

Mikrokontrollerlab i Linux

TTK4235 Tilpassede Datasystemer

1 Bakgrunn

1.1 Læringsutbytte

Denne laben er delt i tre deler, som hver skal gi en introduksjon til innnevde datasystemer, som mikrokontrollere. Del 1 vil introdusere dere til mikrokontroller IO og kommunikasjon mellom datamaskin og mikrokontroller via USART¹. Del 2 vil ta for seg lesing av analoge signaler via en Analog/Digital-omformer (ADC). Del 3 vil bruke joysticken på kortet til å kontrollere en servomotor via PWM.

1.2 Datablad

For å få en mikrokontroller til å gjøre noe interessant setter man verdier i interne registre (via *software*), som igjen dikterer hvordan *hardware* vil oppføre seg. For å finne hvilke registre man skal sette, tar man en titt i mikrokontrollerens datablad; der står som regel alt man trenger å vite. Haken er selvsagt at datablad er svære beist som lett kan vokse seg over tusen sider, og vel så det. Derfor er det viktig å kunne finne frem til det man trenger av informasjon, slik at man unngår et nervøst sammenbrudd på lab. Heldigvis har datablader gode innholdsfortegnelser, og i Atmels tilfelle også en nyttig **registeroversikt** på slutten av hvert kapittel. Bruk denne oversikten flittig!

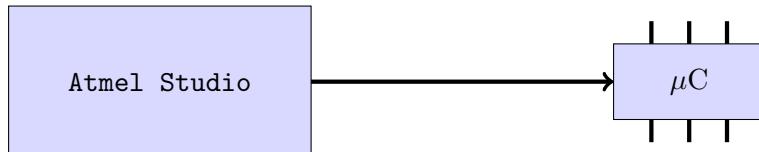
Datablad for mikrokontrolleren brukt i denne laben (AT90USB1287) ligger på it's learning, men kan også finnes på Atmel sine sider.

1.3 Linux vs. Windows

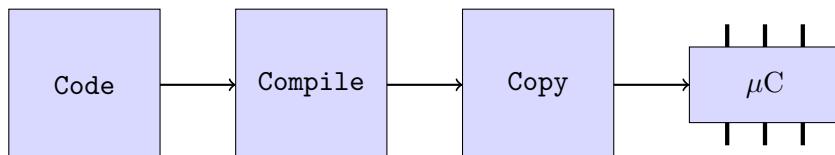
Dere kan selv velge om dere vil bruke Windows eller Linux i denne laben. Stort sett kan forskjellene oppsummeres som i figur ??: På Windows håndterer Atmel Studio alt, mens i Linux er det fritt frem. Altså er koden og sluttresultatet likt, men om man velger Windows er man mer eller mindre låst til Atmel Studio. I Linux har man større valgmulighet - er man misfornøyd med ett av stegene involvert; står man fritt til å velge annen software som gjør samme jobb.

¹Universal Synchronous/Asynchronous Receiver/Transmitter

Windows:



Linux:



Figur 1: Windows workflow vs. Linux workflow.

Den viktigste forskjellen er at man står fritt til å velge tekstredigeringsprogram. Hvis man er kjent med for eksempel Vim eller Emacs, er det ingen grunn til å lide under Atmel Studios klønnete IDE.

1.4 Ærlige ord basert på erfaring

Innnevde datasystemer, eller *embedded computers*, er vanskelig å komme inn i første gang. Mest sannsynlig kommer dere til å hate livet litt før dere blir kjent med databladet, og før dere kommer inn i *mikrokontrollerparadigmet*. Dette er helt normalt, så fortvil ikke.

Laben blir ganske kul i del 3 - da faller delene på plass :)

2 Utstyr

2.1 Hardware

På starten av hver lab får dere utlevert en eske med utstyr. Denne skal inn igjen på slutten av hver lab.

