

I3GFW - LAB Experiment 2

Communication busses

SW	Gustav Nørgaard Knudsen	AU612485
SW	Rasmus Møller Nielsen	AU633064
SW	Christian Bach Johansen	AU577526



Hold nr. 4

Contents

1	Introduktion	1
2	I2C Experiment: PSoC Master and LM75 Slave	1
2.1	Introduktion	1
2.2	Overvejelser	1
2.2.1	LM75 \iff PSoC (I2C Master)	1
2.2.2	PSoC \iff PC (UART)	2
2.3	Implementering	3
2.3.1	Indlæsning af data fra I2C	4
2.3.2	Print af float over UART	5
2.4	Dokumentation	7
2.5	Diskussion	8
2.6	Konklusion	9
3	SPI Experiment: PSoC Master and PSoC Slave	9
3.1	Introduktion	9
3.2	Overvejelser	9
3.2.1	SPI	10
3.3	Implementering	11
3.3.1	SPI Slave	11
3.3.2	SPI Master	12
3.4	Dokumentation	14
3.5	Diskussion	16
3.6	Konklusion	16
4	(Optional)I2C Experiment: PSoC Master and PSoC Slave	17

1 Introduktion

2 I2C Experiment: PSoC Master and LM75 Slave

2.1 Introduktion

I denne opgave skal vi undersøge, hvordan man med en I2C master kan snakke med temperatur sensoren LM75. Hertil skal vi bruge viden fra tidligere opgaver og snakke med vores master gennem en UART forbindelse. Dette gør vi for at aflæse LM75'erens målinger.

2.2 Overvejelser

For at vi kan snakke med LM75'eren er der 2 forbindelser vi skal fokusere på.

1. LM75 \iff PSoC (I2C Master)
2. PSoC \iff PC (UART)

Disse forbindelse vil vi følgende beskrive og kigge nærmere på, hvilke udfordringer der opstår og hvordan vi planlægger at overkomme dem.

2.2.1 LM75 \iff PSoC (I2C Master)

Denne forbindelse foregår via I2C og her skal vi fra PSoC'en sende beskeder som LM75'en modtager og håndtere. Den opbygning vi skal give beskederne kan vi se i datasheetet¹. Først skal vi sætte den adresse, som vi skal kommunikere med LM75'en igennem. Dette gør vi med følgende besked struktur.

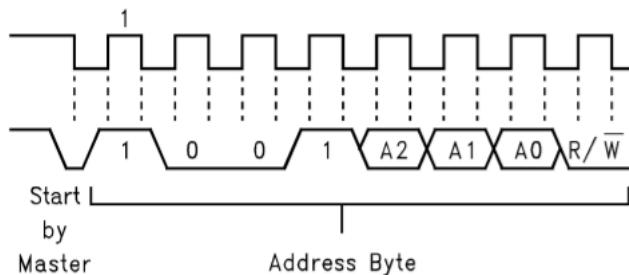


Figure 1: I2C adresse til LM75

Efter at have sat adressen skal vi modtage temperaturen. Den modtager vi som to 8-bit integers og den følger følgende format:

¹<https://www.ti.com/lit/ds/symlink/lm75b.pdf>

Temperature	Digital Output								Hex
	Binary								
125°C	0 1111 1010								0FAh
25°C	0 0011 0010								032h
0.5°C	0 0000 0001								001h
0°C	0 0000 0000								000h
-0.5°C	1 1111 1111								1FFh
-25°C	1 1100 1110								1CEh
-55°C	1 1001 0010								192h

Figure 2: Temperatur formattet fra LM75

Problematikken kommer i at få rykket rundt og behandlet de 2 bytes vi får til kun 1 enkelt byte, hvor vores temperatur er ændret fra 2's kompliment til unsigned. De to bytes vi modtager kommer til at ligne følgende:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MSB								LSB	x	x	x	x	x	x	x

I ovenstående tabel er x brugt til at vise bits som er ligegyldige for os.

Herefter planlægger vi at omskrive det til 1 byte med følgende trin:

- Gem fortegn (+/-) MSB
- Bitshift LSB til plads 15
- Bitshift MSB'en ud og resten af første byte 1 til plads 0
- OR de 2 bytes sammen så vi får LSB ind på plads 7
- Hvis MSB er 1 (-) skal vi invitere plads 0-7 og trække 1 fra for at fjerne 2's compliment
- Returner det halve og cast til en float

Da LM75'en giver os antallet af halve grader halverer vi resultatet og returner det i stedet. Så ved stue temperatur ville man få 40 fra LM75'en i stedet for 20. Dette kan vi herefter sende videre til vores PC gennem UART.

Når denne protokol er opbygget kan vi bruge den til opsætningen af flere LM75'er. Her skal vi bare indstille adressen til en anden og så kan vi forbinde dem alle serielt. Ved at gøre dette kan vi få flere LM75 slaver på samme kommunikations bus. I koden skal vi loope gennem de forskellige adresser og spørge dem en af gangen, når vi så har fået svar skal vi lukke forbindelsen og spørge den næste.

2.2.2 PSoC \leftrightarrow PC (UART)

For at snakke mellem PSoC'en og vores computer bruger vi et UART komponent. Dette skal sættes op sådan at PSoC'en sender den læste data fra LM75'en til PC'en. På grund af vi ikke bruger computerens input til noget, har vi ikke noget interrupt på RX benet.

Vores computer sættes op med RealTerm til at modtage fra den USB port som PSoC'en er sat til. Selve formateringen af teksten foregår alt sammen på PSoC'en.

