Journal 2 - Communication Busses

I3GFV - E18

Andreas Ellegaard Svendsen	201707772
Martin Gildberg Jespersen	201706221
Marcus Gasberg	201709164

[1]

[1] Modified picture from source: https://www.corkracecourse.ie/free-shuttle-bus/

Indholdsfortegnelse

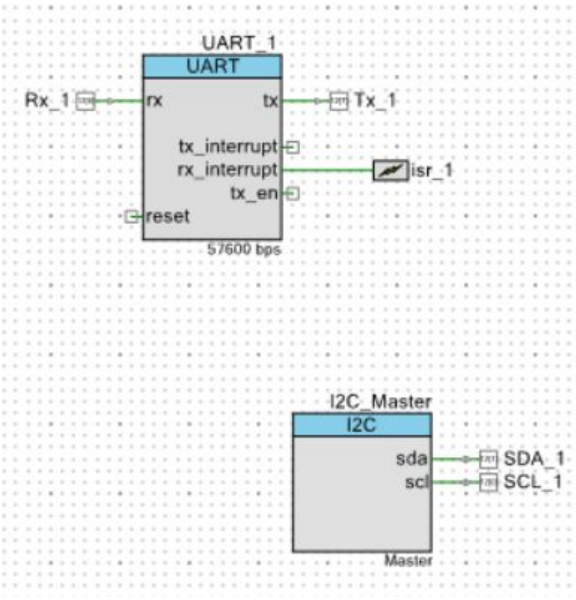
I2C experiment: PSoC master and LM75 slave.

Introduktion

Dette eksperiment omhandler I2C kommunikation. Mere nøjagtigt vil vi finde ud af, hvordan man læser data, fra en LM75, gennem I2C protokollen. Dataen skal konverteres til en temperatur og udskrives i en terminal.

Design

I eksperimentet skal der måles temperatur på en LM75 gennem I2C kommunikation. Til dette skal bruges selve LM75 IC'en som er udleveret på et print. Den skal forbindes til en PSoC, hvor den hhv. Skal bruge 5 V VCC, ground, SCL og SDA. SCL og SDA gør at der kan kommunikeres med PSoC'en. Mellem PSoC'en og computeren er en UART, så PSoC'en kan skrive temperaturen til en terminal.



Figur 1 - Design for I2C kommunikation mellem PSoC og LM75.

Implementering

For at kunne læse fra LM75 blev der implementeret en enkelt metode; getTemp(), der både henter temperaturen, konverterer den og skriver den til UART. getTemp() er implementeret, som ses i figur 2.

```
void getTemp(uint8_t slaveAddr)
{
    uint8_t tempMsb, tempLsb = 0; //Variable for storing temperature bytes
    char buf[256]; //Buffer for writing temperature to uart
    int status = I2C_Master_MasterSendStart(slaveAddr, 1); //Read from slave
    if(status == I2C_Master_MSTR_NO_ERROR )
    {
        //Get the bytes and print to uart using the temperature converter
        tempMsb = I2C_Master_MasterReadByte(I2C_Master_ACK_DATA);
        tempLsb = I2C_Master_MasterReadByte(I2C_Master_NAK_DATA);
        sprintf(buf,"The Temperature of slave %d: %.1f deg\r\n",slaveAddr,temperatureConverter(tempMsb,tempLsb));
        UART_1_PutString(buf);
    }
    else
    {
        //Print the error
        sprintf(buf,"Erro in sending start: errno %d\r\n",status);
        UART_1_PutString(buf);
    }
    I2C_Master_MasterSendStop(); //Stop
    CyDelay(1000); //Delay
}
```

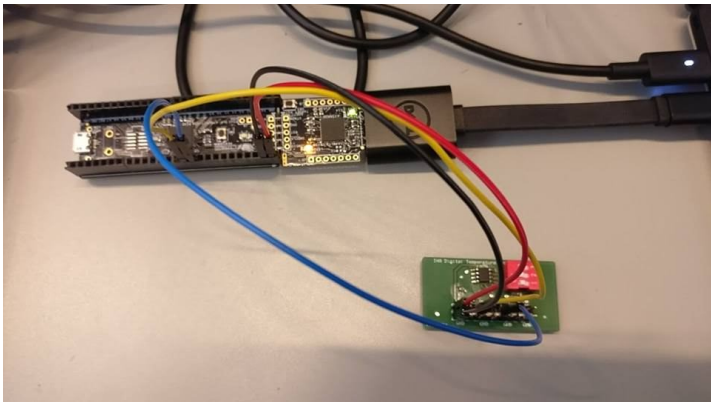
Figur 2 - getTemp() funktionens implementering

Det ses af figur 2, at der først sendes et start fra master til slaveAddr, hvor det specificeres, at der skal læses fra slaven. Hvis starten lykkedes så hentes, der to bytes fra slaven. Disse converteres med temperatureConverter til en float, der indeholder temperaturen i grader og det skrives til uart. Til sidst sendes et stop og der kaldes et delay.

