin `P0.05` tilhører `GPIO0`.
- Ikke forveksl `GPIO` med `GPIOE`-modulen (`GPIO Tasks and Events`). Sistnevnte brukes for å lage et hendelsesbasert system. Vi skal i første omgang kun bruke `GPIO`-modulene.
- Om dere skriver inn `GPIO`-modulenes 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 `DETECTMODE` starter på `0x524`, som betyr at den byten slutter på `0x527`. Altså starter ikke `RESERVED1` på `0x524`, men på `0x528`.
- `BTN` brukes veldig ofte som en forkortelse for *button*.
- Sjekk ut appendiks B for hvordan man kan manipulere bits.
- Sjekk ut appendiks A for hvordan man bruker databladet til å typecaste.
- Om dere sliter veldig med å finne hvilke pins som tilsvarer hva, så kan dere finne en forenklet pinmap for `micro:bit V2` her: <https://tech.microbit.org/hardware/schematic/#v2-pinmap>. For vårt bruk så burde `V2` pinmappen tilsvare `V2.2`.

2 Oppgave 2 - UART

2.1 Beskrivelse

I denne oppgaven skal vi sette opp toveis kommunikasjon mellom datamaskinen og `micro:bit`-en. Dette gjøres med `UART` (`Universal Asynchronous Receiver-Transmitter`, se gjerne videoforelesninger om dette). Tradisjonelt ble signalene mellom to `UART`-moduler ofte overført via et `RS232-COM` grensesnitt. På Santidssalen 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:bit`-en, vil dere se en liten chip med nummer `N52820QDAACO2130C0`. Dette er en `nRF52820` mikrokontroller som har blitt inkludert i `micro:bit`-en. Denne lar oss programmere `nRF52833-SoC` over `USB`. I tillegg til dette implementerer den en `USB CDC` (`Communications Device Class`), 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 C 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:bit-en.

I denne oppgaven, så trenger dere ikke å endre noe som helst annet enn `Makefile`. Dette er fordi dere skal implementere en `main.c` selv som bruker logikk fra GPIO-modulene til å kommunisere med en datamaskin via UART-modulen som dere kommer til å lage. Igjen, de spesielt interesserte kan se på mappen `.build_system`.

Filer	Skal denne filen endres?
<code>2_uart/gpio.c</code>	nei
<code>2_uart/gpio.h</code>	nei
<code>2_uart/.build_system</code>	helst ikke
<code>2_uart/Makefile</code>	ja

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 nRF52833-brikken som er merket `UART_INT_RX`, og hvilken pinne som er `UART_INT_TX`.

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:bit-en. De må derfor inkluderes fra `main.c`.

I implementasjonsfilen (`uart.c`) skal vi igjen bruke memory mapped IO, slik vi gjorde for GPIO0 og GPIO1 med `struct`-er til minneoperasjoner:

- Opprett en `struct` som dere skal typecaste til UART-modulen. Gi denne navnet `NRF_UART_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-modulene, 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 nRF52-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 TASKS_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 176 (*UART Transmission* i side 487) i databladet til nRF52-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 `EVENTS_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:bit-en til å sende A om knappen A trykkes, og B om B trykkes i `main.c`.

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

Deretter, lytt etter sendte pakker på micro:bit-en. Om datamaskinen har sendt en bokstav, skal micro:bit-en 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:bit-en). Sjekk ut [appendiks D](#) 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->TASKS_STARTTX = 1;
    char * letter_ptr = *str;
    while(*letter_ptr != \0){
        UART->TXD = *letter_ptr;
        while(!UART->EVENTS_TXDRDY);
        UART->EVENTS_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 følgene 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 gjennomsnittskarakteren 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:bit`-en, 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 `TASKS_STARTTX`. Den vil bare stoppe å sende når den forlater denne tilstanden, altså når den går over i `TASKSSTOPX` (sjekk figur 176, side 487, i referansemanualen).
- Det skal være totalt 12 reserverte minneområder i `UART_struct`-en. De skal ha følgende størrelser: 3, 56, 4, 1, 7, 46, 64, 93, 31, 1, 1, 17.
- Det er ingen fysisk forskjell på tasks, events og vanlige registre annet enn hva de brukes til. 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 M4, 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 nRF52833-en noe som kalles **PPI** (**P**rogrammable **P**eripheral **I**nterconnect). Dette er en teknologi som lar oss direkte koble en periferienhet til en annen, uten at vi trenger å kommunisere først med CPU-en. 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 nRF52833 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**nterface **O**utput **T**asks and **E**vents). I denne oppgaven skal vi bruke GPIOTE-modulen til å definere en hendelse (A-knapp trykket), og fem oppgaver (skru på eller av spenning til de fem LED-matriseforsyningene).

