AARHUS UNIVERSITET

Institut for Elektro- & Computerteknologi

Bilag G5
Lys design og implementering

Tobias Berg-Sørensen	TBS	Studienummer: 202106966	AU-id: au700512
Shukriya Bile	SB	Studienummer: 07346	AU-id: au233984
Mads Evander Jensen	MEJ	Studienummer: 202107547	AU-id: au705673
Charlotte Zacher Molbech	CZM	Studienummer: 202106286	AU-id: au705394
Wouter Jonathan Oudijk	WJO	Studienummer: 202109059	AU-id: au704640
Michael Andreas Pedersen	MAP	Studienummer: 202104658	AU-id: au706724
Jon Nørgaard Poulsen	JNP	Studienummer: 201303423	AU-id: au483121
Sandra Foss Sørensen	SFS	Studienummer: 202106467	AU-id: au681707

Vejleder: Lars G. Johansen
Kunde: Henrik Olsen

DATO
16. december 2022

Versionshistorik

Version	Dato	Hvem	Hvad
1.0	12-12-2022	MEJ	Oprettelse af dokument
1.1	14-12-2022	MEJ	Modultest
1.2	15-12-2022	MEJ	3D render af ring fil

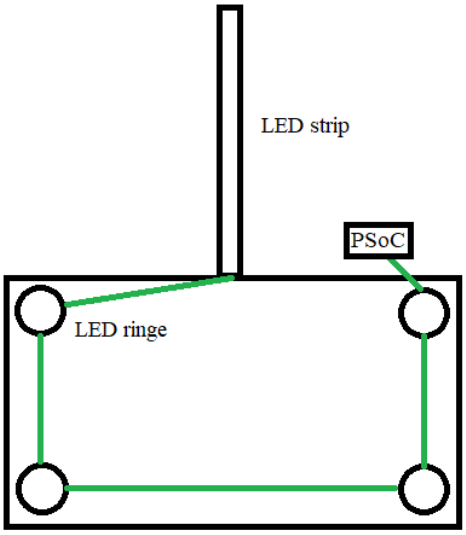
Indhold

1 Lysdesign

En central del af projektet er lyset fra spillet, både spilpladen og en scoresøjle skal kunne lyse. Vi vælger at bruge LED'er til begge dele. Der findes ikke noget standard bibliotek til PSoC'en der kan styre LED'er, så vi har valgt at lave vores egen PSoC LED driver. Det betyder også at vi har bedere kontrol over LED'erne.

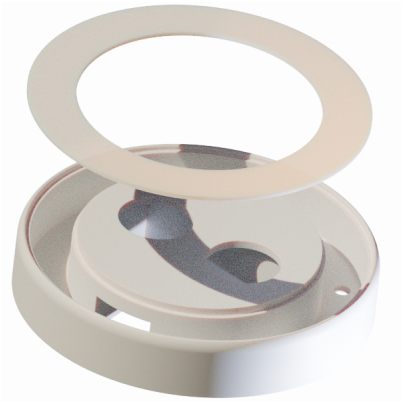
1.1 Komponent valg

Vi har flere muligheder ift. LED'er, men da det er billigst og mest udbredt, bruger vi ws2812 LED'er, også kaldt neopixels. De er veldokumenterede og nemme at arbejde med. Da vi selv har nogle på lager betyder det også at vi kan begynde at prototype med det samme. I ws2812 protokollen er der en enkelt datalinje der kan sende til en lang række LED'er, vi kan derfor serieforbinde alle LED'erne og kontrollere dem med en pin på PSoC'en. Til LED strippen bruger vi en aluminiums profil der kan holde LED'erne og til LED ringene 3d printer vi holdere. Herunder ses 3d filen vi bruger til at holde LED'erne. I midten er der plads til ultralyd sensoren.



