

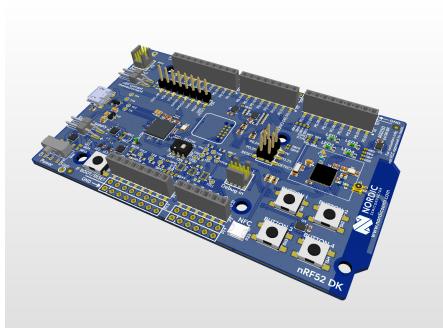


Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang
2022	Kiet Tuan Hoang
2023	Kiet Tuan Hoang Tord Natlandsmyr
2024	Terje Haugland Jacobsson Tord Natlandsmyr
2025	Kristian Blom

I Introduksjon - Kort om Nordic nRF52 DK

Nordic nRF52 Development Kit er et utviklingskort (se figur 1) utviklet av Nordic Semiconductor i Trondheim. Vi skal på denne laben bruke utviklingskortet til å utforske programvareutvikling med mikrokontrollere.



(a) 3D Render



(b) Forsiden med led-lys og knapper

Figure 1: Nordic nRF52 DK brukt i mikrokontrollerlaben.

Utviklingskortet er utstyrt med en Arm Cortex-M4 prosessor med en klokkefrekvens på 64 MHz, 512/256 kB flash og 64 kB RAM. Det er ingen hemmelighet at tilpassede datasystemer har blitt utrolig kraftig. En smart USB-C-lader i dag er

omtrent 500 ganger raskere enn Apollo 11 sitt navigasjonssystem¹. Utviklingskitet vi jobber med i denne laben har støtte for å kjøre et lite operativsystem, men for å få mest mulig om mikrokontrollere holder vi oss på et maskinnært abstraksjonslag, ofte kalt ”bare-metal”. Vi snakker ofte om abstraksjonslag i programvareutvikling. Enkelt forklart er hensikten med abstraksjonslag å forenkle noe ved å skjule unødvendige detaljer. Dette impliserer at å gå ned i abstraksjonslag vil introdusere kompleksitet. Dette vil dere få oppleve i denne laben. Funksjoner som kun trenger én linje kode på operativsystemnivå (f.eks. `Print()` i Python) vil kreve betydelig mer kode på en mikrokontroller. Fordelen med et lavere abstraksjonslag er at man har mer kontroll over hva prosessoren faktisk gjør.

I denne laben vil vi utforske hvordan de forskjellige abstraksjonslagene henger sammen ved å programmere mikrokontrollerens registre direkte. Dette gjøres i C, som er det laveste abstraksjonsnivået vi har tilgang til, uten å gå over til ARM-prosessorens instruksjonssett.

Hvis du ser nøye etter på utviklingskitet, vil du legge merke til at den har to chipper: en større merket med N5340, og en mindre merket med N52832. Den har nemlig to mikrokontrollere, også kalt MCU-er (**Microcontroller Unit**): En som kjører programvare utelukkende for å programmere og feilsøke hovedchippen, og en som faktisk kjører koden vår. Vi skiller mellom dem med navnene ”Target” og ”Interface”. Når du kobler til utviklingskitet til en datamaskin, vil datamaskinen kommunisere med interface MCU-en og oppdage den som USB-lagringssenhett. Interface MCU-en på nRF52 DK kjører SEGGER J-Link Onboard. Den brukes til å programmere og feilsøke Target MCU.