Et af problemer i denne forbindelse er, hvordan vi håndtere at sende vores temperatur værdi som en "floating point" værdi. Dette problem overkommes dog forholdvist nemt ved at følge en guide² givet i undervisningen. Først skal man ind i build settings og sætte float formatting til *TRUE* og herefter skal man bare øge heap sizen til *0x200*. Når dette er gjort kan man caste sin unsigned interger til en float og printe den med printf ved brug af formaterrings type fieldet "%f".

2.3 Implementering

På Figure 3 kan man se topdesignet for vores PSoC. Her har vi et I2C modul til at snakke med LM75'eren og UART modulet til at snakke med computeren.

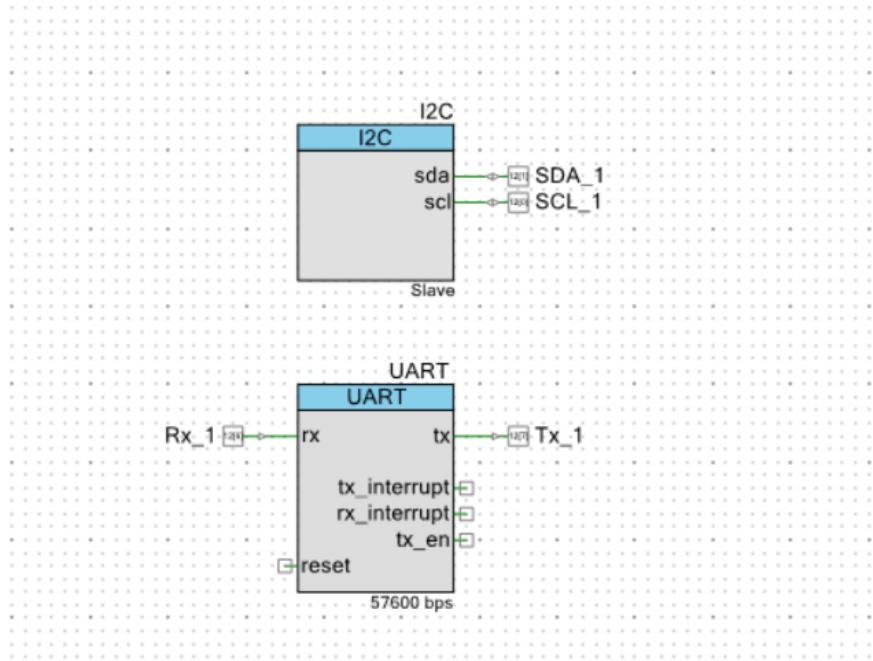


Figure 3: Top design til I2C master

Herefter bestemmer vi, hvor alle pins skal placeres. På Figure 4 kan man se, hvordan pin'sne er blevet fordelt.

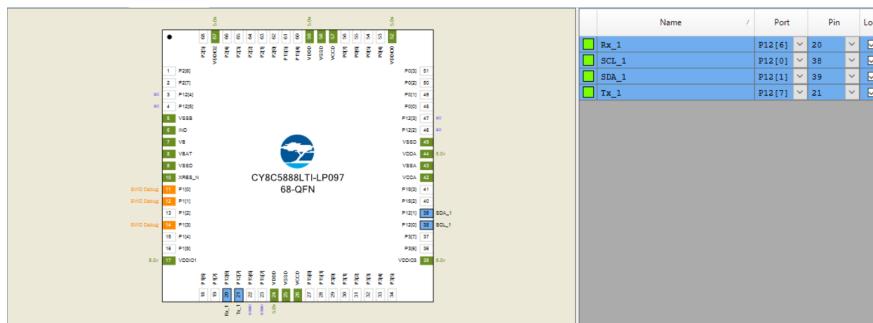


Figure 4: Pin setup på I2C master

²GFV Lektion 5 Communication buses - lab experiment Handouts: PSoC-Creator-Printing-Floating-Point.pdf

I koden er der 2 primære dele:

1. Indlæsning af data fra I2C
2. Print af float over UART

Disse dele vil følgende blive gennemgået og der vil blive beskrevet, hvordan koden modtager og håndterer vores input.

2.3.1 Indlæsning af data fra I2C

Efter at vi i *main()* loopet har startet og slået interrupten for I2C til, skal vi ha lavet en funktion som modtager input fra en I2C adresse og sender det ud som en float. Vores prototype hedder:

```
1 uint8 i2cRead( uint8 addr , float *result );
```

Først har vi returntypen uint8, den bruges til at vise om vi har fået en fejl eller ej, hvis der ikke er en fejl returnere vi '1'. Herefter har vi adressen til vores I2C device, den specificere hvilket device vi gerne vil have data fra. Til sidst har vi resultatet. Dette modtager vi som en adresse for at kunne parse det ud af funktionen samtidigt med at vi har en fejlkode fra returværdien.

Listing 1: i2cRead()

```
1 uint8 i2cRead( uint8 addr , float *result ){
2     uint8 buffer [BUFFER_LEN];
3     I2C_Init ();
4
5     I2C_MasterReadBuf(addr , buffer , BUFFER_LEN, I2C_MODE_COMPLETE_XFER );
6
7 // Wait for transfer to complete
8     while(I2C_MasterStatus () == I2C_MSTAT_XFER_INP );
9     if(I2C_MasterStatus () != I2C_MSTAT_RD_CMPLT){
10         // Re-init I2C after fault
11         I2C_Init ();
12         // Tranfer err
13         return 0;
14     }
15
16 // Interpret LM75 2s compliment into proper float
17     buffer [1] >>= 7;
18     _Bool comp = buffer [0] & 0b10000000 ;
19     buffer [0] <=> 1;
20     uint8 total = buffer [0] | buffer [1];
21     if(comp)
22     {
23         --total;
24         total ^= 0xff;
25     }
26
27     *result = (float)total * 0.5;
28
29 //no err
```

```

30     return 1;
31 }
```

Variablen BUFFER_LEN er defineret som 2.