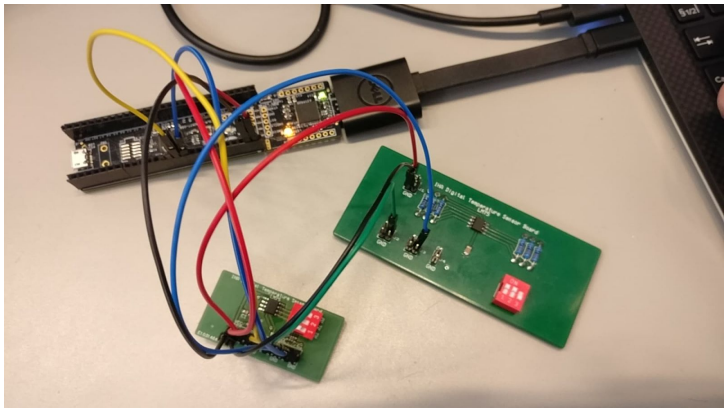
Eksperiment udførelse

Komponenterne der blev brugt til forsøget kan ses herunder.
Komponenter udover PSoC:

- LM75



Figur 3: opstilling for forsøget med en LM75 slave.



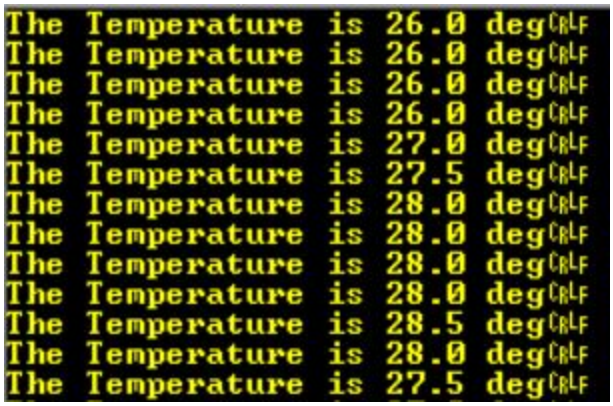
Figur 4: Opstilling for forsøget ved brug af 2 LM75 slaves.

På figur 3 og 4 ses opstillingen, LM75 printene er forbundet med PSoC'en, hvor VCC er de røde ledninger, Ground er sorte ledninger, SCL er blå ledninger og SDA er grøn og gul.

Selve eksperimentet blev udført ved at åbne terminalen og se at temperaturen blev skrevet hertil. Temperaturen ændres også ved at opvarme LM75 med friktions energi.

Resultater

Første del omhandler kun en I2C enhed, der skriver til terminalen. Dette kan ses i figur 5.



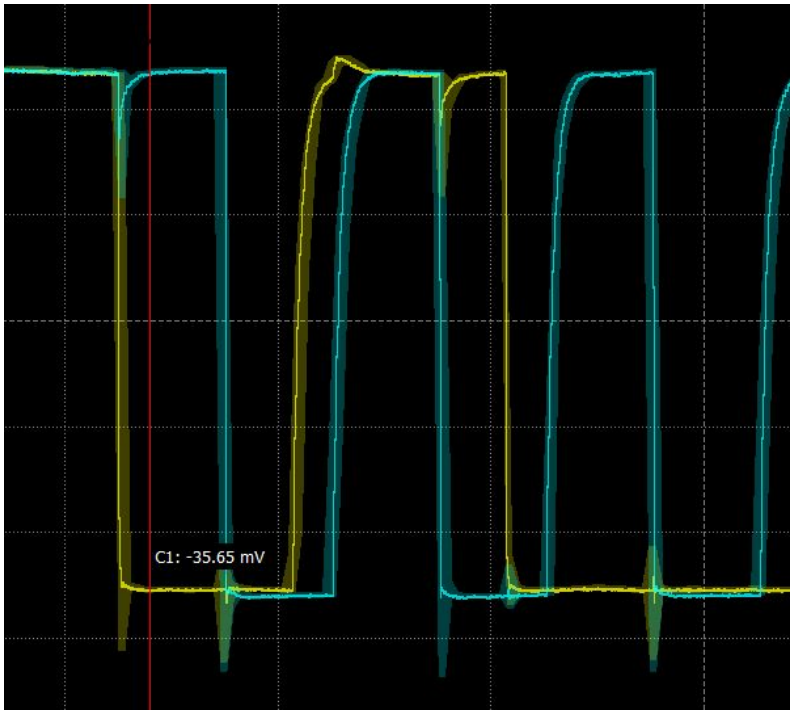
Figur 5 - Terminal vindue, hvor der læses fra en slave

I anden del skulle der tilføjes en slave mere. Dette kan ses i figur 6.

