

GFV - Journal 2

Communication buses

Øvelseshold 69:

Thomas Kaae - 202100350

&

Jakob Damgård Dall - 201805013

&

Mathias Martin Rahbek Markussen - 202107385

&

Shukriya Bile - 202107346

Softwareteknologi 2. Semester

Forår 2022



AU technical sciences

Aarhus University

Denmark

10. marts 2022

Indhold

1	Introduktion	2
2	I2C kommunikation	3
2.1	Formål	3
2.2	Design	3
2.2.1	Indledende overvejelser	3
2.2.2	Softwaredesign	3
2.2.3	Hardwaredesign	6
2.3	Resultater	7
2.4	Diskussion	7
2.5	Konklusion	8
3	SPI	9
3.1	Formål	9
3.2	Design	9
3.2.1	Indledende overvejelser	9
3.2.2	Softwaredesign	9
3.2.3	Hardwaredesign	10
3.3	Resultater	12
3.4	Diskussion	14
3.5	Konklusion	14

1 Introduktion

Udarbejdelsen af denne journal og gennemførelsen af tilhørende laboratorieøvelser, har til sinde at skabe en grundlæggende forståelse for hyppigt anvendte datakommunikationsbusser.

Til laboratorieøvelserne gøres der brug af I2C og SPI, der er to forskellige datakommunikationsbusser. Dette er for at skabe en indsigt i forskellige former for kommunikation mellem enheder.

Øvelsen med I2C, gør brug af to LM75 temperatur sensorer som slave, og en PSoC som Master

Øvelsen med SPI gør brug to PSoC'er, hvoraf den ene anvendes som "*Master*" og den anden som "*Slave*".

2 I2C kommunikation

2.1 Formål

I det følgende undersøges I2C kommunikation. En PSoC samt 2 LM75 sensorer forbindes til en I2C bus. Her agerer PSoC'en som master, og LM75'erne som slaver.

Først præsenteres opstillingen samt programmeringen af PSoC'en.

Dernæst testes kommunikationen, både funktionelt, og ved Oscilloskopmålinger.

Sluteligt diskutes og konkluderes på resultaterne.

2.2 Design

2.2.1 Indledende overvejelser

I2C kommunikation består af en databus og en clock forbindelse. Enheder skal forbindes til begge for at kunne kommunikere. Begge forbindelser er gennem en pull-up modstand forbundet til Vcc (her 5V), og er derfor HIGH som standard. Enhver enhed kan hve begge forbindelser LOW.

Kommunikation foregår ved aflæsning af databussen. Bussen aflæses kun når clock'en er HIGH, og kan kun ændre status når clock'en er LOW. Undtagelse herfor er start/stop kommandoer, der sendes af en master når clock'en er HIGH.

2.2.2 Softwaredesign

PSoC'en programmeres til at indeholde en UART samt en I2C master. I2C masterens data- samt clock-forbindelse, tilknyttes en fysisk pin på PSoC'en. UART'en sættes til at kunne kommunikere gennem USB'en.

Målet med softwaren er simpelt: At kontinuerligt kunne aflæse en temperatur fra en af de tilknyttede LM75 enheder. Det vælges at aflæse temperaturen fra hver enhed skiftevis. Koden beskrives herunder kort:

Koden er delt op i 'functions' header og c-filer, samt en main. I main initieres UART og I2C kommunikation, hvorefter temperaturmålerne skiftevis kaldes i et loop. Der er tilføjet et delay på 500ms mellem hver måling. Databladet specificerede minimum 300ms, for at en ny måling var lagret i sensorens buffer. Koden for main filen er at finde i Code Section 2.1.

Funktionen, hvis ansvar er at aflæse og printe temperatur fra en slave fungerer essentielt som følger: Startkommando sendes, og der ventes på ACK. Første byte modtages - der svares med ACK. Andet byte modtages - der svares med NACK. Stopkommando sendes. De to bytes converteres fra "2's complement" til en float, og udskrives i UART. Se koden for c- og header-filerne "functions" i Code-Section 2.2 og 2.3.