Figur 1: Lys spilplade opsætning

(a) LED ring 3d fil belyst



(b) LED ring 3d fil åbnet

(c) 3d printet LED ring

Figur 2: LED ring design

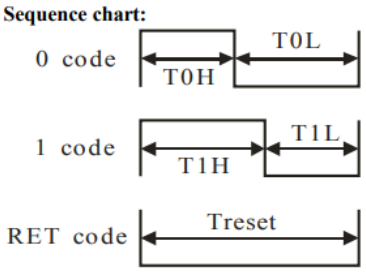
1.2 ws2812 protokol

For at komme i gang tager vi udgangspunkt i en artikel fra IOTexpert [2]. Her bruger forfatteren en PSoC 6 til at skrive ws2812 protokollen ud med SPI og en DMA. Vi vil gerne forsimple processen så meget som muligt, så vores driver er nem at forstå og ændre. Først og fremmest bruges i artiklen en PSoC 6, men en PSoC 5LP, så allerede der er der en del ændring i driveren da vi ikke har adgang til de samme komponenter. Derudover vælger vi ikke at bruge en DMA, men bare at udskrive direkte med en SPI. Det er da den kun er for at fjerne noget overhead fra processoren, men det er ikke nødvendigt i vores projekt, da vi bruger en PSoC specifikt til LED'erne der ikke skal kunne andet.
Herunder ses timing diagrammet for ws2812 LED'erne. [3]

Data transfer time(TH+TL=1.25μs±600ns)

T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.8us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.45us	±150ns
RES	low voltage time	Above 50μs	

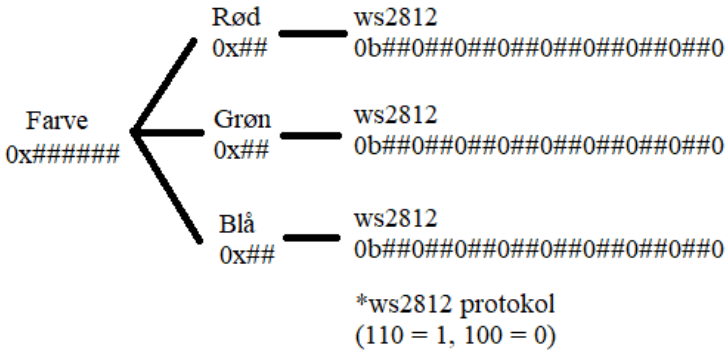
(a) ws2812 transfer time



(b) ws2812 sequence chart

Figur 3: ws2812 timing

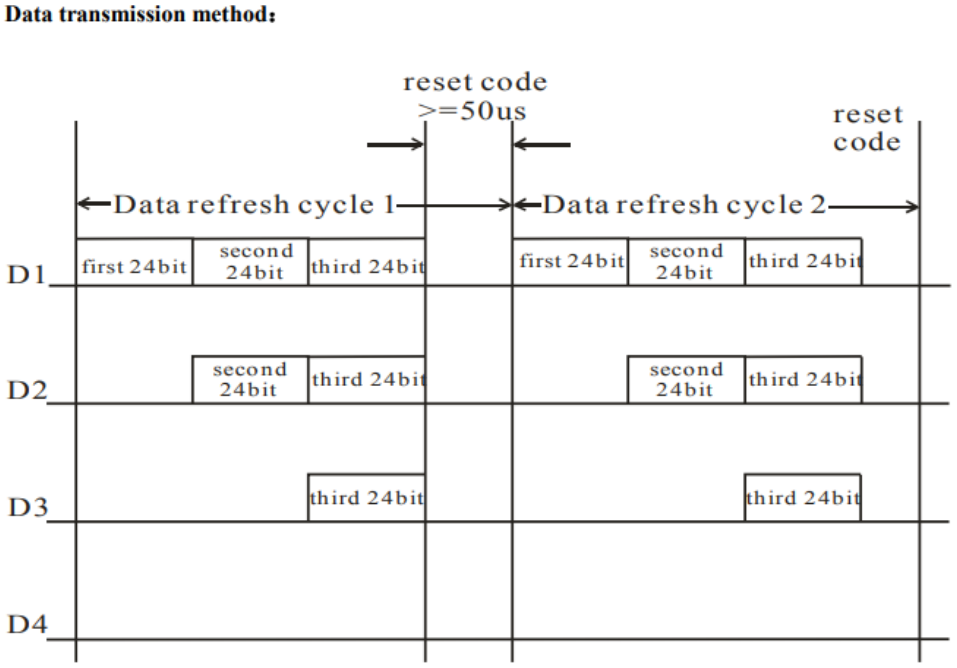
LED'erne skal altså modtage et meget præcist impulstog for at generere en farve. Vi kan se at man kan lave et helt transfer ved den laveste fællesnævner 0.4 μs. Så hvis vi tager det reciprokke får vi frekvensen vi skal sende data med; 2.5 MHz. Dette impulstog kan vi hacke SPI til at sende, hvis vi giver det en konstant data strøm og vælger at den skal sende med 2.5 MHz. Herunder har jeg skitseret idéen.