I denne oppgaven, så får dere igjen utlevert ferdig `gpio.h`. I tillegg, så får dere halvferdige `.h`-filer for PPI og GPIOTE-modulene som dere selv skal implementere. Som i forrige oppgave, skal dere selv implementere de tilhørende `.c`-filene og en `main.c`-fil. Dere må derfor også endre `Makefile`.

Filer	Skal denne filen endres?
3_gpiote/gpio.h	nei
3_gpiote/ppi.h	ja
3_gpiote/gpiote.h	ja
3_gpiote/.build_system	helst ikke
3_gpiote/Makefile	ja

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 `gpiote.h` og `ppi.h` uten riktig informasjon. Dere må selv lese kapitlene om GPIOTE og PPI for å se hvordan de skal brukes og hva som skal fylles inn før dere kan bruke dem. Når dette er gjort, skal dere gjøre følgende:

3.2.1 GPIOTE

Seks 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 fem 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 fem 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)&(GPIOTE->EVENTS_IN[5]);  
PPI->PPI_CH[0].TEP = (uint32_t)&(GPIOTE->TASKS_OUT[0]);
```

Denne kodesnutten setter registeret `EventEndPoint` for PPI-kanal 0 til adressen av `GPIOTE-EVENTS_IN[5]` - typecastet til en `uint32_t`. Tilsvarende vil den sette registeret `TaskEndPoint` for PPI-kanal 0 til adressen av `GPIOTE->TASKS_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 GPIOTE-hendelsen er koblet til de fem GPIOTE-oppgavene gjennom PPI-kanalene, skal LED-matrisen veksle mellom å være av eller på hver gang A-knappen trykkes - uavhengig av hva CPU-en gjør. Test dette ut ved å lage en evig løkke hvor CPU-en ikke gjør noe nyttig arbeid.

Når dere har kompilert og flashet programmet over til micro:bit-en, 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 4). 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 4 I dette tilfellet kan CPU-en registrere spenningstransienten som raske knappetrykk.

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 CPU-en som oftest ikke være rask nok til å merke at transienten er der. Dette er

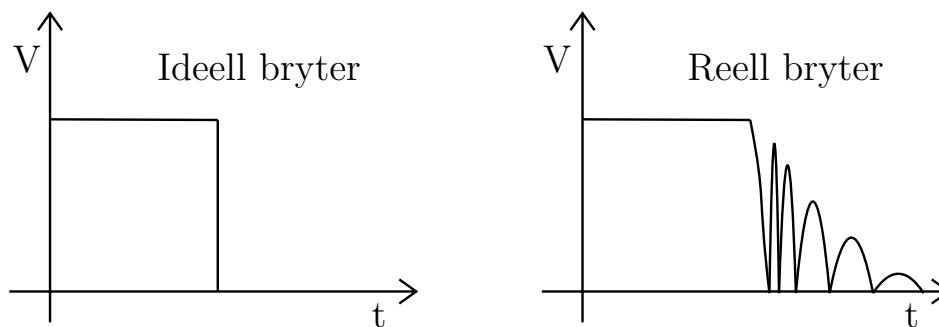


Figure 4: Spenningen over en ideell- og en reell bryter.

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:bit-en 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 seks PPI-kanaler totalt).
- GPIOTE-kanalene trenger ingen eksplisitt aktivering fordi MODE-feltet i CONFIG-registeret automatisk tar hånd om pinnen for dere.

ls

4 Oppgave 4: Two Wire Interface (frivillig)

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 nRF52-serien 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 nRF52833-SoCen til å kommunisere med akselerometeret som finnes på micro:bit-en. Dere skal deretter bruke denne sensorinformasjonen til å lyse opp en LED i matrisen basert på hvordan micro:bit-en er orientert.

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

Uheldigvis så er jeg ennå ikke ferdig med denne oppgaven, men den vil bli gitt ut på BlackBoard snarest mulig.

A Appendiks - Grunnleggende databladkunnskaper

For å bruke micro:bit, eller en hvilket 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 **struct**-ens medlemsvariabler. Se forøvrig pensumlitteratur og forelesninger for mer informasjon om memory mapped IO og hvordan en forholder seg til det i C-programmering.

A.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 enten `0x50000000` eller `0x50000300` (se figur 5).

Som dere kan se, så har noen moduler flere *instanser* av en modul. Et annet eksempel på dette er **Timer**-modulen til nRF52-en. Der finnes det fem forskjellige kopier av samme enhet (se figur 6). Dette er veldig nyttig dersom man ønsker for eksempel flere uavhengige klokker (ikke relevant for denne laben).

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