2.2 Software

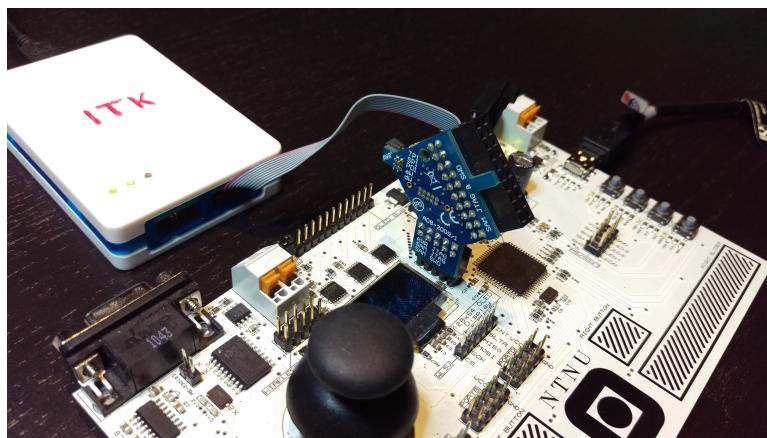
Ta vare på deres egen kode - ikke bare lagre på en lokal datamaskin og gå. Google Drive kan funke, men git kombinert med github anbefales. Mange av dere er allerede kjent med git fra heisprosjektet, men om dere trenger en rask innføring kan det være lurt å ta en titt på gitimmersion.com.

3 Oppsett av labutstyr

3.1 Hardware

P1000-kortet som brukes i denne laben (figur ??) kan trekke strøm både fra klemmene opp i høyre hjørne, i tillegg til USB-koblingen i høyre kant. I denne laben bruker vi USB.

Programmering av kortet forgår via JTAG. Tidligere år har brukt en annen debugger enn det vi gjør nå, så figuren i Windows-laboppgaven er litt feil. Ta utgangspunkt i figur ??.



Figur 2: Kobling av debugger til P1000-kortet.

3.2 Software

Dere som har valgt Linux velger fritt hvilket tekstredigeringsprogram dere ønsker å benytte. Resten av *toolchainen* har vi satt opp for dere, men hvis dette er første labdag kan det hende dere må laste ned *avrdude*, *avr-gcc* og *avr-libc*. Dette gjøres ved å kalle

```
sudo add-apt-repository ppa:ubuntuhandbook1/apps  
sudo apt-get update  
sudo apt-get install -y avrdude gcc-avr avr-libc
```

fra kommandolinjen.

Når dette er gjort lager dere en ny mappe der dere laster ned *Makefilen* fra it's learning til. Denne fungerer på samme måte som den gjorde i heisprosjektet - bare kall **make** fra kommandolinjen, så ordner *gnu-make* resten.



Figur 3: P1000-kort med Atmel AT90USB1287-prosessor.

4 Oppgave 1

Koble opp kortet som beskrevet i seksjon ???. Deretter kobler dere den medfølgende flatkabelen mellom *PORTB* og *LEDS/SW*, som indikert i figur ???. Det er viktig at den røde ledningen på flatkabelen vender i samme retning på de to tilkoblingspunktene (for eksempel til høyre hos både *PORTB* og *LEDS/SW*). Deretter skal dere teste P1000-kortet ved å laste opp filen *example.hex* som finnes på it's learning. Dette gjøres simpelthen ved å kalle **make test** fra mappen med *example.hex*.

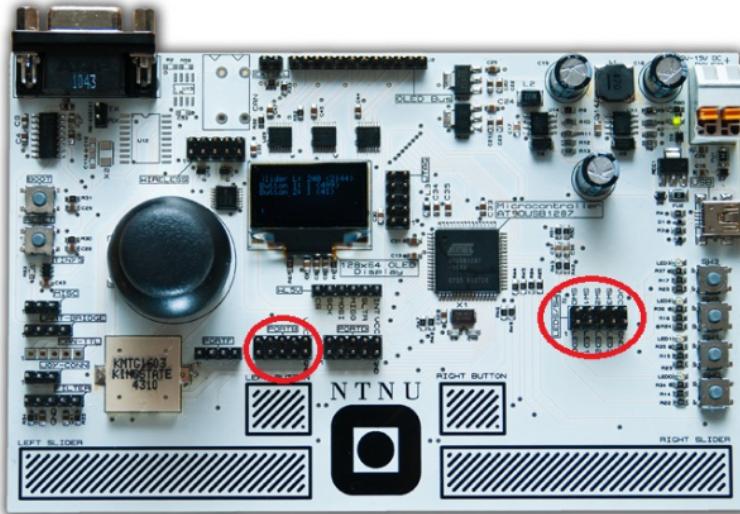
Når dette er gjort skal de fire LEDene (LED0-3) til høyre på kortet blinke etter hverandre. Hvis dette ikke skjer, så huk tak i en studass.