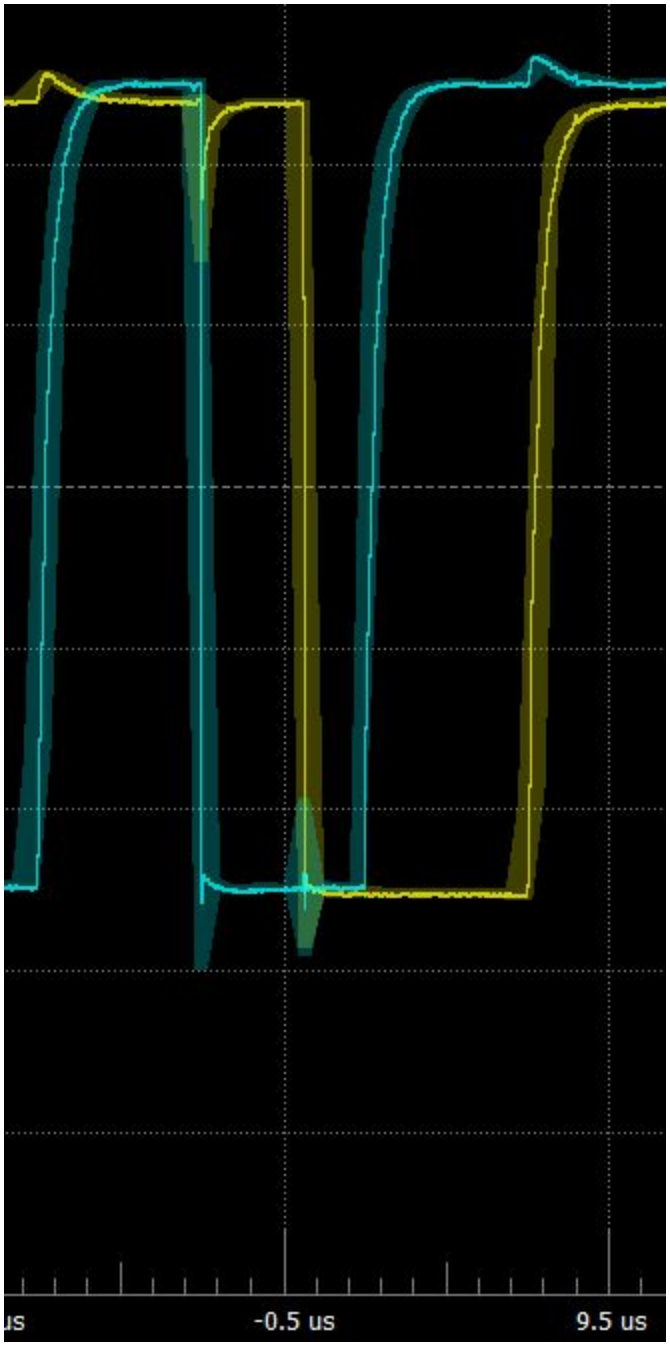
```
The Temperature of slave 72: 28.0 deg CRLF
The Temperature of slave 73: 30.5 deg CRLF
The Temperature of slave 72: 28.5 deg CRLF
The Temperature of slave 73: 30.5 deg CRLF
The Temperature of slave 72: 28.5 deg CRLF
The Temperature of slave 73: 30.0 deg CRLF
The Temperature of slave 72: 29.0 deg CRLF
The Temperature of slave 73: 30.0 deg CRLF
The Temperature of slave 72: 29.0 deg CRLF
The Temperature of slave 73: 29.0 deg CRLF
The Temperature of slave 72: 28.5 deg CRLF
The Temperature of slave 73: 29.0 deg CRLF
The Temperature of slave 72: 29.0 deg CRLF
The Temperature of slave 73: 28.0 deg CRLF
The Temperature of slave 72: 29.5 deg CRLF
```

Figur 6 - Terminal vindue, hvor der læses fra slave 72 og 73

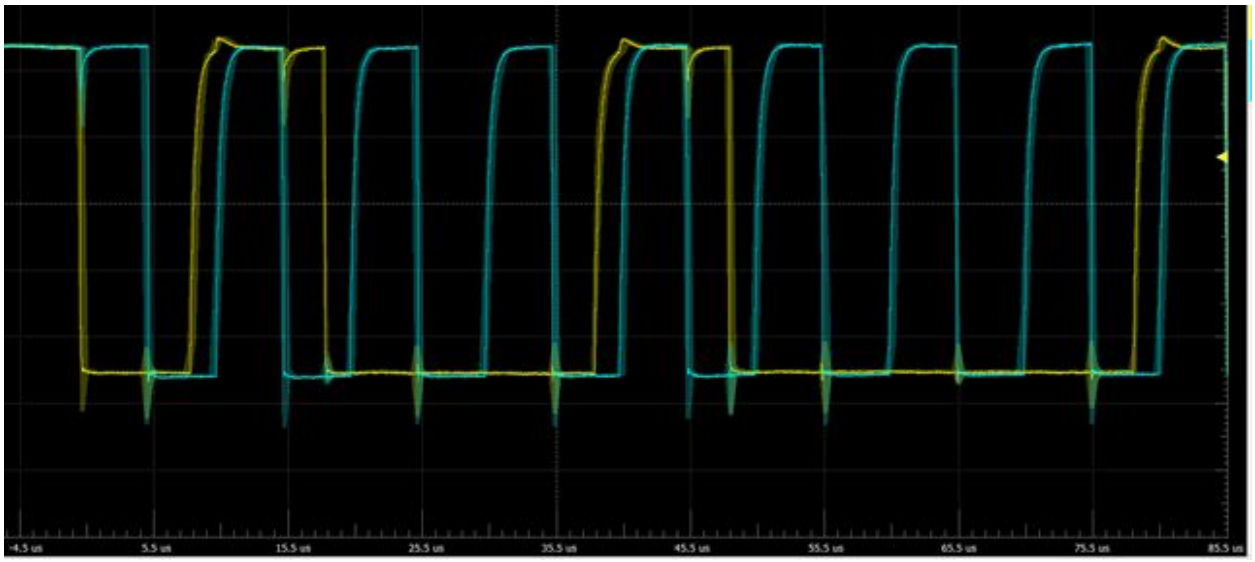
Herudover blev der også lavet en oscilloskop målinger af signalet beskerne, :