```
#define GPIO0 ((NRF_GPIO_REG0*)0x50000000)
```

6.8.2 Registers

Base address	Peripheral	Instance	Description	Configuration
0x50000000	GPIO	GPIO	General purpose input and output	Deprecated
0x50000000	GPIO	P0	General purpose input and output, port 0	P0.00 to P0.31 implemented
0x50000300	GPIO	P1	General purpose input and output, port 1	P1.00 to P1.09 implemented

Table 41: Instances

Figure 5: Startadressene til GPIO-modulene (side 144 fra `nrf52833 Product Specification.pdf`).

6.28.5 Registers

Base address	Peripheral	Instance	Description	Configuration
0x40008000	TIMER	TIMER0	Timer 0	This timer instance has 4 CC registers (CC[0..3])
0x40009000	TIMER	TIMER1	Timer 1	This timer instance has 4 CC registers (CC[0..3])
0x4000A000	TIMER	TIMER2	Timer 2	This timer instance has 4 CC registers (CC[0..3])
0x4001A000	TIMER	TIMER3	Timer 3	This timer instance has 6 CC registers (CC[0..5])
0x4001B000	TIMER	TIMER4	Timer 4	This timer instance has 6 CC registers (CC[0..5])

Table 117: Instances

Figure 6: Startadressene til Timer-modulen (side 446 fra `nrf52833 Product Specification.pdf`).

Denne kodesnutten tilsvarer å definere den første instansen av `GPIO` som en peker til adresse `0x50000000`, hvor pekeren er av typen `NRF_GPIO_REG0`.

Neste steg er å definere hvordan `NRF_GPIO_REG0` ser ut. Strukturen til `NRF_GPIO_REG0` finner man som oftest rett under baseadressen (se figur 7).

Informasjonen som vi trenger for å kunne bruke `NRF_GPIO_REG0` sine registre finner man under `Register` og `Offset`. `Register` beskriver navnet til registrene 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_REG0-struct`-en slik (om dere leser nøye i figur 7 så er den eneste forskjellen mellom `GPIO0` og `GPIO1` antall `PIN_CNF` de har. Resten er det samme):

6.8.2 Registers

Base address	Peripheral	Instance	Description	Configuration
0x50000000	GPIO	GPIO	General purpose input and output	Deprecated
0x50000000	GPIO	P0	General purpose input and output, port 0	P0.00 to P0.31 implemented
0x50000300	GPIO	P1	General purpose input and output, port 1	P1.00 to P1.09 implemented

Table 41: Instances

Register	Offset	Description
OUT	0x504	Write GPIO port
OUTSET	0x508	Set individual bits in GPIO port
OUTCLR	0x50C	Clear individual bits in GPIO port
IN	0x510	Read GPIO port
DIR	0x514	Direction of GPIO pins
DIRSET	0x518	DIR set register
DIRCLR	0x51C	DIR clear register
LATCH	0x520	Latch register indicating what GPIO pins that have met the criteria set in the PIN_CNF[n].SENSE registers
DETECTMODE	0x524	Select between default DETECT signal behavior and LDETECT mode
PIN_CNF[0]	0x700	Configuration of GPIO pins
PIN_CNF[1]	0x704	Configuration of GPIO pins

Figure 7: Registrene i GPIO-modulene (side 144 fra nrf52833 Product Specification.pdf).

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

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 M4-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 `DETECTMODE` (husk at disse registernavnene er spesifikt til GPIO-modulene! Andre moduler har andre registre.) Dette registeret starter på adresse `0x524`, som betyr at det okkuperer de fire adressene `0x524`, `0x525`, `0x526` og `0x527`. Den neste ledige adressen er `0x528`. 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 `DETECTMODE`:

$$700_{16} - 528_{16} = 1792_{10} - 1320_{10} = 472 \text{ byte} = 118 \text{ word} \quad (1)$$

I C, bruker man denne informasjonen på denne måten:

```
typedef struct{
    ...
    volatile uint32_t DETECTMODE;
    volatile uint32_t RESERVED1[118];
    volatile uint32_t PIN_CNF[32];
} NRF_GPIO_REG0;
```

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_REG0`, 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 GPIO0 sitt IN-register, kan man simpelthen bare skrive `GPIO0->IN`.

Husk at dette eksempelet baserer seg på databladet for en `nRF52833`. Ulike datablader for andre type mikrokontrollere kan ha ulik design, men mye av informasjonen er det samme.

A.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`.

B Appendiks - Bitoperasjoner i C

C er et godt egnet språk for mikrokontrollere fordi den ikke gjemmer bort tilgang til plattformspekifikke 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 bruker `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:

```
// GPIO0->IN is a register of 32 bits, and button A is held if
// the 14th bit is zero (zero-indexed)

int ubit_button_press_a(){
    return (!(GPIO0->IN & (1 << 14)));
}
```

```
}

// (1 << 14) gets us bit number 14, counting from 0
// & isolates the 14th bit in GPIO0->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 for the GPIO0-module */
GPIO0->IN & (1 << 12);
/* Checks if bit 2 and 3 in register IN is set for the GPIO0-module */
GPIO0->IN & (1 << 2) | (1 << 3);
```

C Appendiks - Kort om UART

Modulen for UART som finnes på nRF52833-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:bit-en. Grunnen til dette er at vi blir tvunget til å kommunisere gjennom nRF52820-brikken om vi ønsker å kunne tolke signalet som USB. Dette fører til at micro:bit-en 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.

D 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.

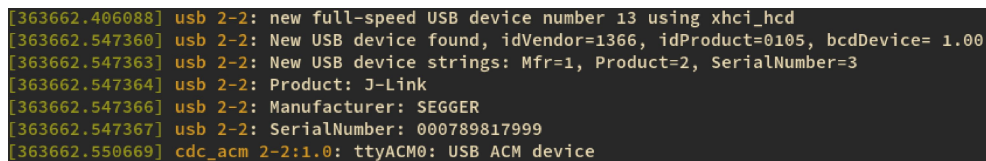
D.1 Vanlige feil ved bruk av `picocom`

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:bit`-en 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:bit`-en
2. Åpne en ny terminal, hvor dere kaller `"dmesg --follow"`.
3. Koble i `micro:bit`-en
4. Det skal nå komme opp en melding om en ny USB-enhet (se figur 8).



```
[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 8: Output fra terminalen.

5. Ta nå navnet som `micro:bit`-en ble tildelt av operativsystemet (i dette eksemplet har `micro:bit`-en fått navnet `"ttyACM0"`) og prøv det etter `"/dev/"` i `picocom`.

E Appendiks - Kort om I²C

I²C (eller bare I2C) er en av de vanligste bussprotokollene for tilpassede datasystemer. Den store fordelen 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 begrensning, ettersom dette er mer enn nok for et par sensorer koblet til en mikrokontroller.

Figur 9 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.

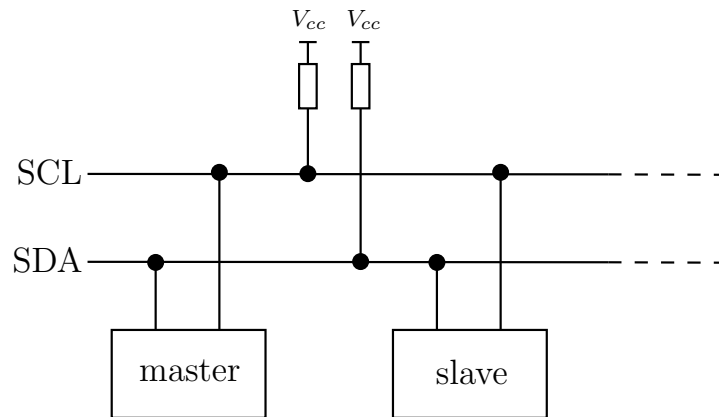


Figure 9: Oppkobling av en I2C-buss

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 10.