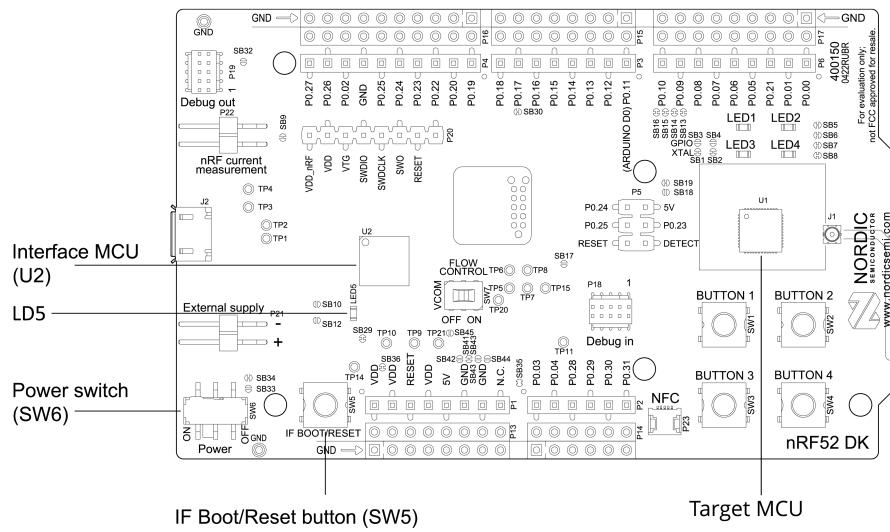


Figure 2: Nordic nRF52 DK ”Target” og ”Interface” MCU

¹Uttregnet av Forrest Heller i denne artikkelen her

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. Dere får også utdelt et utviklingsmiljø for feilsøking i VSCode. Utviklingsmiljøet er inkludert i hver oppgavemappe på Github. For mer info om utviklingsmiljøet se appendiks E og nrf52dk-environment på [Github](#).

III Introduksjon - Praktisk rundt laben

I denne laben brukes ARM GCC som verktøykjeden for programmering av mikrokontrolleren. Denne typen verktøykjede kalles åpen kilde (open-source), og er en del av GNU-prosjektet.

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 **.buid_system**. Denne inneholder det som skal til for å få koden til å kjøre på utviklingskitet.

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

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 kompile C-koden. Dette vil generere en **.hex**-fil som mikrokontrolleren kan kjøre i **build**-mappen. I tillegg til **make**, har denne **Makefilen** også fem andre mål; **make debug** vil komPILE C-koden med debug-flagg for feilsøking av programmet, **make erase** vil slette minnet til mikrokontrolleren, **make recover** vil slette minnet til mikrokontrolleren og deaktivere tilbakelesningsbeskyttelsen ("read-back") hvis den er aktivert, mens **make clean** vil slette ferdigkompilet kode og **hex**-filen fra datamaskinen. For å faktisk overføre **hex**-filen over i programminnet til mikrokontrolleren bruker vi målet **make flash**. **VIKTIG!:** Første gangen man programmerer et nytt utviklingskit må man bruke **make recover** og deretter **make flash**. Dette er fordi mikrokontrolleren kommer forhåndsprogrammert med et skriverbeskyttet program. **Uten dette steget vil ikke make flash fungere.**

III .2 Programmeringstaktikk

For å sette ønskede registre på nRF52832-en bruker vi et kjent triks fra C-programmering. Dette innebærer at man lager **structs**, som dekker nøyaktig det minnet man ønsker å endre - for dermed å *typecaste* en peker til starten av minnet inn i **struct**-en. Dette gjør det mulig å endre på det underliggende minnet ved å endre på **struct**-ens medlemsvariabler.

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 spesifik 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 nRF52 DK, gjelder [nRF52832 Product Specification](#) (denne finner dere i mappen **datablad**). 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 kort innføring i hvordan man leser og bruker databladet i kontekst av memory mapped IO før man begynner med oppgavene.

III .4 Førstegangsoppsett av utviklingsmiljø

Før man bruker nRF52 DK må man laste ned redskapene som trengs for å programmere den på Linux. Disse skal allerede være installert på PCene på Sandtidssalen. Dersom man ønsker å bruke en annen maskin må man installere avhengighetene beskrevet på [Github](#).

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 komplilatoren lukter 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, som oftest USB, (se oppgave 2) mellom utviklingskitet og datamaskinen for å bruke et seriell terminalprogram for å skrive ut feilsøkingsdata (f.eks. `sprintf`). Denne feilsøkingsstrategien er tilstrekkelig i de aller fleste tilfeller.

Den andre metoden er å bruke et debuggingsprogram som kommuniserer med Interface MCU-en (se figur 2). Dette lar deg blant annet inspirere og endre på registre live og kan være svært nyttig for å debugge mikrokontrollere. Denne metoden blir beskrevet i appendiks E og anbefales på det sterkeste.

1 Oppgave 1 - GPIO

1.1 Beskrivelse

I denne oppgaven skal vi skru på alle LED-ene i matrisen når knappen BUTTON 1 trykkes, og skru dem av når knappen BUTTON 2 trykkes. Dette gjøres med GPIO-modulene. Dette er moduler som har ansvarer 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 skumlese appendiks A og B. Utviklingsmiljøet kan også vise seg å være svært praktisk for å teste GPIO-modulen ved å sette registre live (se appendiks E).

