Mikrokontrollerlab med micro:bit

TTK4235 Tilpassede Datasystemer

1 Bakgrunn

1.1 Introduksjon til micro:bit

BBC micro:bit er et lite kort utviklet primært for å vekke interesse for programmering hos barn. For å gjøre dette mulig har man også laget et særs vennlig webgrensesnitt der man kan programmere kortet, simpelthen ved å trekke og slippe "programmeringsklosser" der man vil. Dette er selvsagt ideelt for å vise små barn at vi kan ha det gøy på Kyb, men utover dette får vi ingen kontroll over hva som skjer på et lavere nivå enn nettleseren.

Litt lengre ned på abstraksjonsstigen har vi micro:bit DAL (Device Abstraction Layer). Denne kan programmeres via MicroPython eller JavaScript, men dette vil samtidig spise opp mye av ressursene tilgjengelig på det lille kortet. Eksempelvis vil man typisk kun ha omlag 2 KB SRAM-område tilgjengelig hvis man benytter seg av MicroPython; av de 16 KB kortet i utgangspunktet har.

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

For å få en bedre forståelse for hvordan en micro:bit fungerer, skal vi derfor fjerne DALen fra kortet og programmere prosessorens registre direkte. Prosessoren på kortet er en ARM Cortex M0, som er integrert inn i en Nordic Semiconductor nRF51822 SoC. Dette skal vi gjøre i C, som er det laveste abstraksjonsnivået vi kan få, uten å gå over til Thumb - som er prosessorens instruksjonssett.

1.2 Introduksjon til labformatet

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

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

I mappen dere har fått utdelt ligger det en gjemt undermappe ved navn ".build_system". Denne inneholder det som skal til for å få koden deres til å kjøre på micro:biten. Det er ikke meningen at dere skal endre noe i denne mappen, men om dere vil forstå hvordan koden deres henger i hop med hva som skjer på kortet, er det bare å ta en titt.

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

1.2.1 Makefil

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

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

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

1.2.2 Programmeringstaktikk

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

1.3 Datablad

Innvevde datasystemer er ganske 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 knappecellebatteri. Derfor må vi glemme en del generelle ting som gjelder uavhengig av plattform, og fokusere på ting som kun gjelder plattformen vi arbeider på. Det er her datablad kommer inn.

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

Gjennom dette emnet vil **nRF51 Series Reference Manual**[1] stort sett være databladet vi skal bruke. Dette er et godt datablad som også er forholdsvis kort (datablad kan lett bli over tusen sider), så det er en god introduksjon til måten man jobber med innvevde systemer.

2 Førstegangsoppsett

Aller først må vi sette opp de redskapene vi trenger for å kunne programmere micro:biten som vi vil. Seksjon 2.1 setter opp verktøykjeden på datamaskinen, mens seksjon 2.2 dekker hvordan dere klargjør micro:biten.

2.1 Software

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

For å se om alle verktøyene er på plass før vi starter, kan dere kalle `nrfjprog --version` fra kommandolinjen. Om denne kommandoen svarer med en versjon for både `nrfjprog` og `JLinkARM`, trenger dere ikke gjøre noe her.

2.1.1 arm-none-eabi-gcc

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

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

2.1.2 nrfjprog

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

1. Gå til infocenter.nordicsemi.com.
2. Under `nRF Tools` velger dere `nRF5x Command Line Tools`, og så `Installing the nRF5x Command Line Tools`.

[1] http://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.pdf

3. Velg nedlastingen for Linux 64-bit. Dette vil ta dere til en side hvor dere kan laste ned siste versjon av verktøyet.

4. Når dere har lastet ned `.tar`-filen, navigerer dere til mappen dere lastet den ned til, via terminalen.

5. Brett ut tararkivet ved å kalle `tar -xf nRF5x-Com[...]x86_64.tar`

6. Kall så `sudo chown root:root -R nrfjprog`. Dette vil gi eierskap over mappen `nrfjprog` til rotbrukeren.

7. Kall deretter `sudo mv nrfjprog/* /usr/bin`. Dette vil flytte alt innholdet i nrfjprogmappen inn i `/usr/bin` - som er et sted inkludert i miljøvariabelen `$PATH`. Det betyr simpelthen at dere kan kjøre programmet fra hvor som helst uten å måtte spesifisere hvor det ligger.

2.1.3 JLinkARM

Til slutt må `nrfjprog` vite hvordan det skal snakke med programvaren på micro:biten. Etter seksjon 2.2, vil denne programvaren være JLink, så vi må her finne ut hvordan vi får `nrfjprog` til å snakke JLink.