Figur 4: ws2812 protokol

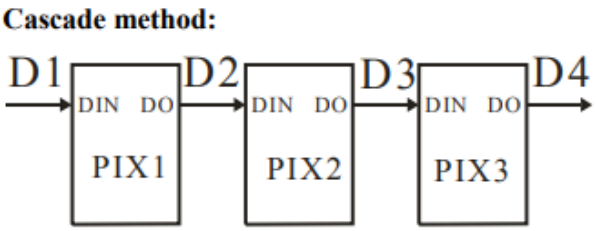
Vi vil gerne have at driveren kan modtage en HEX farve kode, så det starter vi med. Så bryder vi det ned til 3 bytes med rød, grøn og blå. Derefter kan vi konvertere det til ws2812 protokol kode, hvor 1 bit bliver til 3 hackede SPI bits. For hver LED skal vi altså sende 3 farver med 8 bit og så konvertere det til 3 bit per bit. Altså 72 bits per LED.

Udover dette skal der også være et 50 µs delay imellem hver farvekode, dette kan ses på figuren herunder.



Figur 5: ws2812 transfer metode

Det er nemt at implementere i koden med CyDelayUs. LED'erne kan på denne måde kaskade kobles i en meget lang serie, general er det ikke anbefalet at bruge mere end 1024 LED'er, men det får vi heldigvis ikke brug for.



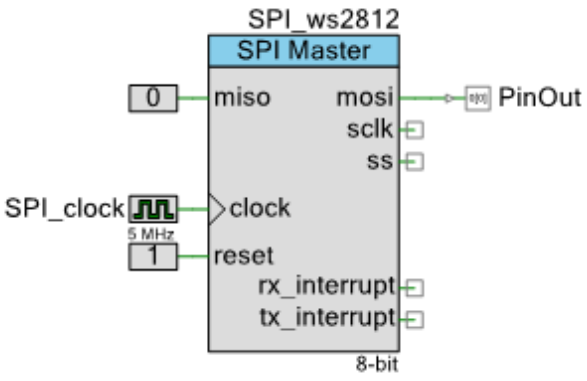
Figur 6: ws2812 kaskade metode

Det er faktisk alt vi skal bruge for at kunne lave vores egen LED driver til ws2812.

1.3 PSoC ws2812 driver

For at sende til data linjen på vores LED'er bruger vi som sagt SPI. Det starter vi med at implimentere i top designet i PSoC Creator. Vi bruger en 5 MHz clock da SPI sender hver anden clock cycle. Vi skal aldrig modtage noget så vi tilslutter miso til grund og reset til 1. Det er kun PinOut vi bruger der er forbundet til LED'ernes data linje. SPI sender 8 bit af gangen så vi skal på en måde lave et array med uint8 værdier der indeholder protokol koden. Jeg kalder dette for framebufferen.

Herunder ses min init funktion hvor jeg laver framebufferen og starter SPI. Sidst kalder jeg to funktioner til at genstarte alle LED'er. Jeg har taget nogle definationer med så man kan se hvordan jeg definere FRAMEBUFFER_SIZE. WS_ONE3 og WS_ZERO3 bruges senere til at konvertere til ws2812 protokol koden.



Figur 7: PSoC Creator ws2812 driver topdesign

```
#define WS_ONE3  (0b110<<24)
#define WS_ZERO3 (0b100<<24)
#define WS_SPI_BIT_PER_BIT (3)
#define WS_COLOR_PER_PIXEL (3)
#define WS_BYTES_PER_PIXEL (WS_SPI_BIT_PER_BIT * WS_COLOR_PER_PIXEL)
#define FRAMEBUFFER_SIZE (MAX_PIXELS*WS_BYTES_PER_PIXEL)

static uint8_t WS_frameBuffer[FRAMEBUFFER_SIZE];
static uint16_t numberOfPixels;

void ws2812_init(uint16_t numPixels)
{
        if(numPixels > MAX_PIXELS)
            return;

        numberOfPixels = numPixels;

    for (uint16_t i = 0; i < FRAMEBUFFER_SIZE; i++){
        WS_frameBuffer[i] = 0x00;
    }

    SPI_ws2812_Start();

    ws2812_setMultiRGB(0,MAX_PIXELS,0x00,0x00,0x00);
    ws2812_update();

    return;
}
```