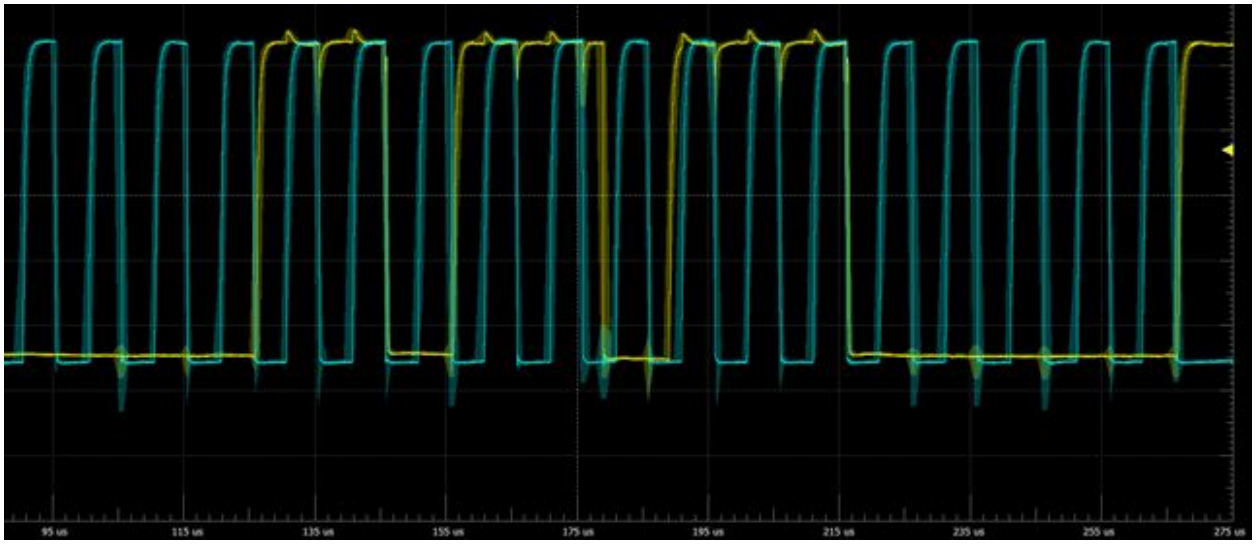
Figur 7 - Oscilloskop målinger af start bit og begyndelsen af adresse byten i I2C signalet. Man ser her at SDA signalet går lavt mens clocken er høj, hvilket signalerer en start på en I2c besked.



Figur 8 - Oscilloskop målinger af stop bit hvor man i modsætning til start bit ser SDA signalet gå højt, når clocken er høj, hvilket signalerer en slutningen på en I2c besked.



Figur 9: Her ses adressebyten, hvor det er muligt at se, hvilken LM75 komponent PSoC kommunikerer med. I dette eksempel kommunikerer PSoC med LM75, hvor alle de justerbare bit er sat til 0er. Det gule spike i slutningen af addresse byten fortæller om det Masteren nu skal læse fra slave-enheden LM75.



Figur 10: Her ses de to bytes, der sendes til PSoC efter at adresse byte er sendt og har fået besked. Man kan afkode dette og se at temperaturen er 27.5 grader celsius. Man kan se ACK bit, som det gule signal, der går lavt og ser lidt anderledes ud end de andre. Dette sker efter det niende clock signal går lavt. NACK bit ses, som det gule spike i slutningen af figuren efter det sidste clock signal går lavt på figuren.

Diskussion

- **Where are the start and stop conditions for a single communication? • How do you see which LM75 the PSoC communicates with?**

I main kaldes funktionen getTemp(), getTemp() tager en variable som er en int der indeholder slave addresen. Addresen fås tildels fra datablad dog er 3 bits “valgfrie” da de kunne skiftes på LM75 printet.

- **How do you see, if a data transfer is a read or write?**
  Dette kan ses i implementeringen af getTemp(), da master kalder start tager den 2 variabler, den ene er addresen på LM75 slaven, og den anden er et bit. Dette bit er Read/Write bittet, i tilfældet set på figur 2 er den sat til 1 da det er read.
- **Where is the data, which is being transferred?**
  De sidste to bytes der bliver sendt er temperaturen. Da temperaturen kan findes ud fra kun 9 bit består den første byte af de 8 første bits i temperaturen og den MSB i sidste byte er så den sidste bit.
- **In which direction does the data flow?**
  Adresse byte sendes fra Master(PSoC) til Slave(LM75), hvor den sidste bit er read/write, hvilket fortæller slave at den skal sende de næste bytes til Master. Herefter sender slave data den første data byte efterfulgt af ACK fra slave som fortæller at den har modtaget data. Herefter sender Slave nu den sidste byte efterfulgt af NACK, som fortæller Master at slave nu ikke har mere at sende. Master sender herefter et stop bit.

- **Where are the ACK/NACK parts of the the communication?**
  ACK er den bit der adskiller de to data bytes og adresse byte og første data byte, mens NACK er den sidste bit efter den sidste byte er sendt. Det kan ses i på oscilloscop billederne i resultat afsnittet.

Konklusion

Eksperimentet var en succes, da det lykkedes at læse temperaturen fra en eller flere LM75 slaves, og udskrive temperaturen i grader Celsius i terminalen. Vi fik også brugt oscilloscopet til at tage billeder af resultatet, hvor vi kunne afkode I2c beskeden på oscilloscopet og få det til at stemme overens med det vi fik skrevet ud på terminalen.