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

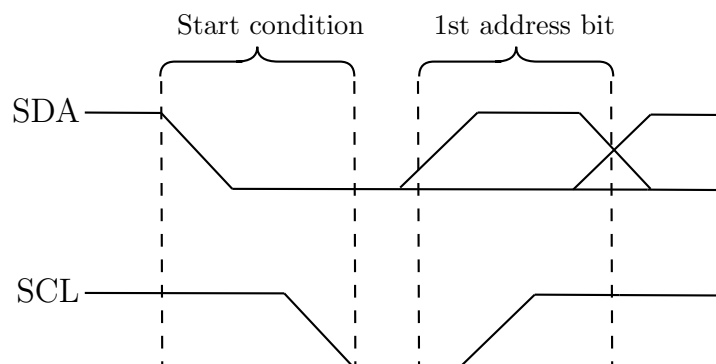


Figure 10: Start condition, og første adressebit, på I2C-buss.

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.

Adressering

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:bit-en) 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 **D**etection and **A**rbitration by **M**essage **P**riority). 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 adressebytet 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 (**C**ontroller **A**rea **N**etwork) 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 (**C**arrier **S**ense **M**ultiple **A**ccess with **C**ollision **D**etection and **A**rbitration by **M**essage **P**riority)

F Appendiks - Debugging med OpenOCD

For å debugge micro:bit-en kan vi bruke et program som heter [OpenOCD](#) (Open On-Chip Debugger). Dette programmet kommuniserer med interface MCUen, og gjør det mulig å hente ut nyttig informasjon direkte fra target MCU. Programmet skal allerede være installert på lab-PCene, men kan installeres med:

```
sudo apt install openocd
```

OpenOCD med GDB

For å starte programmet skriv inn kommandoen:

```
openocd -f interface/cmsis-dap.cfg -f target/nrf52.cfg
```

Denne kommandoen forteller OpenOCD at vi ønsker å snakke til en nRF52-chip, som er target MCUen vår. (`interface/cmsis-dap.cfg` forteller bare at vi ønsker å bruke programvaren som kjører på interface MCUen, og er ikke viktig for oss.) OpenOCD vil så koble til micro:bit-en og starte en GDB-server som vi ønsker å koble oss til.

Åpne så et nytt terminalvindu og naviger deg til oppgavemappen du ønsker å debugge. Bygg så programmet med `make debug`. Dette er viktig for at GDB skal kunne lese programmet vårt. Skriv deretter inn kommandoen:

```
/opt/arm-none-eabi-12.2/bin/arm-none-eabi-gdb .build_system/main.elf
```

Dette vil starte GDB og laste inn programmet du nettopp har bygget. Du kan erstatte `main.elf` med programmet du ønsker å debugge. Det siste steget er å koble seg til GDB-serveren OpenOCD har startet for oss. Dette kan vi gjøre med å skrive den følgende kommandoen i GDB:

```
target extended-remote localhost:3333
```

Du vil nå være koble til GDB-serveren, og vil kunne debugge micro:bit-en som om koden kjørte på din egen maskin. (Se Øving om GDB)

OpenOCD med VSCode

Mange foretrekker å debugge og programmere i en moderne IDE som VSCode. Vi har derfor laget et utviklingsmiljø som dere kan laste ned. Utviklingsmiljøet baserer seg på OpenOCD og utvidelsen [Cortex-Debug](#). For å bruke dette utviklingsmiljøet må du laste ned `vscode-utviklingsmiljøet` fra Blackboard og pakke ut filene i oppgavemappen du ønsker å debugge. Sørg for at `.vscode`-mappen ligger i samme mappe som `.build_system`. Begge disse mappene er ”skjulte”, som vil si at du må bruke `ls -a` for å se dem i et terminalvindu. Åpne så oppgavemappen i VSCode (dette kan du gjøre med kommandoen `code .`). VSCode vil så be deg om å laste ned utvidelsen `Cortex-debug`. Etter å ha installert denne kan du trykke på **Run and Debug**-fanen for å så debugge programmet. Se VSCode sin dokumentasjon via [denne lenken her](#) for bruk av debug-verktøyet.