Jeg har brugt to meget nyttige funktioner fra IOTexpert artiklen[2]. Den første ses herunder. Den konvertere en byte til 3 bytes med ws2812 protokol kode. Funktionen går igennem hver bit i input bytet, bitshifter og sætter de tidligere definerede WS_ONE3 og WS_ZERO3 ind på outputtet.

```
uint32_t ws2812_3code(uint8_t input)
{
    uint32_t rval=0;
    for(int i=0;i<8;i++)
    {
        if(input%2)
        {
            rval |= WS_ONE3;
        }
        else
        {
            rval |= WS_ZERO3;
        }
        rval = rval >> 3;

        input = input >> 1;
    }
    return rval;
}
```

Den anden funktion jeg bruger fra artiklen er denne herunder. Den opdatere et område i framebufferen der svare til en specifik LED. Her bruges også 3code funktionen så framebufferen med det samme indeholder ws2812 protokol kode. Først er en type defination der gør det nemmere at indsætte vores 24 bit værdier ind i et byte array. Så konvertere vi det første farve argument til 3code og sætter det ind på 3 bytes i arrayet. Det gør vi for alle de 3 farver. Vigtigt her at ws2812 LED'er modtager grøn først, så rød og sidst blå, så GRB i stedet for RGB.

```
void ws2812_setRGB(uint16_t led, uint8_t red, uint8_t green, uint8_t blue)
{
    if( led > (numberOfPixels))
            return;

        typedef union {
    uint8_t bytes[4];
    uint32_t word;
    } WS_colorUnion;

    WS_colorUnion color;
    color.word = ws2812_3code(green);
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+0] = color.bytes[2];
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+1] = color.bytes[1];
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+2] = color.bytes[0];

    color.word = ws2812_3code(red);
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+3] = color.bytes[2];
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+4] = color.bytes[1];
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+5] = color.bytes[0];

    color.word = ws2812_3code(blue);
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+6] = color.bytes[2];
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+7] = color.bytes[1];
    WS_frameBuffer[(led*WS_BYTES_PER_PIXEL)+8] = color.bytes[0];

    return;
}
```

Det er klart at for at ændre flere LED'er på en gang laver jeg en funktion der køre den overstående funktion i et for-loop. Det ses herunder.

```
void ws2812_setMultiRGB
  ...(uint16_t startLED, uint16_t endLED, uint8_t red, uint8_t green ,uint8_t blue)
{
        if( (startLED > endLED) || (endLED > MAX_PIXELS ) )
            return;

    for(int i=startLED; i<=endLED; i++)
        ws2812_setRGB(i, red, green, blue);

    return;
}
```

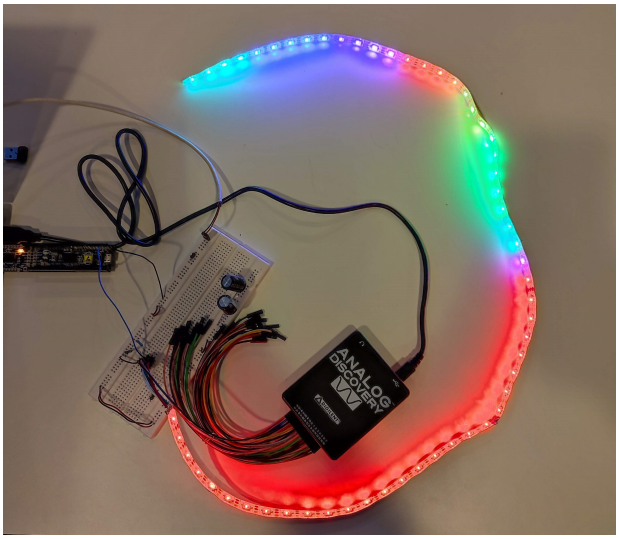
For at opdatere LED'erne bruger jeg funktionen herunder. For hver pixel går jeg igennem og udskriver til SPI. Der er et delay imellem hver LED.

```
void ws2812_update(){

    uint32_t frameBufferQueue = 0;

    for (uint32_t i = 0; i < MAX_PIXELS; i++){

        for (uint32_t k = 0; k < WS_COLOR_PER_PIXEL; k++){
            SPI_ws2812_WriteTxData(WS_frameBuffer[frameBufferQueue]);
            frameBufferQueue++;
        }
        CyDelayUs(20);
    }
        return;
}
```