Til at kunne aflæse beskederne sendt af PSoc'ens UART anvendes softwaren RealTerm.

2.1: I2C main.c

```
1 #include "project.h"
2 #include "functions.h"
3
4 int main(void)
5 {
```

```

6     CyGlobalIntEnable; /* Enable global interrupts. */
7
8     //Slave addresser:
9     uint8 slave1 = 0b1001000;
10    uint8 slave2 = 0b1001111;
11
12    UART_1_Start();
13    I2C_1_Start(); /*I2C komponentet initialiseres. I2C interrupts enables */
14
15    UART_1_PutString("Application started\r\n");
16
17    for(;;)
18    {
19
20        readAndPrint(slave1);
21        CyDelay(500);
22        readAndPrint(slave2);
23        CyDelay(500);
24    }
25 }
```

2.2: I2C functions.h

```

1 #include "project.h"
2
3
4 float convertMeasurement(uint8 byte1, uint8 byte2);
5 void writeTempToUART(float temp);
6 void readAndPrint(uint8 adress);
```

2.3: I2C functions.c

```

1 #include "functions.h"
2
3 void readAndPrint(uint8 adress)
4 {
5     uint8 errorMSG, byte1, byte2;
6     float temperature;
7     /*Send start til slave 1. Gem error MSG */
8     errorMSG = I2C_1_MasterSendStart(adress,1);
9     if(errorMSG != I2C_1_MSTR_NO_ERROR)
10         UART_1_PutString("START NOT ACKNOWLEDGED");
```

```

11
12 //Læs første byte. MSB kommer først. Send ACK efter
13 byte1 = I2C_1_MasterReadByte(I2C_1_ACK_DATA);
14
15 //Læs andet byte. Kun LSB her. Send NACK, så den ikke sender mere.
16 byte2 = I2C_1_MasterReadByte(I2C_1_NAK_DATA);
17
18 //Bed den om at stoppe
19 I2C_1_MasterSendStop();
20
21 //Konverter fra 2's complement til float
22 temperature = convertMeasurement(byte1, byte2);
23 //Udskriv float i UART
24 writeTempToUART(temperature);
25 }
26
27 void writeTempToUART(float temp)
28 {
29     char outputbuffer[256];
30
31     sprintf(outputbuffer, sizeof(outputbuffer), "Temperature: %f \r\n", temp);
32     UART_1_PutString(outputbuffer);
33 }
34
35
36
37 float convertMeasurement(uint8 byte1, uint8 byte2)
38 {
39     //temporary variable til bitshifting
40     uint8 temp1, temp2;
41     //Det ene most significant bit bliver gemt her til sidst. (BIT 8)
42     uint8 msb;
43     //De 8 LSBs bliver gemt her. Bit 0:7
44     uint8 lsb;
45     float temperature;
46
47     msb = byte1>>7; // Fortegn ligger her. MSB
48     temp1 = byte2>>7;
49     temp2 = byte1<<1;
50     lsb = temp1 + temp2; /* Temperatur ligger her */
51
52     if(msb == 0)
53     {
54         temperature = (float)lsb/2;
55     }
56     else if(msb == 1)
57     {

```

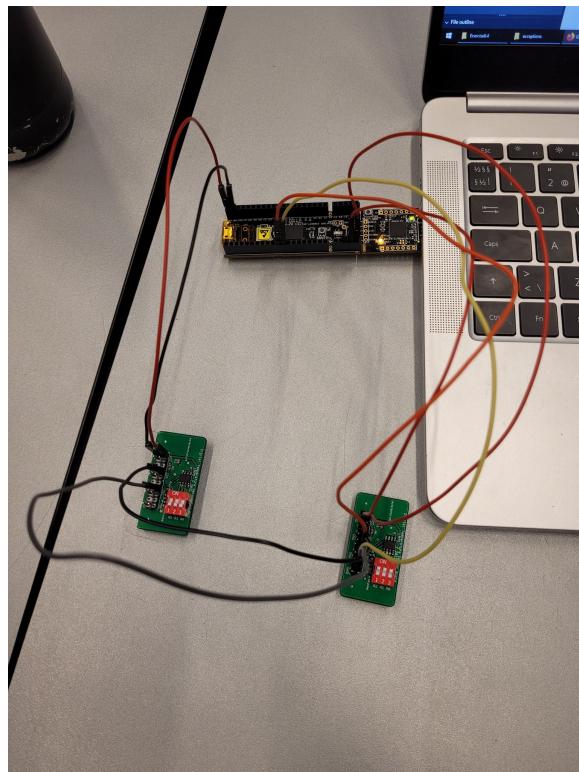
```

58         lsb = ~lsb + 0b00000001;
59         temperature = -((float)lsb)/2;
60     }
61     else
62     {
63         UART_1_PutString("SOMETHING WENT WRONG IN MEASUREMENT CONVERSION");
64     }
65
66     return temperature;
67 }
```