1. Gå til segger.com/downloads/jlink.

2. Under `J-Link Software and Documentation Pack`, finner dere alternativet `DEB installer 64-bit`.

3. **NB:** Velg `Older versions`, fordi versjon **V6.30f** i skrivende stund har en bug der dere mister forbindelse med micro:biten hver gang dere kjører `nrfjprog`.

4. Velg versjon **V6.30** (altså uten noen bokstav bak).

5. Via terminalen, gå til mappen dere lastet ned til.

6. Kall `sudo dpkg -i JLink_Linux[...]x86_64.deb`

Til slutt kan dere nå kalle `nrfjprog --version` igjen, for å se om alt er installert riktig.

2.2 Hardware

Før vi kan programmere micro:biten må vi kvitte oss med DALen. Last ned BBC micro:bit J-Link OB Firmware[2] fra Segger.com. Dette er en hex-fil som vi skal legge inn på micro:biten for å få tilgang til programmering via Nordics flashverktøy, **nrfjprog**, og bedre debuggingsmuligheter enn vi har fra før.

[2] www.segger.com/downloads/jlink#BBC_microbit

1. Hold inne Reset på micro:biten. Dette er knappen på siden med logoen til micro:bit. Mens knappen holdes inne, koble inn USBen.
2. Det skal nå komme opp en enhet med navn "MAINTENANCE" på datamaskinen. Nå kan Reset slippes.
3. Trekk hex-filen fra Segger inn i "MAINTENANCE".
4. Etter at hex-filen er ferdig overført, vil micro:biten automatisk tilkobles på nytt. Denne gangen som "MICROBIT".
5. Kjør kommandoen `nrfjprog -f nrf51 -e` fra terminalen.
6. LED-matrisen skal nå slutte å lyse.

3 Oppgave 1: GPIO

3.1 Beskrivelse

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

Denne laboppgaven vil være litt kokebok, for å introdusere noen konsepter dere skal bruke senere. Deretter vil det bli gradvis mindre håndholding.

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

3.2 Oppgave

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

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

I mainfilen vil dere se at det er definert en struct ved navn `NRF_GPIO_REGS`. Denne structen representerer alle registrene til GPIO-modulen. Ved å *typecaste* adressen til GPIO-modulen inn i structen, kan vi så endre på structens

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

a) micro:bit LED matrise

0	2	4	19	18	17	16	15	11	P13
14	10	12	1	3	23	21			P14
22	24	20	5	6	7	8	9	13	P15
P4	P5	P6	P7	P8	P9	P10	P11	P12	

b) Implementasjon i hardware

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

medlemsvariabler for å skrive til registrene. Det er nettopp dette som er formålet med kodelinjen

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

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

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

Dere vil også se at medlemsvariabelen `RESERVED0` er en array av type `volatile uint32_t` med 321 elementer. Dette er fordi databladet forteller oss at `OUT`-registeret har en offset på 0x504 (504_{16}) fra GPIO-modulens baseadresse. 504_{16} er det samme som 1284_{10}. Altså er det 1284 byte mellom baseadressen og `OUT`-registeret. Siden vi bruken en ordstørrelse på 32 bit, deler vi dette tallet på fire, ettersom 32 bit er 4 byte. Dermed har vi $1284/4 = 321$.

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

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

3.3 Hint

1. nRF51822 har en GPIO-modul, og en GPIOTE-modul (**G**eneral **P**urpose **I**nput/**O**utput **T**asks and **E**vents). Sistnevnte brukes for å lage et hendelsesbasert system. Vi skal i første omgang kun bruke GPIO-modulen.

2. Om dere skriver inn GPIO-modulens baseadresse i base 16 (hexadecimal), må dere huske "0x" foran adressen. Hvis ikke vil kompilatoren tro dere mener base 10.

3. Når dere skal finne `__RESERVED1_SIZE__`, så husk at `DIRCLR` starter på 0x51C, som betyr at den byten slutter på 0x51F. Altså så starter ikke `RESERVED1` på 0x51C, men på 0x520.

4 Oppgave 2: UART

4.1 Beskrivelse

I denne oppgaven skal vi sette opp toveis kommunikasjon mellom datamaskinen og micro:biten. Dette skal vi gjøre ved å bruke UART, som står for **U**niversal **A**synchronous **R**eceiver-**T**ransmitter. Tradisjonelt ble signalene sent mellom to UART-moduler båret over et RS232 COM grensesnitt.

Nå har det seg derimot slik at moderne datamaskiner ikke lages med en slik port. Labdatamaskinene på Sanntidssalen har en DSUB9-port, som vi kunne brukt, men i dette tilfellet er vi heldigere enn som så.