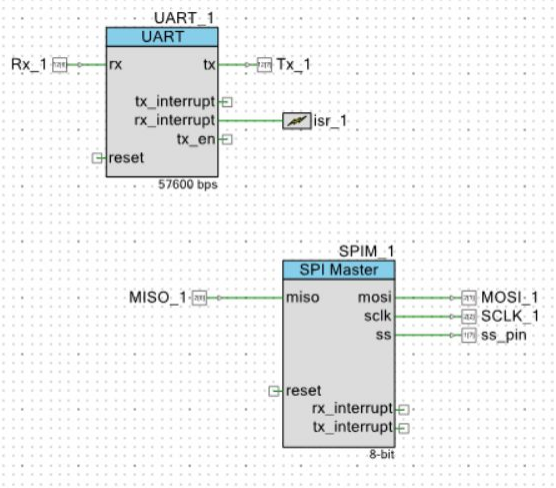
SPI experiment: PSoC master and PSoC slave.

Introduktion

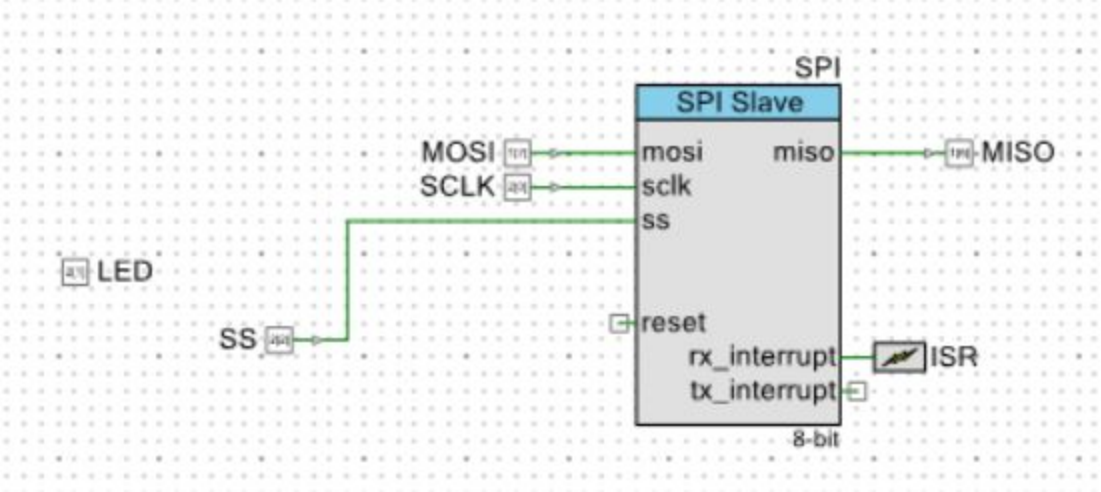
I denne opgave skal to PSoC forbindes gennem SPI, så den ene er master og den anden er slave. Master PSoC'en skal kunne tænde og slukke LED'en på slave PSoC'en, og den skal kunne læse om LED'en er tændt eller slukket.

Design og Implementering

Det overordnede design for SPI-kommunikationen kan ses i figur 11og figur 12.



Figur 11 - Design for SPI- master



Figur 12: Det overordnede design for slave med SPI og LED.

Den sværeste implementering i SPI-masteren, var at læse Led'ens status. Vi starter ud med at sende et request om at den skal sende sin status. For så at læse den status, bliver master nødt til at sende en masse trash, for at skubbe status tilbage til sig selv. Til sidst læses dette data. Dette kan ses i figur 13.

```
uint8_t getLedStatus()
{
    SPIM_1_WriteTxData(requestStatus); //Request status of led

    for (uint32_t i = 0; i < 8; i++){
        SPIM_1_WriteTxData(0b11111111);
    } //Write 8 times to push the value back to the master

    return SPIM_1_ReadRxData(); //return data read
}
```

Figur 13: Her ses SPI-masters getLedStatus()-funktion.

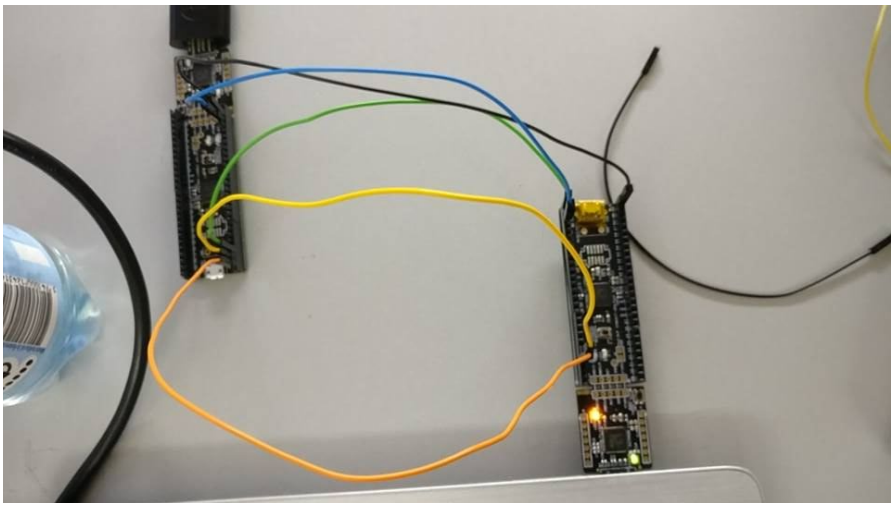
Koden for slaven ses nedefor. Man kan se at vi først læser buffer size, så gør vi ind i en lykke, s alt i bufferen bliver læst. Inden i lykken tænder slukker vi LED'en eller sender status tilbage til master alt afhængig af hvad masteren sendte til slaven. Dette ses i figur 14.

```
15  CY_ISR(ISR_handler)
16  {
17      uint8_t bytesToRead=SPI_GetRxBufferSize();
18      while(bytesToRead!=0)
19      {
20          uint8_t ByteReceived=SPI_ReadRxData();
21          if(ByteReceived==0b00000001) //LED turn on message from master
22              LED_Write(1u);//turn on LED
23          else if(ByteReceived==0b00000000)//LED turn off message from master
24              LED_Write(0u);  //turn off LED
25          else if(ByteReceived==0b00000011)//LED requist status message from master
26              SPI_WriteTxData(LED_Read());//write to master with LED status
27  --bytesToRead;
28    }
29
30  }
```

Figur 14: Her ses SPI slavens interrrupt vektor.