4.1 Knapper og LEDer

Dere skal bruke de fire knappene til høyre på kortet til å sette av og på tilhørende lys. Lag en fil kalt *main.c*. I toppen av denne inkluderer dere biblioteket *avr/io* ved å skrive **#include <avr/io.h>**.

Videre skal main-filen inneholde følgende funksjoner:

- void init_led()
- void init_switch()
- void set_led(int n, int v)
- int read_switch(int n)



Figur 4: Koblingspunkt av flatkabel.

Her skal n være et tall mellom 0 og 3, som bestemmer hvilken knapp/led vi snakker om. I set_led skal v være enten 0 eller 1, som bestemmer om det skal lyse eller ikke.

De to init-funksjonene skal sette nødvendige IO-registre før set_led og read_switch kallas. Alt dere trenger står i databladet under kapittel 11. Spesielt kapittel 11.1 og 11.2.1.

4.1.1 Hint

1. Både knappene og LEDene er koblet til PORTB på mikrokontrolleren. Knappene er koblet til pinne 1, 3, 5, 7. LEDene er koblet til pinne 0, 2, 4, 6.
2. LEDene er koblet i såkalt *sinking mode*. Det vil si at de lyser når 0 er skrevet til pinnene i PORTB. *Sourcing mode* er det motsatte.
3. Ta en titt på appendiks ?? for å se hvordan vi gjør bitvis operasjoner som å sette enkelte bit i C. Særlig i kodesnutt ?? er det tips å hente.

4.2 USART - Serieport

UART og USART, som står for **U**niversal (**S**ynchronous) **A**synchronous **R**eceiver **T**ransmitter, er to forkortelser som brukes litt om hverandre. Dette gjøres mest sannsynlig for å forvirre de uinntektede.

Uansett årsak skal vi bruke UART til å kommunisere med datamaskinen. Koble pinne 1 på UART BRIDGE til pinne 4 på PORTD, og pinne 2 på UART BRIDGE til pinne 3 på PORTD. Se figur ???. Etter dette oppretter

dere filene *usart.c* og *usart.h*. To funksjoner skal implementeres:

- int init_usart(int baudrate);
- int usart_putchar(char c);

Baudrate betyr hvor mange bit vi sender hvert sekund. Init-funksjonen skal bare sette opp nødvendige register. Funksjonen usart_putchar skal først sjekke om det er klart for å sende, for så å dytte bokstaven *c* over seriekabelen.

Bruk følgende oppsett:

- Baudrate 19200.
- Asynkron overføring.
- Ingen paritetsbit.
- Symbolstørrelse på 8 bit.

Alt dere trenger står i kapittel 19 i databladet, men hvis dere vil ha noe som kanskje er mer lettfattelig, så ta en titt på extremeelectronics.co.in.

Når dere er klare for å teste kobler dere P1000-kortet til datamaskinenes øverste DSUB9-port. Etterpå kaller dere **dmesg** fra terminalen. En av de nederste linjene burde fortelle hvor P1000-kortet ligger registrert - for eksempel under */dev/ttys0*. Når dere har funnet ut hvor kortet ligger, kaller dere **sudo stty -F /dev/ttys0 19200** (dersom det ble lagt til som ttys0), etterfulgt av **cat /dev/ttys0**. Hvis dere får bokstaver fra P1000-kortet har dere greid oppgaven.