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

4.2 Kort om UART på nRF51 og micro:bit

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

Dermed er vi totalt oppe i fire linjer: `RXD` (Mottakslinje), `TXD` (Sendelinje), `CTS` (**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 ganske imponerende med tanke på at "vanlig" UART-hastighet ligger på 115200 bit per sekund.

Uheldigvis er ikke alt bare fryd og gammen, rett og slett fordi vi blir tvunget til å snakke gjennom Freescalechipen om vi ønsker å kunne tolke signalet som USB. Grunnen til at dette er problematisk, er at micro:biten bare kobler to UART-linjer mellom de to chipene, så vi må holde oss til UART uten flytkontroll. Dette betyr at den høyeste baudraten vi pålitelig kan sende med, er 9600 bit per sekund. Stort sett vil micro:biten kunne håndtere høyere rater enn dette, men om dere ønsker minimalt med pakketap, er det lurt å holde seg til 9600 baud. Forutsatt at dere setter pakkestørrelsen til 8 bit, og bruker 2 stoppbit, vil dette uansett gi dere en overføringshastighet på omlag 800 bokstaver per sekund - som burde være mer enn nok.

4.3 Oppgave

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

vært opp til oss.
Slik er det altså ikke, så vi må ta en titt i "schematics.pdf". Finn ut hvilken pinne fra nRF51822-chipen som er `TGT_RXD`, og hvilken pinne som er `TGT_TXD`. Disse pinnene skal vi 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();
```

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

```
#include <stdint.h>

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

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

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

4.3.1 `void uart_init()`

Denne funksjonen skal først konfigurere de nødvendige GPIO-pinnene som input/output. Derfor må "gpio.h" inkluderes i "uart.c". Denne filen er allerede skrevet for dere.

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

Om dere ser i "schematics.pdf", vil dere se at vi ikke har noen CTS- eller RTS-koblinger fra nRF51-chipen (**C**lear **T**o **S**end og **R**equest **T**o **S**end). Dette betyr at vi ikke er i stand til å gjøre flytkontroll i hardware. Dermed står vi i fare for å miste pakker om vi prøver å sende for fort. Velg derfor en baudrate på 9600. "Baudrate" forteller simpelthen hvor mange bit vi sender per sekund.
Av samme grunn må vi fortelle UART-modulen at vi ikke har CTS- og RTS-linjer. Sett opp de riktige registrene for dette.

Til slutt skal vi gjøre to ting. Først må vi skru på UART-modulen, som

gjøres via et eget `ENABLE`-register. Deretter skal vi starte å ta imot meldinger, sett derfor `STARTRX` til 1.

4.3.2 `void uart_send(char letter)`

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

4.3.3 `char uart_read()`

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

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

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

4.4 Sendefunksjon

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

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

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

NB: Den siste bokstaven bak ”ACM” er et null-tegn.

4.5 Mottaksfunksjon

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

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

4.6 Mer avansert IO

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

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

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

Vi vil heller inkludere `<stdio.h>` og bruke en heltallsvariant av `printf`, kalt `iprintf`. Nå har vi derimot ett problem: `printf` vil i utgangspunktet snakke med en strøm som heter `stdout`, som tradisjonelt sett peker til en terminal.
Mikrokontrolleren vår har ingen skjerm - og dessuten er ikke `printf` egentlig implementert ferdig for oss på denne plattformen. Dette er fordi vår variant av `<stdio.h>` kommer fra et bibliotek kalt `newlib` - som er skrevet spesielt for innvevde datamaskiner.
Når `printf(...)` kalles, vil et annet funksjonskall til `_write_r(...)` skje i bakgrunnen. Denne funksjonen vil deretter kalle `ssize_t _write(int fd, const void * buf, size_t count)`, som foreløpig ikke gjør noe. Grunnen til at denne finnes, er at den trengs for at programmet skal kompilere, men den er i utgangspunktet tom, fordi vi gir lenkeren flagget `--specs=nosys.specs`, som dere kan se i Makefilen.

Gjennom `newlib` kan vi faktisk lage mange varianter av slike skrivefunksjoner, om vi har et komplekst innvevd 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 det, legger vi til følgene i mainfilen:

```
#include <stdio.h>

[...]
```

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

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

Prøv å kalle `iprintf("Norway has %d counties.\n\r", 18)`. Om `picocom` da forteller dere at Norge består av 18 fylker, så har dere greid oppgaven.