Det er smart at have separate funktioner til at opdatere framebufferen og LED'erne. Det er hvis man f.eks gerne vil opdatere flere forskellige stedet på ens LED datalinje. Hvis man opdaterede LED'erne hver gang ville man miste ret meget processor kraft. Det er også vigtigt her at LED'erne selvfølgeligt alle skal opdateres på samme tid. Det er ikke muligt kun at opdatere enkelte LED'er. Det er da hver LED modtager nogle bytes indtil de er 'fulde' og så lader de næste bytes passere.

1.4 Troubleshooting

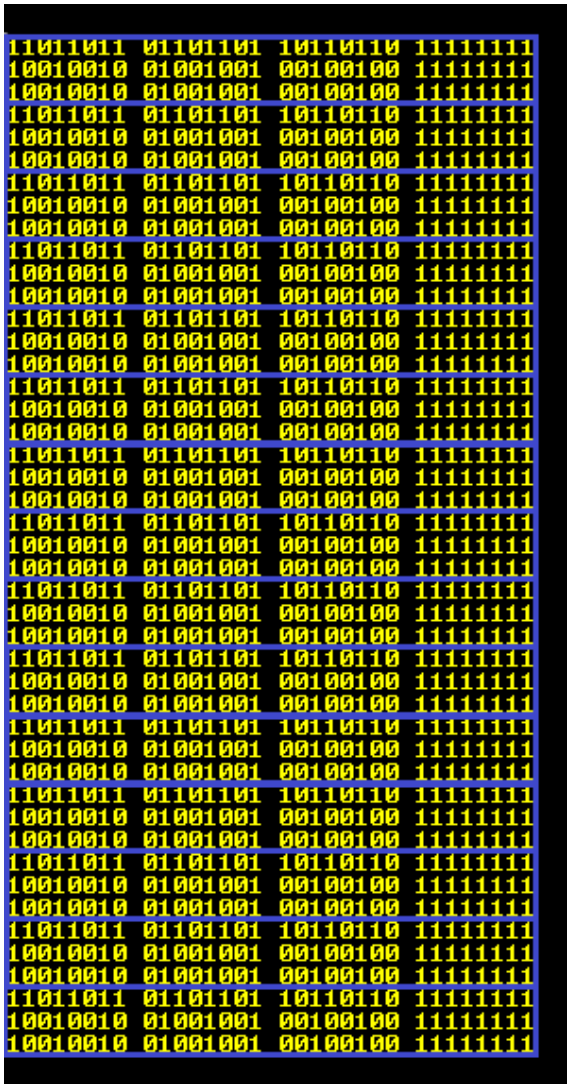
Efter at have testet lidt opdager vi at LED*erne virker som de skal, men kun på de første 29 LED'er... Herunder bruger vi driveren til at udskrive fuld rød til alle LED'erne. Der går tydeligt vis noget helt galt.



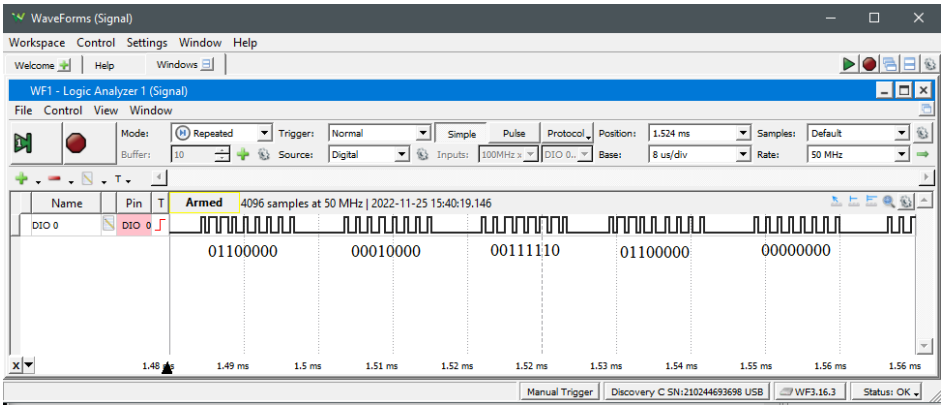
Vi kan udskrive frambufferen ved de array der bliver sendt til LED'erne lige omkring LED 29 for at se hvad der sker. Det ligner dog at der ikke er noget galt med framebufferen. Alt ws2812 koden ser fin og ens ud, her prøver jeg at udskrive 100 % grøn til LED'erne.