4.2.1 Hint

1. Mikrokontrolleren kjører på 16 MHz, legg til **#define F_CPU 1600000UL** i toppen av *uart.c*.
2. Det er lurt å sende '\r' etterfulgt av '\n' for å avslutte linjer sendt fra mikrokontrolleren.

A Bitoperasjoner i C

C har følgende bitvise operatorer:

& - Bitvis **og**

| - Bitvis **eller**

- \wedge - Bitvis **XOR** (exclusive or)
- \sim - Bitvis **ikke** (not)
- $<<$ - Venstreskift
- $>>$ - Høyreskift

Et eksempel på hvordan disse operatorene virker er gitt i kodesnutt **??**. Du forstår best hvordan koden i snutt **??** fungerer hvis du skriver opp eksemplene selv og gjør de for hånd!

Kodesnutt 1: Bitvise operatorer i C.

```
// Prefix 0b means number in binary
char a = 0b10101010;
char b = 0b11110000;
char c;

c = a | b;           // c is now 0b11111010
c = a & b;           // c is now 0b10100000
c = b >> 2;          // c is now 0b00111100
c = a ^ b;           // c is now 0b01011010
c = ~b;              // c is now 0b00001111
```

Når vi programmerer mikrokontrollere setter vi ofte ett enkelt bit på eller av i et register. Dette gjøres ofte på følgende måte:

Kodesnutt 2: Setting av enkelte bit.

```
char c = 0b10000001;

// To set bit 3 from right side, we do:
c |= (1 << 2);           // c is now 0b10000101

// To clear it again, we do:
c &= ~(1 << 2);          // c is now 0b10000001
```

Igjen: Du lærer hvordan koden i snutt **??** fungerer ved å gå gjennom den for hånd.

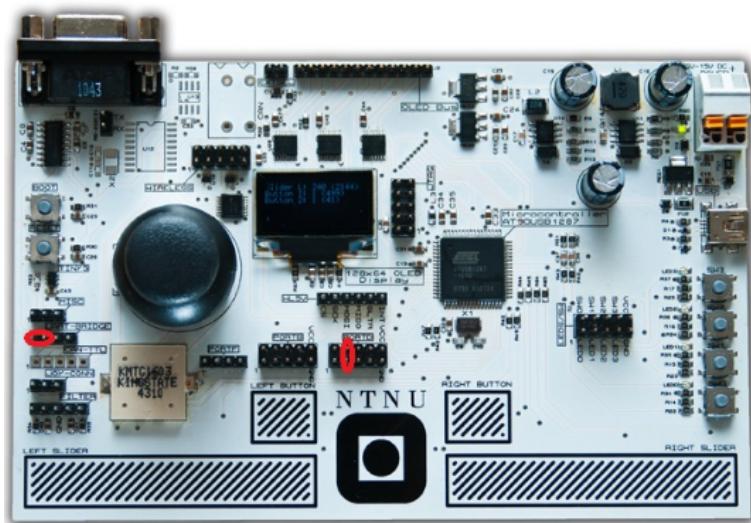
På mikrokontrolleren brukt i denne laben er registrerne 8 bit, og kan fint settes på måten vi gjorde i kodesnutt **??**. Dette er imidlertid bare tull, fordi koden blir helt uoversiktlig. For å gjøre jobben din enklere har hvert register og hvert bit i hvert register sitt eget navn. Disse navnene er definert i biblioteket *avr/io*. Slik bruker vi det:

Kodesnutt 3: Bruk av registernavn.

```
#include <avr/io.h>
...
// To set bit PB0 in DDRB
DDRB |= (1 << PB0);

// To clear bit PB0 in PORTB
PORTB &= ~(1 << PB0);
```

I kodesnutt ?? demonstrerer vi hvordan vi setter pinne 0 i **Data-Direction-Register-B**, slik at pinne 0 blir satt i *output*-modus. Deretter skrur vi av pinne 0 i PORTB, slik at pinnen er logisk lav.



Figur 5: Kobling UART BRIDGE og PORTD.