Eksperiment udførelse

Til eksperimentet bruges 2 stk PSoC.

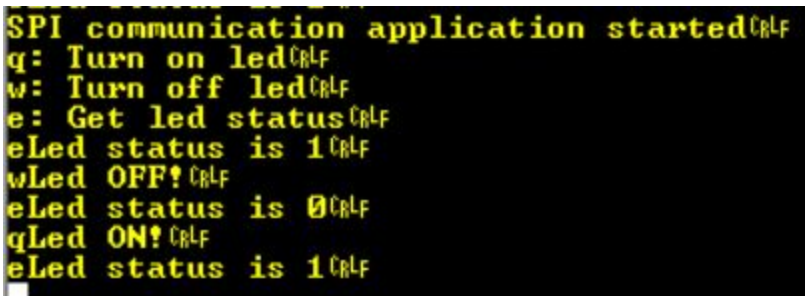


Figur 15: De to PSoC er forbundet med commonground(sort), MOSI(gul), MISO(Orange), SCLK(grøn) og SS(blå).

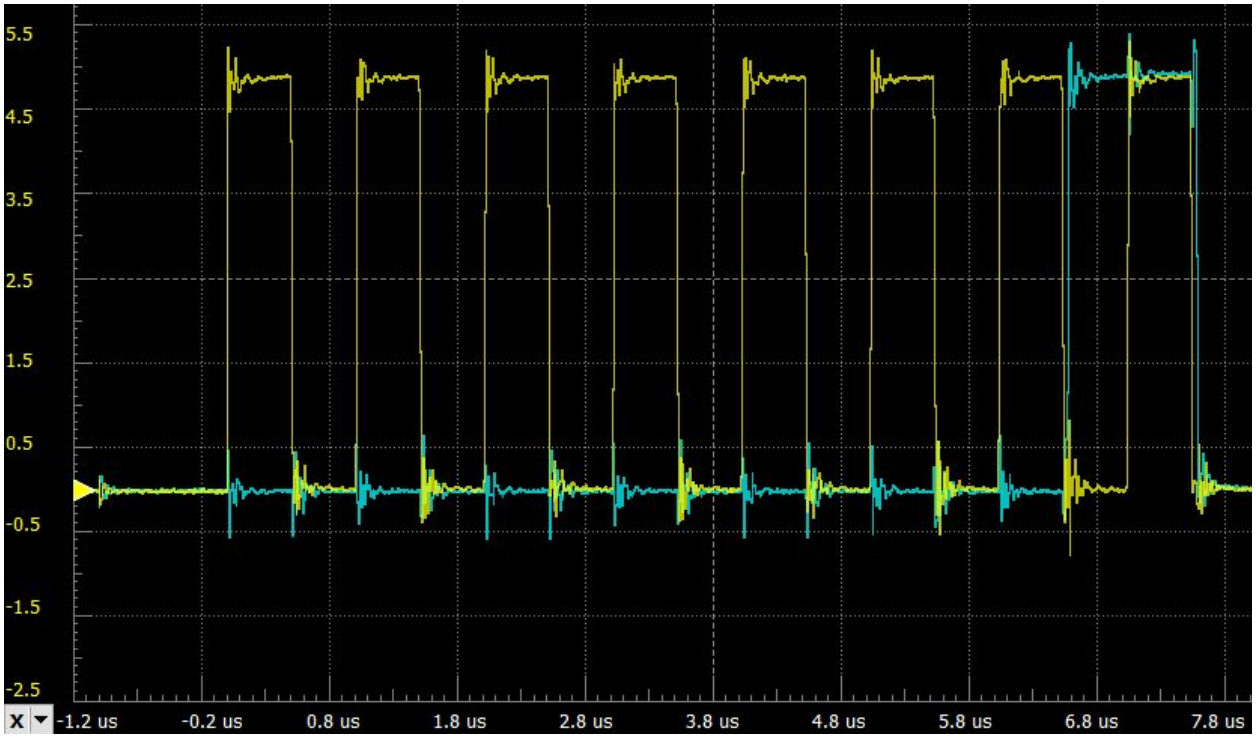
Eksperimentet blev udført ved at lade slavens led være tændt, hvorefter der prøves at læse fra slaven. Herefter slukkes lyset, og der læses igen. Herefter tændes lyset, og der læses igen.

Resultater

Test af SPI-kommunikation kan ses i figur 16, der viser et terminal udsnit. Ud over dette kan, der i den medlagte video også ses en live test.