2.2.3 Hardwaredesign

Til øvelsen er færdiglavede print med en LM75 forbundet gjort tilgængelige. Her er allerede klargjort pins til både Vcc, gnd, data, samt clock. Derudover er en tredobbelt switch forbundet til chippens pin 5-7, hvilket tillader programmering af enhedens adresse på bussen.

Printet forsimpler hardwareopsætningen markant. To print forbinderes til 5V Vcc samt PSoC'ens data og clock linjer. Derudover forbindes alle tre enheder til fælles stel. Opstillingen er vist på figur 1



Figur 1: Opstilling af I2C master og 2 slaver. Ledningsfarver er uden betydning

2.3 Resultater

Programmet fungerede som ønsket. Temperaturen blev målt og printet på UART, og målingerne gav mening - De steg hvis den blev rørt, og faldt hvis den blev bragt udenfor. Se screenshot på figur 2.

Kommunikationen imellem slaver og master blev yderligere dokumenteret. På figur 3 ses masterens kommunikationssekvens med slave1, hvor temperaturen aflæses. Aflæste bits er markeret. Figuren diskuteres nærmere i diskussionsafsnittet.

```
Temperature: 26.000000 CRLF
Slave1 aflaeses CRLF
Temperature: 26.500000 CRLF
Slave2 aflaeses CRLF
Temperature: 26.000000 CRLF
Slave1 aflaeses CRLF
Temperature: 26.500000 CRLF
Slave2 aflaeses CRLF
Temperature: 26.000000 CRLF
Slave1 aflaeses CRLF
Temperature: 26.500000 CRLF
Slave2 aflaeses CRLF
Temperature: 26.000000 CRLF
Slave1 aflaeses CRLF
Temperature: 26.500000 CRLF
Slave2 aflaeses CRLF
Temperature: 26.000000 CRLF
Slave1 aflaeses CRLF
Temperature: 26.500000 CRLF
```

Figur 2: Screenshot fra realterm med temperaturmålinger

2.4 Diskussion

Oscilloskopmålingen på figur 3 viser meget tydeligt, hvordan kommunikationen foregår. Den gule er datalinjen, den blå er clock. Læses der fra venstre, ses først start signalet, der sendes af master. Her ændres datalinjen - fra høj til lav - uden clock linjen er lav. Som beskrevet er start/stop kommandoer de eneste, der kan sendes uden ændring i clock.