4.7 Frivillig: `_read()`

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

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

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

        *str = letter;
        return 1;
}
```

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

4.8 Hint

1. Det skal være totalt 11 reserverte minneområder i UART-structen. De skal ha følgene størrelser: 3, 56, 4, 1, 7, 110, 93, 31, 1, 1, 17.

2. Det er ingen forskjell på *tasks*, *events* og vanlige registre. Når LSB er satt i et *event*-register, har en hendelse skjedd. Når LSB settes i et *task*-register, startes en oppgave.

3. `int fd` i `_read` og `_write` står for "file descriptor". Den er der i tilfelle noen vil bruke `newlib` i forbindelse med et operativsystem. Vi trenger derimot ikke tenke på det.

4. Husk å legge til "uart.c" i Makefilen, bak `SOURCES := main.c`.

5 Oppgave 3: GPIOTE og PPI

5.1 Beskrivelse

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

For å løse dette problemet, har nRF51822en noe som kalles **P**rogrammable **P**eripheral **I**nterconnect. Dette er en teknologi som lar oss direkte koble en perifer enhet til en annen, uten at vi trenger å snakke med CPUen. For å dra nytte av denne teknologien må vi innføre *oppgaver* og *hendelser* - eller *tasks* og *events*. Oppgaver og hendelser er egentlig bare registre, men brukes annerledes enn "vanlige" registre.
Om et hendelsesregister inneholder verdien 1 har en hendelse inntruffet; om registeret inneholder verdien 0, har hendelsen ikke inntruffet. Oppgaveregistre er knyttet til en gitt oppgave (derav navnet), som kan startes ved å skrive verdien 1 til det. Oppgaver kan derimot ikke stanses ved å skrive verdien 0 til samme register som startet oppgaven.

De fleste perifere enhetene som finnes på nRF51822 har noen form for oppgaver og hendelser. For å knytte oppgaver og hendelser til GPIO-pinnene, har vi en egen modul kalt GPIOTE - som står for **G**eneral **P**urpose **I**nput **O**utput **T**asks and **E**vents.

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

5.2 Oppgave

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

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

5.2.1 GPIOTE

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

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

5.3 PPI

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

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

Denne kodesnutten vil sette registeret **E**vent**E**nd**P**oint for PPI-kanal 0 til adressen av `GPIOTE-IN[3]` - typecastet til en `uint32_t`. Tilsvarende vil den sett registeret **T**ask**E**nd**Point** for PPI-kanal 0 til adressen av `GPIOTE->OUT[0]`, etter å ha typecastet denne til en `uint32_t`.

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

5.4 Opphold CPUen

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

Når dere har kompilert og flashet programmet, skal LED-matrisen fungere som beskrevet. Det kan allikevel hende at matrisen ved enkelte knappetrykk

blinker fort av og på, eller ikke veksler i det hele. Grunnen til dette er et fenomen kalt *inputbounce*.

5.4.1 Debouncing av input

I en perfekt verden ville spenningen til A-knappen sett ut som spenningskurven til den ideelle bryteren i figur 2. I virkeligheten vil de mekaniske platene i bryteren gjentatt slå mot-, og sprette fra hverandre. Når dette skjer, får vi spenningskurven illustrert for den reelle bryteren i figur 2. I dette tilfellet kan CPUen registrere spenningstransienten som raske knappetrykk.

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

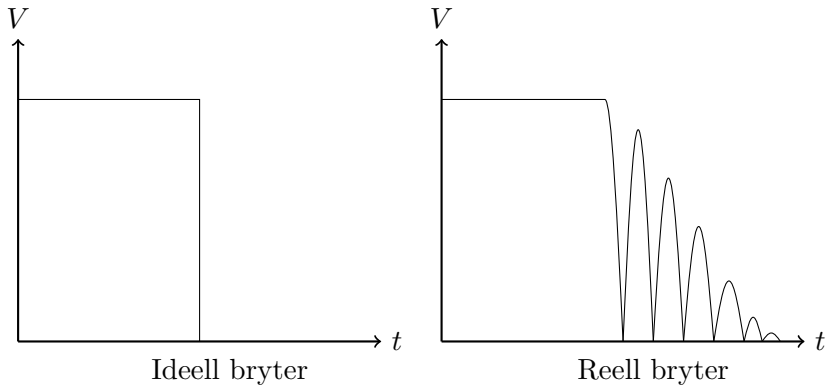

Figure 2: Spenningen over en ideell- og en reell bryter.

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

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

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

5.5 Hint

1. Husk å aktivere hver PPI-kanal. Når de er konfigurert riktig, aktiveres de ved å skrive til `CHENSET` i PPI-instansen.

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