På listing 1 kan man se, hvordan vi i første halvdel opretter en buffer, som vi bruger til at opbevare vores data fra I2C forbindelsen. Herefter tjekker vi for fejl og genstarter I2C, hvis der er fejl.

Efter dette følger vi bare de trin der blevet specificere i afsnit 2.2.1 for at gøre det til en enkelt byte. Denne byte smider vi ind på adressen fra result og herefter returnerer vi '1' da der ikke har været nogle fejl.

2.3.2 Print af float over UART

På listing 2 kan man se vores main funktion. Kort beskrevet, så modtager den floats fra *i2cRead()*, kopiere det over i vores printBuf sammen med vores forklarende tekst streng og til sidst printer den det over UART til vores computer, hvor vi kører RealTerm til at modtage dataen fra UART fobindelsen.

Listing 2: main()

```

1 int main(void)
2 {
3     CyGlobalIntEnable; /* Enable global interrupts. */
4
5     // Initiate UART & I2C
6     UART_Start();
7     I2C_Start();
8     I2C_EnableInt();
9
10    for(;;)
11    {
12        if(!i2cRead(0x48, &temp1)){
13            //temp error value
14            temp1 = -0.1;
15        }
16        sprintf(printBuf, "Temperaturen_paa_slave_1_er : %.1f\r\n", temp1);
17        UART_PutString(printBuf);
18        CyDelay(1);
19
20        if(!i2cRead(0x49, &temp2)){
21            //temp error value
22            temp2 = -0.1;
23        }
24        sprintf(printBuf, "Temperaturen_paa_slave_2_er : %.1f\r\n", temp2);
25        UART_PutString(printBuf);
26        CyDelay(500);
27    }
28 }
```

Variablen PRINT_LEN er defineret til 50.

En vigtig del af vores main loop er det delay vi indsætter efter vi har læst fra begge LM75'ere. Dette gør vi for ikke at læse for hyppigt på slaverne. Hvis vi gjorde det risikerede vi at overophede slaverne og dermed få

en forkert temperatur, da LM75'eren har en anden temperatur end den omkringværende.

Fra datasheetet³ lyder det nemlig at

"The LM75 should not be accessed continuously with a wait time of less than 300 ms."

Hvis vi spørger oftere end 300ms risikerer vi at afbryde en temperatur læsning hos LM75'en. Så starter den bare forfra og dette kan risikere, at den skaber en unødig varme. Derfor har vi valgt at indsætte et delay på 500ms. Se linje 29 på listing 2.

Efter vi har klaret alt dette kan vi nu lave vores opstilling med PSoC, LM75 boards og computer.

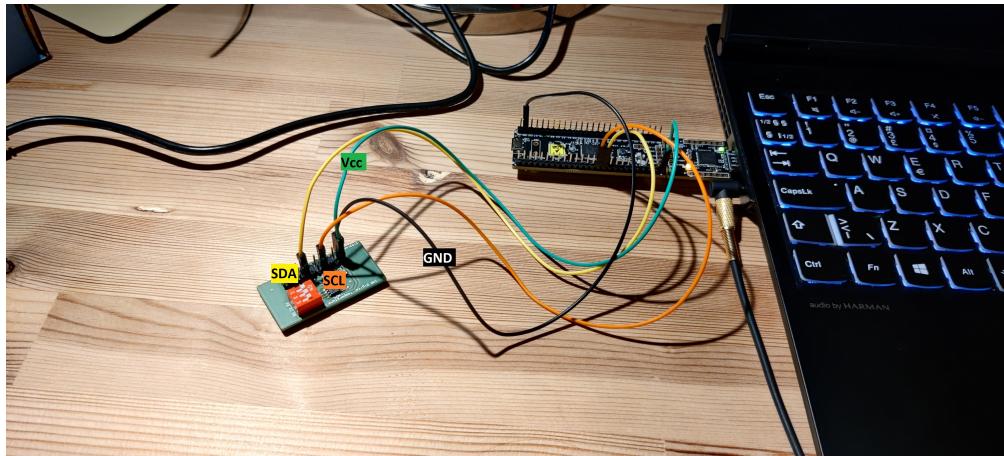


Figure 5: Opstilling af I2C forbindelser

På Figure 5 kan man se opstillingen, hvorpå der er indsat bokse, der beskriver ledningernes formål.

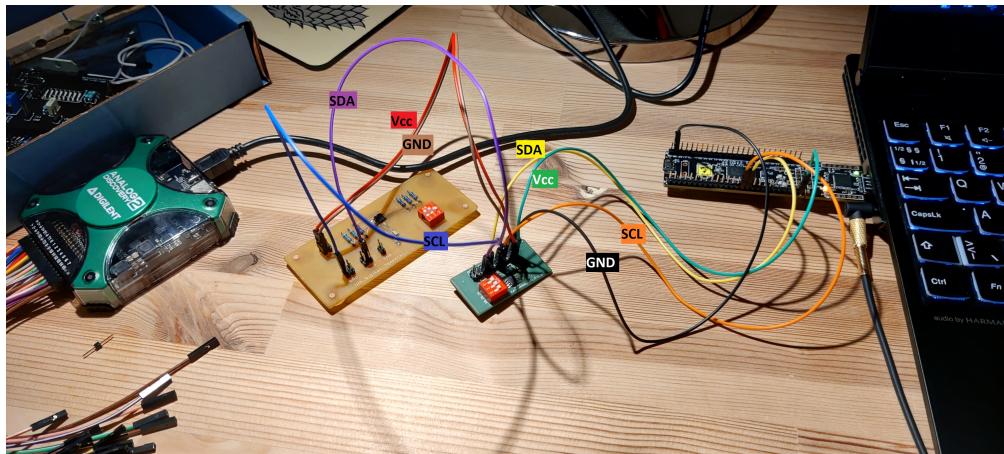


Figure 6: I2C opstilling med 2 LM75 slaver

³<https://www.ti.com/lit/ds/symlink/lm75b.pdf?> - Side 5, Afsnit 6.5, note (5)

På Figure 6 kan man se opstillingen, hvor vi har 2 LM75'ere forbundet til kommunikations bussen. Her er der igen indsats beskrivende bokse.