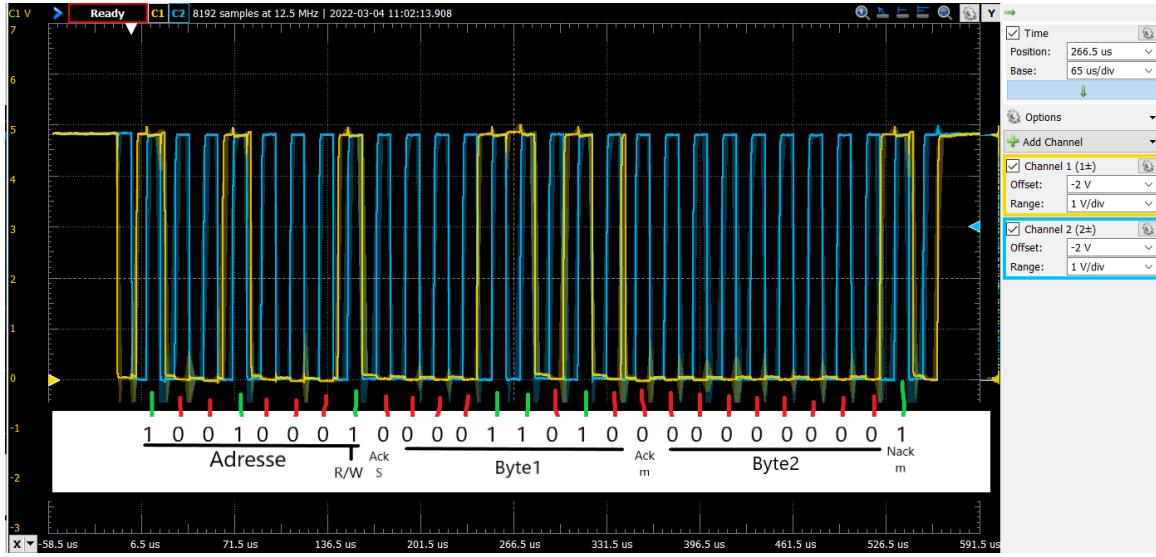
Efter startsignalet sendes master adressen på slave1 ud på datalinjen, efterfulgt af et r/w bit. Adressen er 7 bits, så der sendes i alt et byte. Slave1 svarer ved at hive datalinjen lavt, og sender ACK tilbage.

Dernæst sendes første byte af temperaturlæsningen. Slaven driver datalinjen højt/lavt når clocken er lav, og master mäter mens clocken er høj. Efter første byte, sender master ACK byte. Her kan det tydeligt ses, at slaven slipper datalinjen, og masteren med det samme hiver den lavt igen.

Til sidst sendes sidste databyte, efterfulgt af NACK fra master. Slaven sender ikke mere, og master sender en stopkommando.

Der kan ydermere kommenteres på tiden, hvort bit tager for at blive formidlet. Dataraten valgt i PSoC-Creator, er sat til at være 50kbps, svarende til, at én clock cycle tager 20 µs. Det passer med, at der mellem 6.5 µs og 71.5 µs mærkerne på figur 3, er lidt mere end 3 clock cycles. Ingen ser signalet ud som ventet.

Slutteligt skal det nævnes, at softwaredesignet programmet på PSoC'en er essentielt for at få så



Figur 3: Osciloskop måling. Channel1 er data, channel2 er clock

fint et signal, som på figur 3. Kommunikationen mellem master og slave skal ikke afbrydes af anden kode. I2C kommunikationen afhænger af clocken, og bruges der processortid på at køre andre kommandoer, kan signaler blive forlængede, og i værste fald ugyldige.

Ovenstående blev lært ved erfaring, da signalet pludselig så forkert ud. Det viste sig at være en UART besked, der var kommet ind mellem de to bitlæsninger.

2.5 Konklusion

I2C kommunikation mellem en master og to slaver blev succesfuldt opbygget. En PSoC agerede som master, og var i stand til at aflæse temperaturen gemt i en temperatursensors buffer. Målingerne blev konverteret til decimaltal og printet ud via en UART til en terminal.

Kommunikationen på de to linjer mellem slave og master, blev analyseret. Det blev eftervist, at clock- og datalinjerne agerer som ønsket - både ift timing og sendt information.

3 SPI

3.1 Formål

I følgende undersøges SPI kommunikation.

Det ønskes at lave en protokol til kommunikation mellem to PSoC'er som gør brug af SPI datakomunikationsbussen.

Under øvelsen anvendes den ene PSoC som SPI *Master* og den anden som SPI *Slave*.

Til øvelsen skrives der et program, der kan tænde/slukke for LED'en på SPI Slaven ved brug af en UART på Masteren. I forbindelse med øvelsen undersøges, hvordan SPI kommunikation samt, hvordan den konstruerede protokol fungerer, gennem brug af oscilloskop.

Slutteligt vil resultaterne blive diskuteret og konkluderet.