Figur 16: Test af spi kommunikation

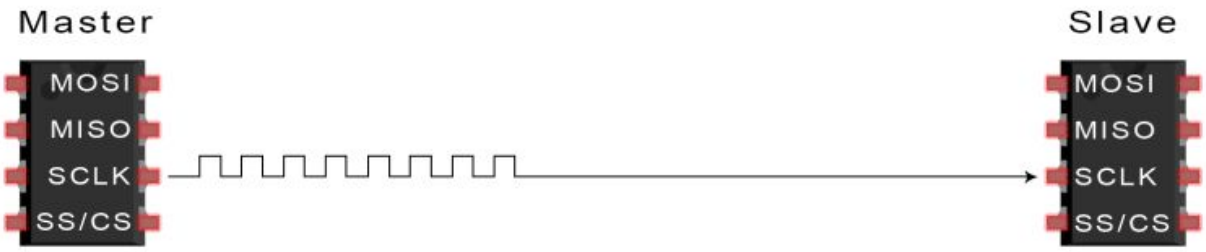


Figur 17: Her ser vi oscilloscop billedet for SPI kommunikationen.

Diskussionen

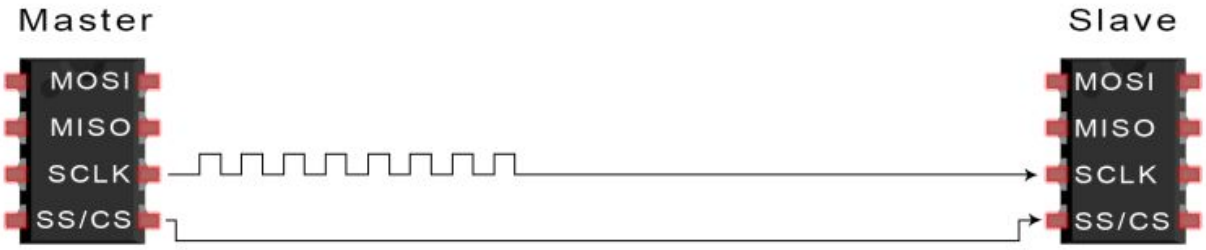
Det er vigtigt for vores kommunikation, at den rigtigt værdi læses ved opstart af programmet. Derfor kan der i figur 16 ses, at der laves en test, hvor led'en er tændt, lige når programmet går i gang. Der ses også at det virker.

- **How does the SPI communication work?**
  SPI fungerer via 4 ledninger, som er MISO, MOSI, SCLK og SS. Ligesom i I2c er det en master og en til mange slaves. Det første der gøres er at aktiverer den slave der skal kommunikeres med, hvilket sker på SS ledningen. Så skerdet at Master-enheden(kunne være PSoC) sender et clocksignal ud med de antal pulser svarende til længden af beskeden. Kan ses på figuren nedenfor.



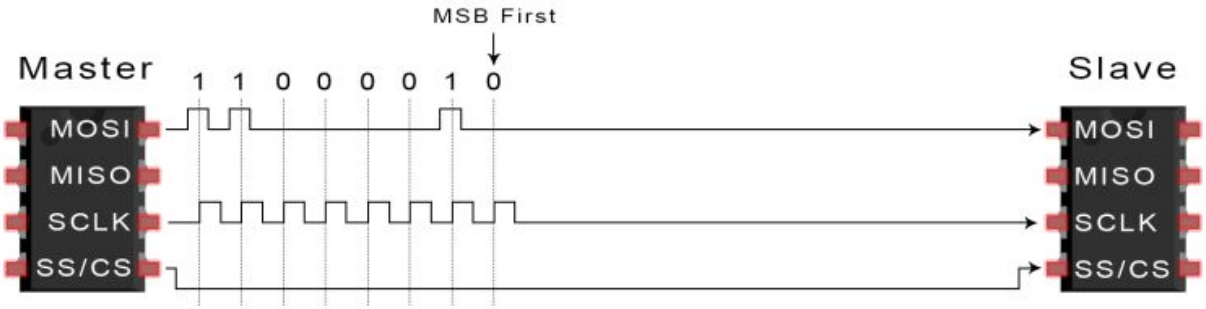
Figur 18: Clock signal sendes til slave.[2]

[2] http://www.circuitbasics.com/basics-of-the-spi-communication-protocol/



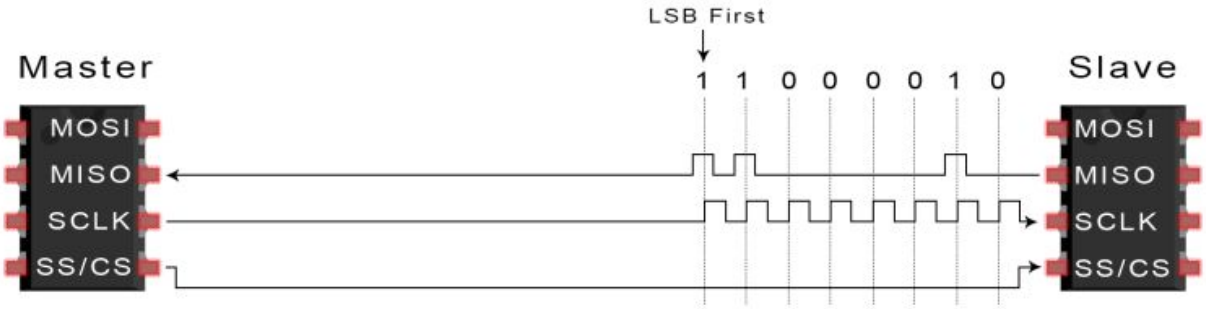
Figur 19: Master sender signal ud til Slave med et clock signal.[3]

Beskeden sendes fra Master til slave via MOSI lednignen, som lægges oveni det udsendte clocksignal(SCLK). Ses nedenfor.