2.4 Dokumentation

På Figure 7 kan man se vores resultat fra computeren når vi har 1 slave på vores kommunikations bus. Her aflæser vi en temperatur fra 27 grader til 30 grader.

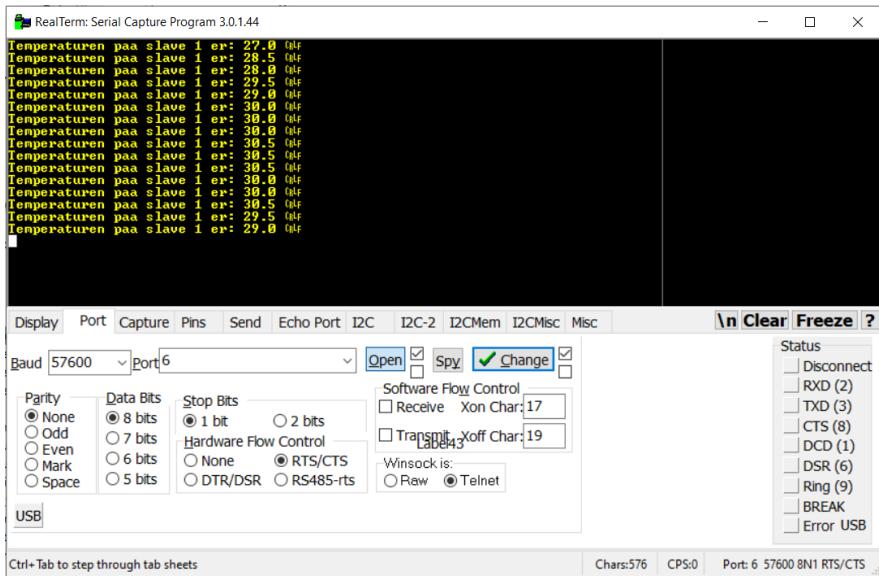


Figure 7: I2C forbindelse med 1 slave

Herefter på Figure 8 kan man se, resultatet efter vi forbandte 2 slaver til kommunikations bussen.

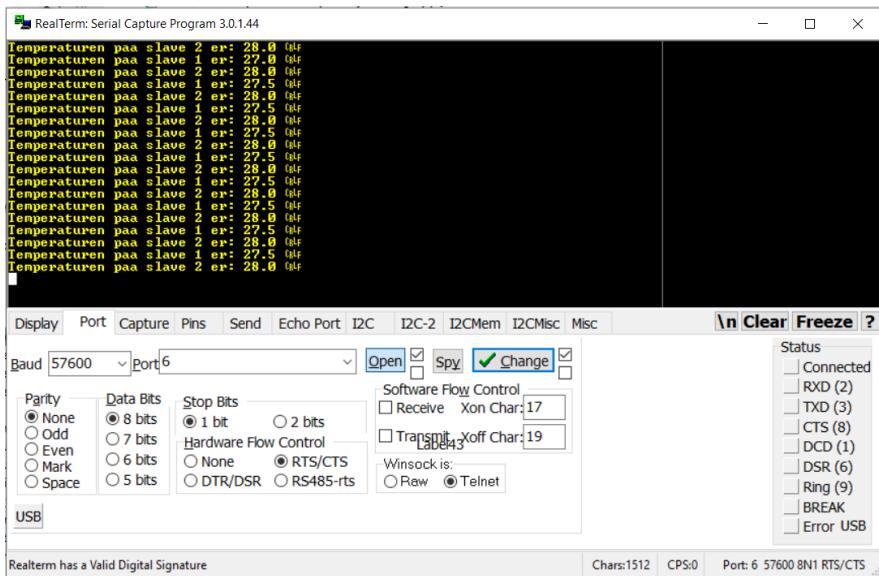


Figure 8: I2C forbindelse med 2 slaver

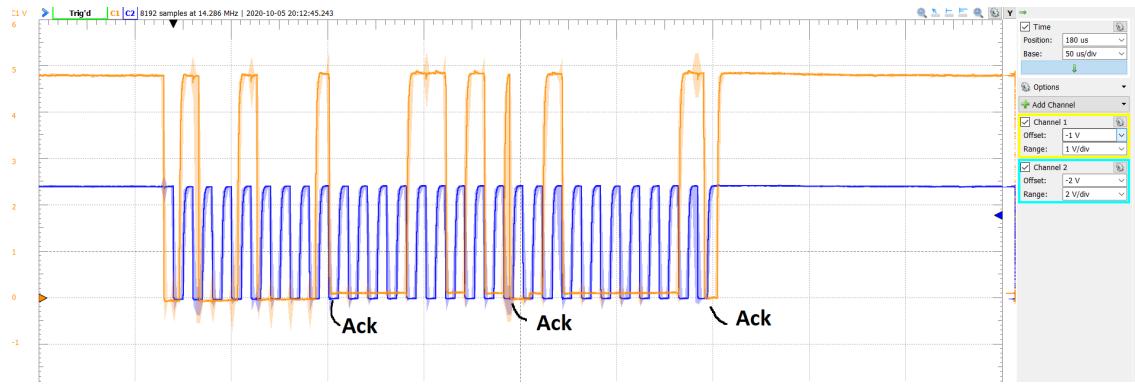


Figure 9: Oscilloscope billede med clock

På Figure 9 kan man se vores I2C data på channel 1 (den orange) og samtidigt på channel 2 (den blå) kan man se clock signalet, der synkronisere det hele.