I denne oppgaven, så trenger dere bare å endre på `main.c`. De spesielt interesserte kan se på mappen `.build_system` og `Makefile`. Sistenevnte 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/.vscode</code>	helst ikke
<code>1_gpio/Makefile</code>	helst ikke

1.2 Oppgave

LED-matrisen på nRF52 DK består av en 2x2 plassert rett over target MCU (se figur 2). Det er en GPIO-port assosiert med hver av LED-ene. Disser er aktivt lave, som vil si at vi må trekke porten lav dersom vi vil at LED-en skal lyse.

Til å starte oppgaven, ta en titt på den vedlagte filen `PCA10040_Schematic_And_PCB.pdf` i mappen `datablad`. Dette er referansedesignet for et nRF52832 DK. Finn først ut hvordan de to knappene BUTTON 1 og BUTTON 2 er koblet.

- Hvilke pinner på nRF52832-en brukes? Vil pinnene være høye eller lave dersom knappene trykkes?

Se deretter i databladet til nRF52832-serien ([nrf52832 Product Specification](#)).

- Hvordan ser minnekartet for mikrokontrolleren ut? Hva er baseadressen til GPIO-modulene? Bytt ut `__GPIO_BASE_ADDRESS0__` i `main.c` med den faktiske baseadressen.

I `main.c` vil dere se at det er definert et `struct` som heter `NRF_GPIO_REGS0`. Dette `struct`-et representerer alle registrene til GPIO-modulen. Ved å typecaste adressene til GPIO-modulen inn i `struct`-en, kan vi så endre på `struct`-en 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__)
```

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

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

Dere vil også se at medlemsvariabelen `RESERVED0` i `GPIO0` er en array av type `volatile uint32_t` med 321 elementer. Dette er fordi databladet forteller oss at `OUT`-registeret i modulen `GPIO0` har et offsett på $0x504$ (504_{16}) fra modulens baseadresse. 504_{16} er det samme som 1284_{10} . Altså er det 1284 byte mellom baseadressen og `OUT`-registeret. Siden vi bruker en ordstørrelse (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.

Deretter må dere fylle inn resten av `button_init()`. Se på side 136 databladet for å se hva denne konfigurasjonen gjør, og om det stemmer med hvordan knappene er koblet til portene til Target MCU.

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 `BUTTON 1`, og skrur seg av når vi trykker på knapp `BUTTON 2`. 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

- Det er fort gjort å forveksle GPIO med GPIOTE-modulen (**GPIO Tasks and Events**). Sistnevnte brukes for å lage et hendelsesbasert system, og brukes ikke i denne oppgaven.
- 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*.
- Er knappene aktivt lav eller høy? Hvorfor trenger vi en pull-up på knappene?
- Sjekk ut appendiks [B](#) for hvordan man kan manipulere bits.
- Sjekk ut appendiks [A](#) 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 nRF52 DK. 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 Sandtidssalen finnes det en DSUB9-port som vi kunne brukt til dette, men i denne øvingen trenger vi ikke det.

Som nevnt i introduksjonen kommuniserer vi med target MCU gjennom en interface MCU. Dette er en nRF5340 mikrokontroller som lar oss programmere nRF52832-SoCen 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 nRF52 DK.

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.

Filer	Skal denne filen endres?
<code>2_uart/gpio.h</code>	nei
<code>2_uart/.build_system</code>	helst ikke
<code>2_uart/.vscode</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 PCA10040_Schematic_And_PCB.pdf.

- Finn ut hvilken pinne fra nRF52832-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 deretter 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 mikrokontrolleren. De må derfor inkluderes fra `main.c`.

I implementasjonsfilen (`uart.c`) skal vi igjen bruke memory mapped IO, slik vi gjorde for `GPIO0` 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 akkuratt 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 konfiguere 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 PCA10040_Schematic_And_PCB.pdf, vil dere se at vi ikke har noen CTS- eller RTS-koblinger fra nRF52-brikken til interface-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 163 (*UART Transmission* i side 536) 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.

- I tillegg må dere sette EVENTS_TXRDY lik 0. Dette er for å *clear* interruptet som genereres når dere er ferdige å sende.

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 unngå 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 utviklingskitet til å sende A om knappen BUTTON 1 trykkes, og B om BUTTON 2 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. Det kan også hende at datamaskinen velger en annen port, f.eks. `/dev/ttyACM1` eller `/dev/ttyACM2`.

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

2 .4 Mottaksfunksjon

Deretter, lytt etter sendte pakker på utviklingskitet. Om datamaskinen har sendt en bokstav, skal mikrokontrolleren 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`. Standardoppførselen til `picocom` er å sende alle bokstaver som skrives inn i terminalen når det kjører. Bokstavene vil derimot ikke bli skrevet til skjermen, så dere vil ikke få noen visuell tilbakemelding på datamaskinen (gitt at dere ikke manuelt sender bokstaven tilbake). 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 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 `fprintf`.

Når `fprintf(...)` 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`).

Vi kan lage mange varianter av slike skrivefunksjoner dersom vi har et komplekst system med mange skriveenheter – eller om vi har flere tråder. Denne arkitekturen har bare én kjerne og vi vil bare bruke `UART`, så vi kan fint implementere en global variant av denne skrivefunksjonen. For å gjøre dette, legger vi til følgene i `main.c`:

```
#include <stdio.h>
#include <sys/types.h> // For ssize_t
[...]
```

```

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 eksempelvis skrive:

```
iprintf("The average grade in TTK%d was in %d was: %c\n\r", 4235
, 2022, 'B')
```

Om `picocom` da forteller dere gjennomsnittskarakteren i tilpassede datasystemer i 2022, så har dere fullført oppgaven.

2 .6 Oppgave - `_read()` (Frivillig)

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 main-filen:

```

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 utviklingskitet, 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 side 537 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: GPIOE og PPI

3.1 Beskrivelse

Akkurat nå jobber vi med en mikrokontroller som er basert på en ARM Cortex M4 prosessor som bare har en kjerne. Vi har derfor ikke mulighet til å kjøre kode i sann parallelisering. En mulighet er å bytte veldig fort mellom to eller flere oppgaver (også kalt fibre) samtidig, men dette kan være problematisk om man trenger nøyaktige tidsfrister for programmene vi skriver.

For å løse dette problemet, har nRF52832-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 register. 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 nRF52832-en har noen form for oppgaver og hendelser. For å knytte disse til GPIO-pinnene, har vi en egen modul kalt **GPIOE** (**G**eneral **P**urpose **I**nput **O**utput **T**asks and **E**vents). I denne oppgaven skal vi bruke **GPIOE**-modulen til å definere en hendelse (**BUTTON 1** trykket), og fire oppgaver (skru på eller av LED-matrisen).

I denne oppgaven, så får dere igjen utlevert ferdig `gpio.h`. I tillegg, så får dere halvferdige `.h`-filer for PPI og **GPIOE**-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/.vscode	helst ikke
3_gpiote/Makefile	ja

3 .2 Oppgave - Grunnleggende GPIOTE og PPI

Først skal LED-matrisen konfigureres og lysene skrus av. Husk at pull-up må konfigureres.

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

Fem GPIOTE-kanaler skal brukes.

- Bruk en kanal til å lytte til BUTTON 1. Denne kanalen skal generere en hendelse når knappen trykkes.
- De resterende kanalene skal alle være konfigurerert som oppgaver, og koblet til hver sin forsyningsspinne for LED-matrisen. Forsyningsspenningen skal veksle hver gang oppgaven aktiveres. Hvilken initialverdi disse GPIOTE-kanalene har er opp til dere.

3 .2.2 PPI

For å koble BUTTON 1-knapphendelsen til forsyningsoppgavene, trenger vi fire PPI-kanaler; en for hver forsyningsspinne. 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[4]);
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[4]` - 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 BUTTON 1 trykkes - uavhengig av hva CPU-en gjør. Test dette ut ved å lage en evig løkke hvor CPU-en ikke gjør noe nyttig arbeid (altså tom).