3.2 Design

3.2.1 Indledende overvejelser

Det er muligt at have flere *Slaver* tilknyttet en *Master*, men ikke omvendt, hvor SPI *Master*'en, så vil bestemme, hvilken *Slave*, der kommunikeres med. SPI er generelt opsat til "Full-duplex" kommunikation, hvorved der ikke kan sendes uden at modtage. Dette kan i øvelsen løses ved at sende en konstant datastrøm.

3.2.2 Softwaredesign

Igennem hele øvelsen med SPI kommunikation, gøres der brug af UART og en SPI Master og Slave. UART'ens formål er til dels at bekraefte forbindelsen mellem SPI Slaven og Masteren, men også, ved brugerens input, at kunne slukke og tænde for en LED på Slaven. Initieringen af UART sker på samme måde som i øvelse 1, og SPI i samme stil. (Se bilag)

Som tidligere nævnt, er SPI full duplex, hvilket giver anledning til udvikling af en protokol til data kommunikationen. Den valgte protokol for øvelsen ses i nedenstående kode.

3.1: SPI Protokol

```
1  uint8_t emptyData = 0x00; // Send ingenting
2  uint8_t onData = 0x45; // Nu skal LED tænde
3  uint8_t offData = 0x12; // Nu skal LED slukke
4  uint8_t buttonData = 0x02; // Nu er der registreret et tryk på knap
```

For at tænde og slukke LED'en på Slaven, skal den modtage og sende data til masteren konstant. I øvelsen, hvor Masteren skal tænde en LED på Slaven, er det fra Masteren ikke nødvendigt at sende data til Slaven kontinuerligt. Når knappens tilstand skal læses fra Masteren, er det dog nødvendigt. I nedenstående eksempel sendes `emptyData`, da begge parter er enige om dens ubetydning.

3.2: Tænd og sluk af LED (Slaven)

```
1  for(;;)
2  {
3      ByteRecieved = SPIS_1_ReadRxData(); // Læs modtaget data
4      SPIS_1_WriteTxData(emptyData); // Skriv ingenting tilbage (Nødvendigt)
5
6      if (ByteRecieved == onData) // Tænd LED
7      {
8          Pin_1_LED_Write(1);
9      }
10
11     else if (ByteRecieved == offData) // Sluk LED
12     {
13         Pin_1_LED_Write(0);
14     }
15 }
```

Som set i kodeeksempel 3.2, gør slaven brug af SPI protokollen der er blevet lavet.

Når slaven skal sende tilstanden på dens trykknap, gøres det i samme stil som når Masteren sender data afsted, hvor protokollen igen tages i brug.

3.3: Trykknap fra Slave til Master

```
1  for(;;)
2  {
3      if (Pin_2_Button_Read() == 0) // Knappen er blevet trykket på
4      {
5          SPIS_1_ReadRxData(); // Vi er nødt til at læse for at kunne sende
6          SPIS_1_WriteTxData(buttonData); // Send signal afsted om trykket
7      }
8 }
```

Slaven læser tilstanden på pin `Pin_2_Button` som er aktiv lav, for derefter at sende et signal ud såfremt den bliver trykket ned. Koden kører i samme loop som modtagelsen af signal fra Masteren, når LED skal tændes (Se bilag), men er klippet ud i kodestykke 3.3.

3.2.3 Hardwaredesign

Hver PSoC er forbundet til sin egen computer, der gennem fysiske såvel som interne pins og fælles stel kan kommunikere med hinanden. Forbindelserne for Master ses på tabel 2, og forbindelserne for Slave ses på tabel 1.