2.5 Diskussion

På Figure 7 kan man se, at temperaturen går mellem 27 grader og 30 grader. Denne ændring kom da vi placerede en finger på sensoren og det passer dermed at temperaturen stiger til 30 grader. Dette passer med forventningen om, at den omkringværende luft er koldere en vores finger.

Tilgengæld på Figure 8 kan man ikke se en særlig stor forskel. Denne forskel kommer af, at vi ikke placerede en finger eller et varmt objekt på en af sensorne. Dermed har vi 2 værdier som ligger meget tæt op af hinanden fordi luften omkring dem er ens temperatur.

På Figure 9 kan man se, hvordan vores protocol følges af signalet. Inden den første acknowledgement har vi PSoC'en der sender adressen ud på data linjen og herefter kan man se, på grund af, at ground bliver forskudt fra vores PSoC's ground, at det er LM75'en der svarer tilbage. Nu ved vi at LM75'en svarer med 2 bytes af data og vi kan se, at ind imellem disse 2 bytes er der en acknowledge fra masteren. Dette kan man igen se på den lille forskel der sker på vores data forbindelse og at vores LOW bliver skudt en lille smule op fra 0V. Derudover kan man se, at vores data kun skifter når vores clock er LOW, dette er specificeret i protocollen for I2C⁴, hvor den siger:

"The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW (see Figure 4). One clock pulse is generated for each data bit transferred."

Figure 4, som quoten referer til, kan ses på Figure 10.

⁴<https://www.nxp.com/docs/en/user-guide/UM10204.pdf> - Side 9, Afsnit 3.1.3 Data validity

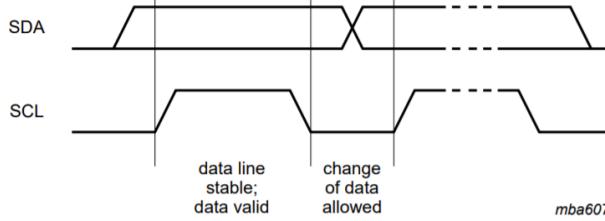


Fig 4. Bit transfer on the I²C-bus

Figure 10: Specifikation for ændring af Data linjen

Et af de problemer vi løb rigtigt meget ind i under vores programmering af PSoC'en var, at når vi havde begge LM75'ere forbundet opstod fik vi ikke ventet på, at den første var færdig med at skrive på bussen og derved når vi skulle læse fra den næste LM75 læste vi bare en masse blandet fra begge slaver. Derfor har vi det lille tjek på linje 8 & 9 i listing 1.

2.6 Konklusion

På baggrund af denne øvelse kan vi konkludere, at man kan kommunikere LM75 med over en I²C forbindelse. Hertil kan man forbinde flere LM75'ere (slaver) på samme kommunikations bus og derved aflæse fra flere slaver over samme forbindelse. Dog er det vigtigt, at man ikke spørger for ofte, da det kan øge risikoen for at få en forkert temperaturn.

Herudover kan vi konkludere, at vi modtager temperaturen over 2 Bytes og med forholdsvis få trin kan man få det omregnet til en enkelt byte, som man herefter meget nemmere kan sende til sin computer og få vist.

3 SPI Experiment: PSoC Master and PSoC Slave

3.1 Introduktion

I denne del af øvelsen vil vi nu arbejde med kommunikations interfacet. Der vil hertil benyttes 2 styk PSoC 5LP, som implementeres som henholdsvis slave og master i systemet. Vha. en UART-forbindelse til PC, vil vi så forsøge at implementere vores master så instrukser sendt fra PC kan sendes gennem SPI-protokollen fra master til slave.

3.2 Overvejelser

Kommunikation mellem master og slave vil som sagt foregå gennem SPI. Vores fokus vil derfor ligge i at sikre, at signalet sendt fra master og signalet modtaget af slave benytter den korrekte protokol.

Vi forventer at benytte AD til at teste hvorvidt signalerne sendes og modtages korrekt, og om det rigtige antal bits benyttes. Desuden er vi blevet informeret om at SPI-protokollen i PSoC-Creator benytter en cirkulær buffer, hvilket vi vil uddybbet senere i opgaven.

3.2.1 SPI

SPI fungerer som en datastrøm imellem 2 devices. Det er kun muligt at sende på en SPI, hvis der også modtages fra den samtidig. Modtageren og senderen fungerer som en cirkulær buffer, som læses og skrives til hele tiden. Man bør derfor være opmærksomme på ikke at komme bagud i bufferen, så vores data bliver overskrevet. Da protokollen til denne opgave er forholdsvis nem, og kan anses for binær, er det valgt simpelthen at ignorere bufferen, og nulstille den, hver gang den bruges.

Fobindelsen mellem Master og Slave er forholdsvis simpel i denne implementering. Der skal oprettes en MOSI, MISO og en SCLK. Det er vigtigt vores SPI Master og SPI Slave kører på samme modes. Vi har defor brugt standardinstillingen på modes, som er $CPHA = 0$ og $CPOL = 0$. CPOL er vores clock polaritet. I mode 0 starter klokken LOW, når der ikke tælles. CPHA fortæller hvornår vores bits bliver læst. Ved 0 er det ved starten af en clock puls der samples, ved 1 er det i slutningen der samples.

Til at opstille Slave modulet, skal vi desuden bruge en Slave Select. Da der i dette system kun eksisterer en enekelt Slave, vælges der at forbinde Slave Select direkte til GND.