Figur 20: Fra MOSI ledningen sendes besked til slave oveni det clocksignallet.[4]

Hvis en slave skal sende noget tilbage foregår dette på en måde så registret der skal sendes tilbage til Master bliver bit shiftet ud af registret videre ud på MISO ledningen tilbage til Master. Hver bit shift foregår efter et bit modtages fra Master enheden. Slave eheden sender signalet videre med den frekvens den modtager på clock ledningen SCLK.



Figur 21: Bits sendes tilbage tilbage til Master via MISO ledningen.[5]

[3] http://www.circuitbasics.com/basics-of-the-spi-communication-protocol/
[4] http://www.circuitbasics.com/basics-of-the-spi-communication-protocol/
[5] http://www.circuitbasics.com/basics-of-the-spi-communication-protocol/

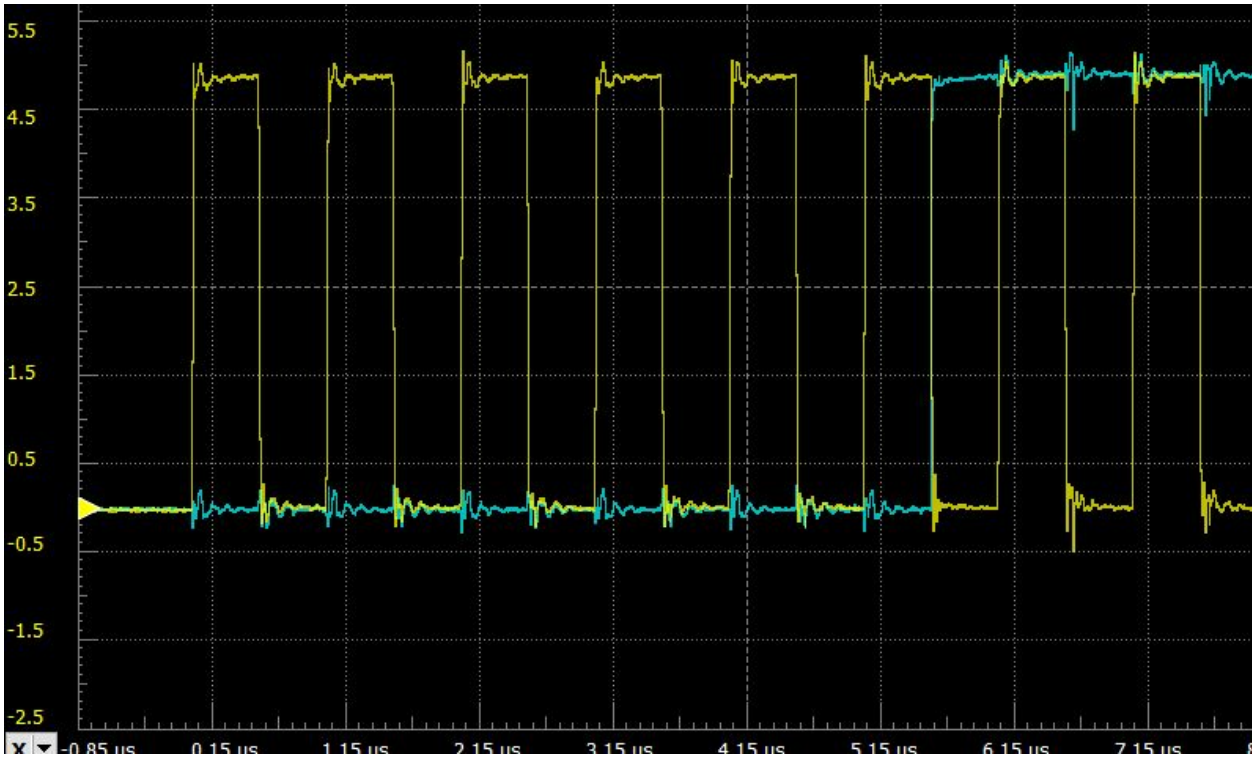
- **Which SPI mode are you using? How can you see that on the oscilloscope?**
  På oscilloscop billedet figur 22 ser vi at clocken idles ved 0, så clock polaritetten er 0(CPOL=0). Fasen er CPHA=0, da leading egde ses i midten af det høje bit.
- **Explain the details of your protocol, using the oscilloscope plot(s).**
  Den anvendte protokol kan ses i figur 21, samt test af funktionerne turnOn(), turnOff() og requestStatus() i henholdsvis figur 22, 23 og 24.

```
volatile const uint8_t turnON = 0b00000001;
volatile const uint8_t turnOFF = 0b00000000;
volatile const uint8_t requestStatus = 0b00000011;
```

Figur 21 - Protokollen som anvendes set i kode

Figur 22: Oscilloskopbillede af turnOn

Figur 23: Oscilloskopbillede af turnOff



Figur 24: Oscilloskop billede af requestStatus. (blå gr i lavt til aller sidst hvilket skyldes vores implementation af masteren, hvor sender gabbage ud i starten).

Konklusion

I sidste ende kom alle af parametrene til at virke som forventet. Man kunne fra slave slukke og tænde LED'en på slaven og få status. Koden er dog ikke optimal, da vi er nødt til at sende noget tilfældigt ud 8 gange inden det rigtige sendtes. Efter dette virker det dog, som det skal.

Referencer

http://www.circuitbasics.com/basics-of-the-spi-communication-protocol/