Når dere har kompilert og flashet programmet over til utviklingskitet, 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*.

Ideelt sett, ville spenningen til BUTTON 1 sett ut som en spenningskurven til en ideell bryter (se figur 3). I virkeligheten vil de mekaniske platene i bryteren gjen-att slå mot-, og sprette fra hverandre. Når dette skjer, får vi spenningskurven for den reelle bryteren i figur 3. I dette tilfellet kan CPU-en registrere spenningstransienten som raske knappetrykk.

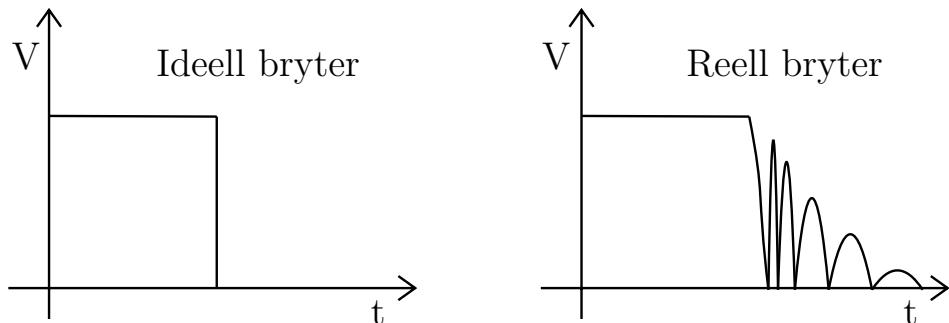


Figure 3: Spenningen over en ideell- og en reell bryter.

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

1. Vi har tactile pushbuttons på utviklingskitet. Disse er mye bedre på å redusere bounce enn andre typer knapper.
2. Utviklingskitet har debouncer-kretser for hver bryter, som reduserer problemet.
3. 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 grunnen til at dere sannsynligvis ikke hadde dette problemet da dere brukte GPIO-modulen.

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 fire PPI-kanaler totalt).

- GPIOTE-kanalene trenger ingen eksplisitt aktivering fordi MODE-feltet i CONFIG-registeret automatisk tar hånd om pinnen for dere.

A Appendiks - Grunnleggende databladkunnskaper

For enhver mikrokontroller er det viktig å kunne mestre bruken av datablade. Mer spesifikt, er det veldig viktig for å forstå hvordan man bruker det som kalles memory mapped IO. I praksis, betyr memory mapped IO at man typecastar 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 pensum litteratur 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 0x50000000 (se figur 4).

20.3 Registers

Table 31: Instances

Base address	Peripheral	Instance	Description	Configuration
0x50000000	GPIO	GPIO	General purpose input and output	Deprecated
0x50000000	GPIO	P0	General purpose input and output	

Figure 4: Startsadressene til GPIO-modulene (side 116 fra [nrf52832 Product Specification.pdf](#)).

Noen moduler kan ha flere *instanser*. Et annet eksempel på dette er Timer-modulen til nRF52-en. Der finnes det fem forskjellige kopier av samme enhet (se figur 5). Dette er veldig nyttig dersom man ønsker for eksempel flere uavhengige klokker.

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

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

Denne kodesnutten tilsvarer å definere 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 6).

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

Table 45: Instances

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])

Figure 5: Startadressene til Timer-modulen (side 239 fra nrf52832 Product Specification).

Table 32: Register Overview

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 behaviour and LDETECT mode
PIN_CNF[0]	0x700	Configuration of GPIO pins

Figure 6: Registrene i GPIO-modulene (side 117 fra nrf52832 Product Specification).

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:

```
typedef struct{
    volatile uint32_t RESERVED0[321];
    volatile uint32_t OUT;
    ...
} NRF_GPIO_REG0;
```

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 ta opp 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 adreslene 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 startsadressen 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 i mål. Da kan man direkte få tilgang til modulens registre ved å dereferere pekeren. Eksempelvis, dersom man har lyst til å lese GPIO0 sitt IN-register, kan man simpelthen bare skrive GPIO0->IN.

Husk at dette eksempelet baserer seg på databladet for en nRF52832. 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 plattsformspesifikke 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 likhetstegegn 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 kodesnuttene 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 nRF52832-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**equest **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 utviklingskitet. 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 utviklingskitet 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 utviklingskitet 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 utviklingskitet
2. Åpne en ny terminal, hvor dere kaller ”`dmesg --follow`”.
3. Koble i utviklingskitet
4. Det skal nå komme opp en melding om en ny USB-enhet (se figur 7).

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

Figure 7: Output fra terminalen.

5. Ta nå navnet som utviklingskitet ble tildelt av operativsystemet (i dette eksemplet har utviklingskitet fått navnet ”`ttyACM0`”) og prøv det etter ”`/dev/`” i `picocom`.

E Appendiks - Debugging av mikrokontrollere

Her vil du få en rask introduksjon om hvordan debugging av mikrokontrollere fungerer, med et spesielt fokus på debugging av Nordic nRF52 DK i et moderne utviklingsmiljø.

E.1 Serial Wire Debug (SWD)

Når vi debugger et program som kjører på datamaskinen vår, har vi direkte tilgang til prosessoren vi programmerer. På mikrokontrollere må vi kommunisere gjennom ett eller flere kommunikasjonslag for å hente informasjon fra prosessoren til mikrokontrolleren. Mikrokontrollere (som nRF52832 SoC) har som regel et grensesnitt for å dele informasjon med eksterne enheter. Dersom vi ser på PCA10040_Schematic_And_PCB, kan vi legge merke til to pins, SWDIO (26) og SWDCLK (25), som går fra target MCU (nRF52832) og videre til interface MCU (nRF5340). Disse to portene betegner det fysiske grensesnittet for Serial Wire Debug (SWD). Dette er en protokoll som definerer hvordan vi kan programmere prosessoren vår (ARM Cortex-M4), men også hvordan vi kan hente informasjon fra prosessoren.

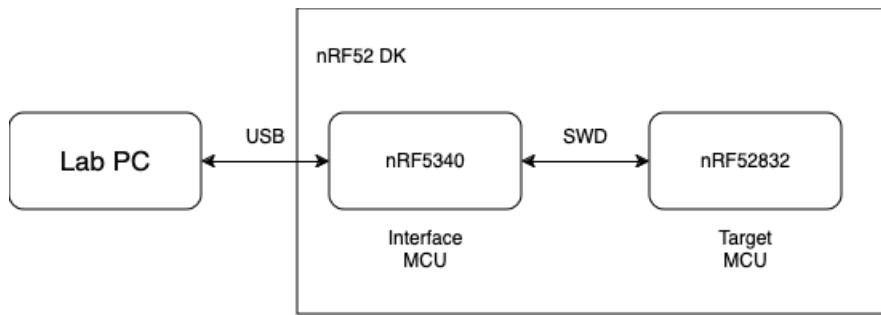


Figure 8: Visualisering av kommunikasjonslag

For å få videresende informasjonen vi får fra mikrokontrolleren må vi bruke et program på interface MCU som fungerer som et kommunikasjonslag mellom vår datamaskin og target MCU. nRF52 DK sin interface MCU (nRF5340) er utstyrt med J-Link, som er et kraftig verktøy for feilsøking og programmering av target MCU (nRF52832). Den innebygde feilsøkingsprogrammet støtter feilsøking i monitormodus, en funksjon som gjør det mulig for utviklere å samhandle med mikrokontrollerens kjøringsmiljø i sanntid. Dette er spesielt nyttig ved feilsøking av komplekse problemer som krever mulighet til å inspisere og endre mikrokontrollerens tilstand (registerverdier etc.) mens koden kjører. Enkelt forklart blir denne informasjonen delt med interface MCU, som kan tolke og videresende informasjonen over til vår maskin over USB (se figur 8).

E.2 GDB Server

For å gjøre nytte av informasjonen vil blir sendt av den innebygde debuggeren (interface MCU) på mikrokontrolleren bruker vi tjener som kan tolke informasjonen

for oss. I vårt tilfelle er dette en GDB-server. Dette gjør slik at andre programmer, som f.eks. VSCode (eller en vanlig GDB session) kan kommunisere med mikrokontrolleren over TCP/IP-forbindelse. ‘Server’-enheten seg av kommunikasjon med både GDB, og hardware, ved å abstrahere bort hvilken type forbindelse man har mellom datamaskinen og plattformen som blir debugget. I prinsippet kan ‘Server’ være hva som helst, så lenge det er et program som støtter kommandoer fra GDB, og er i stand til å kommunisere med målhardwaret.

Debugging med VSCode

Mange foretrekker å debugge og programmere i en moderne IDE som VSCode. Vi har derfor laget et utviklingsmiljø vedlagt i hver oppgave. Utviklingsmiljøet kan også lastes ned ved [git clone via denne lenken her](#). Utviklingsmiljøet baserer seg på utvidelsen [Cortex-Debug](#), som bruker J-Link GDB for å kommunisere med mikrokontrolleren over USB. 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. For mer informasjon om hvordan man debugger nRF52 DK i VSCode, ta en titt på Github-repositoryet [nrf52dk-environment](#).

E.3 Laste opp eksempelkode

I det utdelte utviklingsmiljøet [nrf52dk-environment](#) ligger det eksempelkode som dere kan kompile og flashe ved å kalle `make` og `make flash`. Koden vil blinke LED 1 med jevne intervaller.