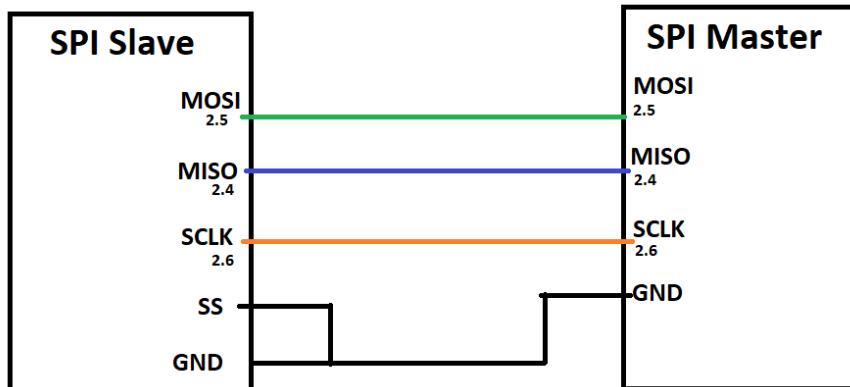


Figure 11: SPI setup diagram

Signal	Beskrivelse
MOSI	Står for Master-Out-Slave-In. Signal i bits der sendes fra master til slave.
MISO	Står for Master-In-Slave-Out. Signal i bits der sendes fra slave til master.
SCLK	Master intern clock, styrer synkronisering af sending af bits.
SS	Master Slave Select. Sættes LOW for at kommunikere til slaven er der sendes til den.
GND	Fælles stel forbindelse for slave og master

3.3 Implementering

Vi vil her introducere hvordan vi har opbygget vores master-slave system, og hvordan det er blevet implementeret i softwaren.

3.3.1 SPI Slave

SPI Slave top design ser ud som på figur 12. Den består af vores SPI modul, en UART til debugging samt en LED. LEDen er forbundet via hardware til pin 2.1, UART er forbundet Tx=12.7 og Rx=12.6, da disse er forbundet til USBen. SPI har ikke en forudbestemt pin opsætning, så den er forbundet MISO=2.4, MOSI=2.5 og SCLK= 2.6. Slave Select er bundet direkte til GND, da vi ønsker at forsimple implementeringen.

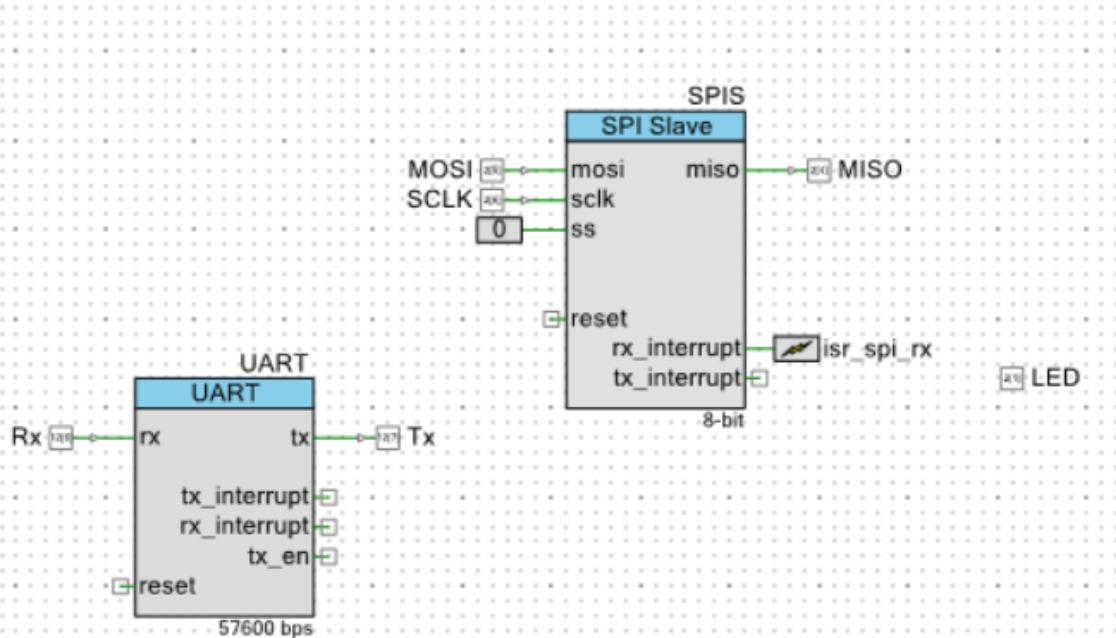


Figure 12: SPI Slave top design

Selve Slaves funktionalitet ligger i den SPI interrupt handler der er beskrevet i listings 3. Funktionen tager læser data fra SPI bufferen, printere dette til UART-debuggeren, nulstiller bufferen og tager så stilling til LEDens tilstand ud fra hvilken data er læst. Da funktionaliteten af SPI slave er den samme i de to implementeringer (med og uden knap) er her kun vist en.

Listing 3: SPI Slave

```

1 CY_ISR(isr_spi_rx_handler)
2 {
3     uint8_t data = SPIS_ReadByte();
4
5     // Write to terminal for DEBUGGING
6     char buf[20];
7     sprintf(buf, "Data_received : %x\r\n", data);
8     UART_PutString(buf);
9
10    // Clear buffer, makes it easier
11    SPIS_ClearRxBuffer();
12
13    switch(data)
14    {
15        case 0xcc:
16        {
17            LED_Write(1);
18            break;
19        }
20        case 0x55:
21        {
22            LED_Write(0);
23            break;
24        }
25        default:
26        {}
27    }
28 }
```

3.3.2 SPI Master

Da der både skal implementeres en SPI Master der kan styres via UART, og en der kan styres via en knap på PSoC, er der implementeret to forskellige versioner ad SPI Master. Deres top design kan ses her under på figur 13 og 14. De er begge implementeret med en UART, hvor dens pins er sat Tx=12.7 og Rx=12.6. SPI er sat op på samme måde som i SPI Slave, med MISO=2.4, MOSI=2.5 og SCLK=2.6. Der er ikke brug for at definere Slave Select, da dette ikke er relevant for opgaven. Desuden er der tilføjet en software forbindelse til kanppen på PSoC boardet, via pin 2.2.