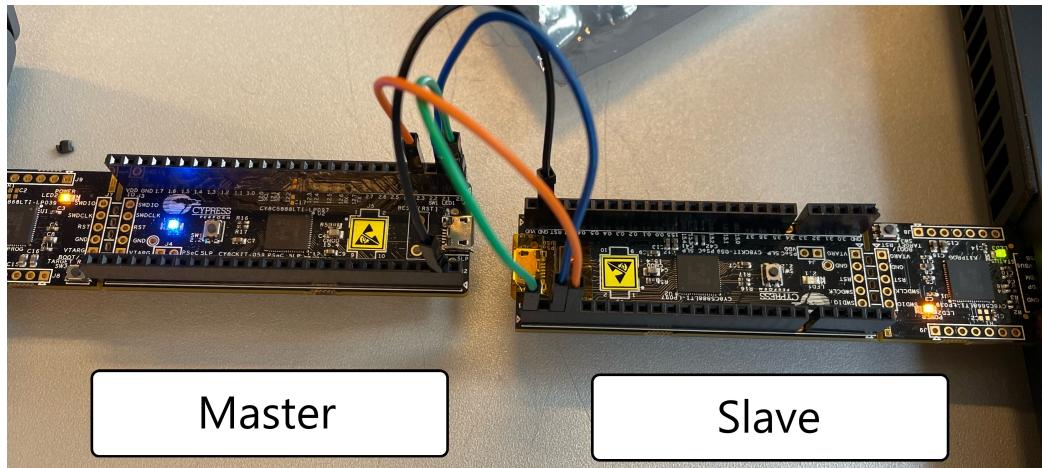
Navn	Port
MISO_1	P2[0]
MOSI_1	P2[3]
SCLK_1	P2[4]
Pin_1_LED	P2[1]
Pin_2_Button	P2[4]

Tabel 1: Slave SPI Forbindelser

Navn	Port
MISO_1	P2[0]
MOSI_1	P2[1]
SCLK_1	P2[3]
Rx_1	P12[6]
Tx_1	P12[7]

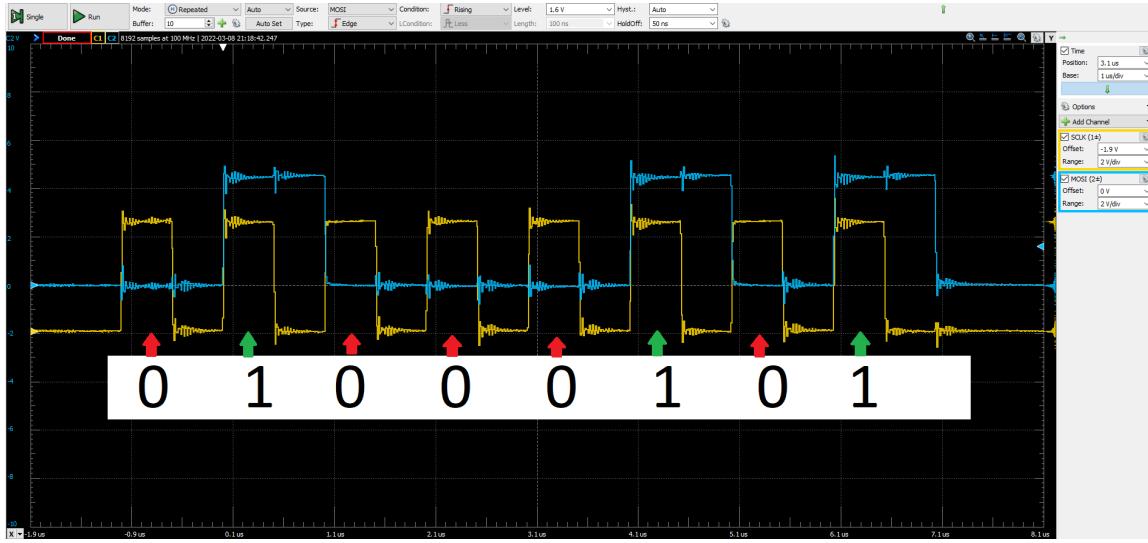
Tabel 2: Master SPI Forbindelser

Realiseringen af øvelsen inklusiv forbindelserne ses på figur 4.