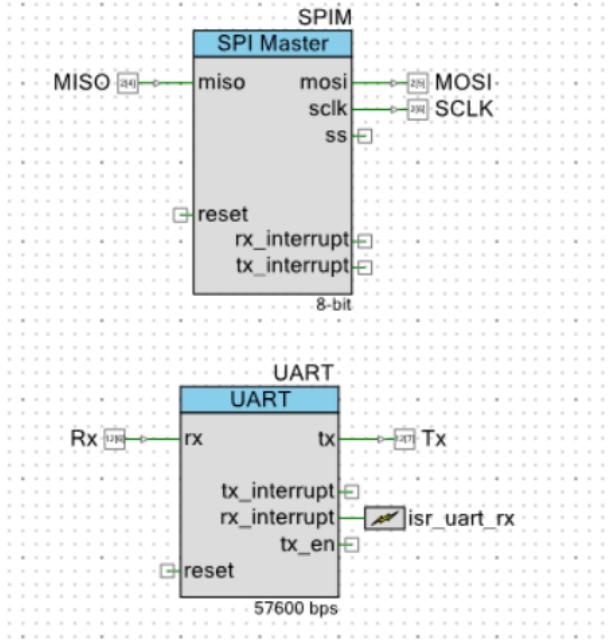


Figure 13: SPI Master top design

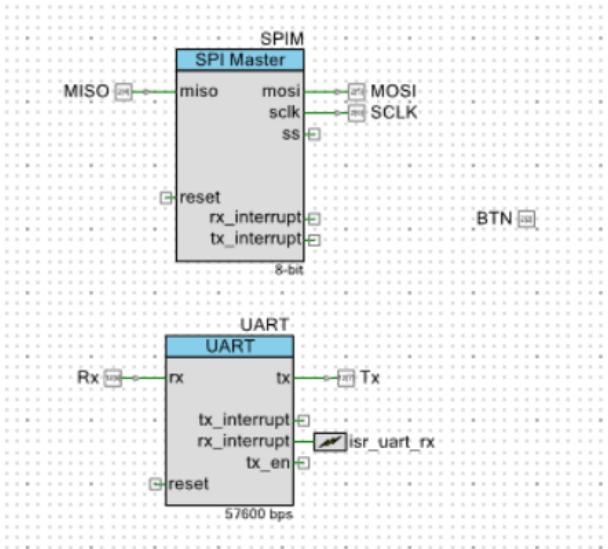


Figure 14: SPI Master top design - med knap

På listing 4 kan ses den funktion der håndterer hvad der modtages på UART. Når der er modtaget besked via UART,

Listing 4: Håndterin af Bytes Received fra UART til SPI Master

```
1 void handleByteReceived( uint8_t byteReceived )
```

```

2  {
3      switch(byteReceived)
4      {
5          //ON case
6          case 'i':
7          {
8              SPIM_ClearTxBuffer();
9              SPIM_WriteTxData(0xcc);
10             break;
11         }
12         //OFF case
13         case 'o':
14         {
15             SPIM_ClearTxBuffer();
16             SPIM_WriteTxData(0x55);
17             break;
18         }
19         default:
20         {}
21     }
22 }
```

Listing 5: Håndtering af knappens input på SPI Master - med knap

```

1 for (;;)
2 {
3     CyDelay(20);
4     if (!BTN_Read())
5     {
6         SPIM_ClearTxBuffer();
7         SPIM_WriteTxData(0xcc);
8     }
9     else
10    {
11        SPIM_ClearTxBuffer();
12        SPIM_WriteTxData(0x55);
13    }
14 }
```

3.4 Dokumentation

På Figure 15 nedenfor ses vores endelige opstilling. Forbindelserne mellem slave og master, beskrevet i implementeringsafsnittet, er etableret som specificeret, og vi ser som forventet at vi kan tænde og slukke for LED'en på slave PSoC'en ved at sende i/o til vores master PSoC.

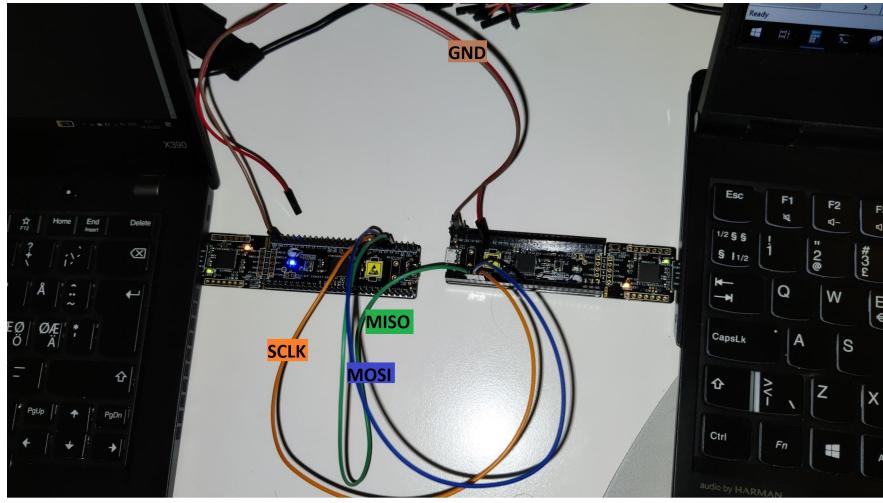


Figure 15: SPI forbindelse mellem Slave (venstre) og Master (højre)

For at få en bedre forståelse af hvad der sker signalmæssigt, og for at sikre systemet ikke bare virker ved et uheld, benytter vi Analog Discovery Oscilloskop til at måle i/o signalet sendt mellem PSoC parret. På Figure 16 ses vores måling af signalet for at tænde LED'en, mens Figure 17 viser signalet sendt for at slukke LED'en. Den blå graf er her vores clk-signal, mens den orange graf er vores SPI-signaal.

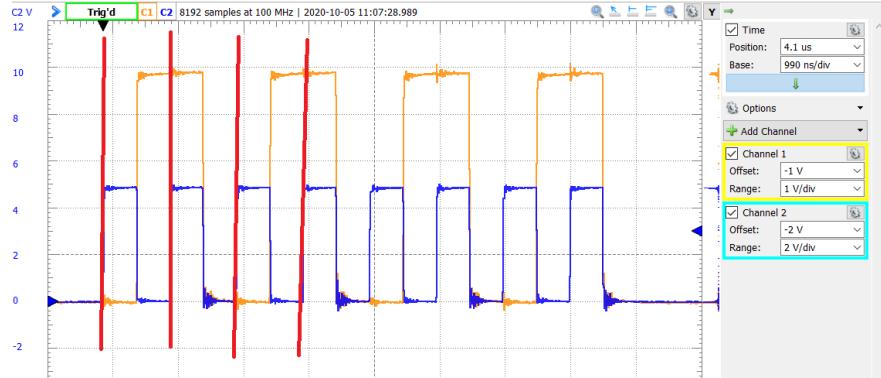


Figure 16: Modtagelse af sluk-byte

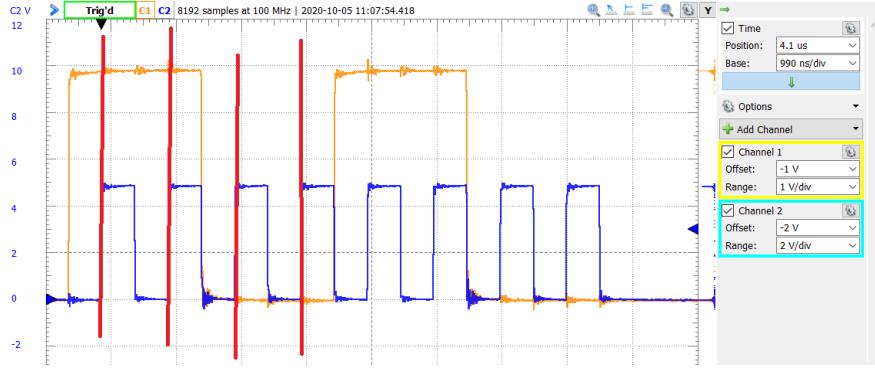


Figure 17: Modtagelse af tænd-byte

3.5 Diskussion

Som vi først ser på Figure 15 virker vores implementering, og vi kan tænde og slukke for vores slave LED ved at sende et signal til vores master PSoC. Idet vi ville sikre at SPI protokollen blev kørt korrekt, valgte vi specifikke hex-tal til at repræsentere I/O signalerne.

Dette gav dog nogle problemer under implementeringen, da vi ved målinger med Analog Discovery Spy funktion fandt at signalet modtaget af slave PSoC'en blev forskudt en halv byte ift. hvad der blev sendt af master. Vi fandt at fejlen opstod hvis et tidligere signal ikke var blevet modtaget korrekt. Hvis bufferen så ikke blev tømt efterfølgende, lagde protokollen blot dele af det nye signal ind i bufferen oveni fejsignalet.

Problemet opstår, idet SPI-protokollen ikke har adgang til en acknowledge, og derfor blot sender en stream af bits fra master til slave. En evt. løsning kunne være at gøre brug af SPI slave-select. Når SS signalet sættes lav for at indikere til slaven at der nu sendes specifikt til den, kan bufferen så tømmes. Vi valgte imidlertid ikke at bruge denne løsning, da vi ikke ønskede at overkomplificere opgaven.

Hvis vi kigger på Figure 16 ser vi det korrekte 'i' signal modtaget af slaven. Vi benyttede i dette tilfælde hex-tallet 0x55, i binært '1010101', hvilket er præcis den kombination vi ser på figuren. Ligeledes benyttede vi hex-tallet 0xcc, '11001100' i binært, til vores 'o' signal, og vi ser på Figure 15 at signalet stemmer overens.

Det er her værd at pointere at amplituden af vores SPI-signal er dobbelt størrelsen af vores clock. Dette skyldes at vi har stillet på akserne, for at gøre målingerne mere visuelt overskuelige, da vi ikke er videre interesserede i amplituderne af vores signaler.

3.6 Konklusion

På baggrund af denne øvelse konkluderer vi, at vi har formået at etablere en SPI-protokol mellem en slave PSoC og master PSoC. Hertil har vi implementeret et master-slave system, der benytter SPI til at kommunikere mellem dens master og slave moduler. Vi har ved hjælp af dette sendt en byte til slaven, som så bestemmer om vi skal tænde eller slukke på LED'en monteret på slaven.

Ligeledes har vi også formået at implementere den fornævnte SPI-protokol, dog nu ved at initiere LED tænd signalet med en switch på master.

4 (Optional)I2C Experiment: PSoC Master and PSoC Slave

Denne opgave er ikke besvaret.