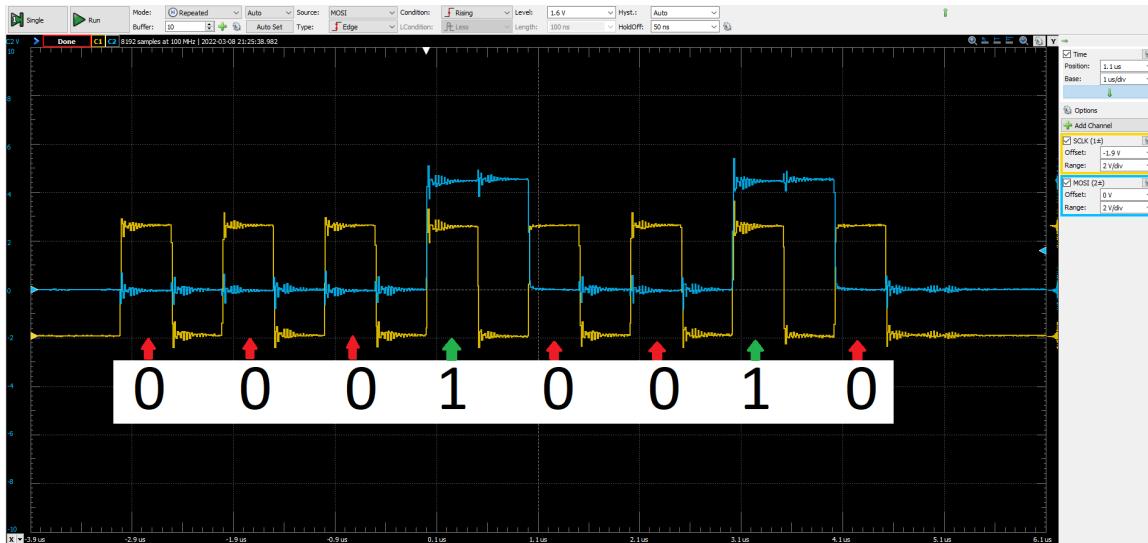


Figur 4: SPI Opstilling

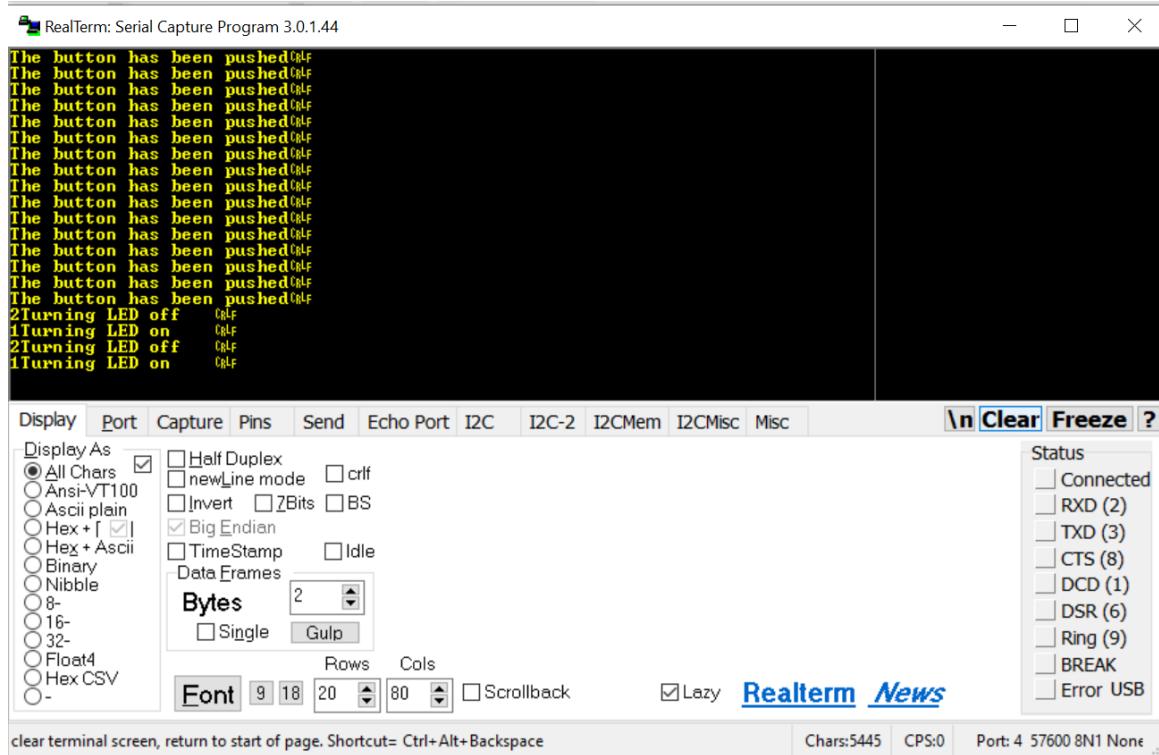
3.3 Resultater



Figur 5: Oscilloskop måling af datakommunikation for tænd LED



Figur 6: Oscilloskop måling af datakommunikation for sluk LED



Figur 7: Realterm screenshot

3.4 Diskussion

PSoC'ens SPI implementering i hardwaren gør brug af en FIFO (First in First out) buffer, som er 4 bit lang. FIFO har et register, hvorfra data transmissionen sker. Bufferen er cirkulær, hvilket betyder, at når sidste bit er skrevet til, hopper den selv tilbage til 0.

Bufferen har en skrive-pointer, der skriver og holder øje med hvor der skal skrives i bufferen næste gang. Den har også en læse-pointer, der bestemmer hvorhenne i bufferen der skal læses. Læsepointeren kan naturligvis ikke gå længere frem end skrive-pointeren, og med SPI protokollen eksisterer der to forskellige faser for transmissionen.

CPHA = 0:

Data bliver sendt afsted fra læse-pointeren til transmitter registret i starten af en clockpuls, hvor dataen først kan læses når hele transmission er færdig. Det kan derfor godt se ud som om, at dataen altid er én bit forsinket.

CPHA = 1:

Data bliver sendt afsted fra læse-pointeren til transmitter registret i slutningen af en clockpuls, og dataen fra læse-pointeren bliver overført før en ny transmission kan starte. Kontra fase 0, vil dataen der bliver skrevet, også være det der i samme tidspunkt bliver overført.

I øvelsen er det blevet valgt, at SPI kommunikationen skulle gøre brug af SPI Mode 1, hvor **CPOL = 0** og **CPHA = 1**. Som beskrevet i ovenstående tekst, og som set på tabel 3, læses dataen på falling edge og sendes ud på rising edge.

SPI Mode	CPOL	CPHA	Clock Polarity in Idle State	Clock Phase Used to Sample and/or Shift the Data
0	0	0	Logic low	Data sampled on rising edge and shifted out on the falling edge
1	0	1	Logic low	Data sampled on the falling edge and shifted out on the rising edge
2	1	1	Logic high	Data sampled on the falling edge and shifted out on the rising edge
3	1	0	Logic high	Data sampled on the rising edge and shifted out on the falling edge

Tabel 3: SPI modes [dhaker·2018]

Hvilken mode der bliver brugt har ikke den helt store betydning for øvelsen, da der i størstedelen af koden konstant bliver sendt data afsted, hvilket minimerer konsekvensen af en forsinket bit.

Det ses også tydeligt på oscilloskop billederne på figur 6 og figur 5, at der læses data når SCLK signalet gør højt (Pilene er en smule misvisende, og burde have været placeret ved rising edge). Som noteret på billederne, stemmer den afsendte data fra protokollen overens med det, der bliver læst igennem oscilloskopet.

3.5 Konklusion

Der er succesfuldt blevet lavet en data protokol til kommunikation mellem de to PSoC'er. Hvordan SPI kommunikationen i PSoC'en er implementeret er blevet forklaret, og fordelen ved den valgte

mode er blevet uddybet. Der er igennem målinger på et oscilloskop blevet undersøgt, hvorvidt den valgte data igennem protokollen er blevet sendt korrekt afsted, hvilket både blev bekræftet visuelt på PSoC'ens LED og igennem oscilloskop.

Det lykkedes at tænde og slukke for en LED på Slaven gennem Masterens UART, og ligeledes lykkedes det at aflæse knappens tilstand fra Slaven hen til Masteren.