Så vi må kigge på selve signalet der bliver sendt med et oscilloskop. Det ses herunder og der er noget galt igen. Det er ikke det samme som der står i framebufferen som SPI skriver ud til datalinjen.

Jeg fandt ud af at jeg havde et for-loop i min updaterings funktion der overflowede og at det var derfor der kom dårlige værdier ud efter de 29 LED'er. Efter at ændre det virker LED'erne som forventet.



Figur 8: ws2812 framebuffer for LED 25-39



Figur 9: Oscilloskop output fra ws2812 datalinjen, med dårlig data

1.5 PSoC kommunikation

Main filen er ret lang og indeholder meget redundant kode. Det er da vi har implimenteret en intern UART til at debugge. Vi vil kun vise det der er relavant for projektet. Til at starte med initialisere vi kommunikations modulet og så ws2812 driveren.

```
uint16_t numberOfPixels = 200;
uint8_t ringSize = 14;

CyGlobalIntEnable;

uint8_t sendedLength=10;
uint8_t recievedLength=10;
uint8_t recievedDataFromUartBuff[sendedLength];
uint8_t recieveComplete=0;
uint8_t *ptrRC=&recieveComplete;

comModulInit(recievedDataFromUartBuff,recievedLength, sendedLength, ptrRC);
ws2812_init(numberOfPixels);
```

Nu kan man skrive til alle LED'erne på denne måde.

```
ws2812_setMultiRGB(0, numberOfPixels, 0xFF, 0xFF, 0xFF);
ws2812_update();
```

Her skriver vi fra LED 0 til den sidste LED med farvekoden #FFFFFF. Altså helt hvidt.

For at integrere med spillogikken laver vi nogle funktioner der kan blive kaldt. Herunder ses de funktioner som LED driveren skal kunne køre. Jeg vil kun vise to af dem da de alle er meget ens. Source koden ligger i bilag [1, ws2812_driver].

Funktion	Beskrivelse	Argumenter
0 - 3	Set ring	RGB
5	Set LED strip farve	stop led, RGB
6	Start animation	ingen
7	Sluk alle LED ringe	ingen
8	Sluk LED strip	ingen

Det eneste tidspunkt hvor vi skal svare spillogikken er når animationen er færdig.

Vi definere først størrelsen af LED ringene, i vores tilfælde bruger vi 14 LED'er per ring. For at få kommunikations modulet til at virke definere vi det første array værdi som funktions kaldet. Derefter har vi aftalt at funktion 4-6 er farve koder R, G og B.

```
uint8_t ringSize = 14;

uint8_t function = recievedDataFromUartBuff[0];

uint8_t pixelColorR = recievedDataFromUartBuff[4];
uint8_t pixelColorG = recievedDataFromUartBuff[5];
uint8_t pixelColorB = recievedDataFromUartBuff[6];
```

Herunder ses funktion 0 der styrre den første LED ring.

```
if (function == 0){
    ws2812_setMultiRGB
  ...(ringSize*0, ringSize*1-1, pixelColorR, pixelColorG, pixelColorB);
    ws2812_update();
}
```

Og her ses funktion 5 der styrre LED strip'en. Vi vælger start som den sidste LED i ringene og slut LED'en som det argument der bliver givet gennem kommunikations modulets 3 array plads.

```
if (function == 5){
    pixelStripStop = recievedDataFromUartBuff[3];

    ws2812_setMultiRGB
  ...(ringSize*4, ringSize*4+pixelStripStop, pixelColorR, pixelColorG, pixelColorB);
    ws2812_update();
}
```

1.6 LED animationer

For at animere LED'erne køre jeg et loop der med et delay opdatere både framebufferen og LED'erne. Herunder ses hvordan jeg gør. Jeg bruger en sinus og cosinus til at lave animation med 180° forskudt fase.

```
for (uint32_t k = 0; k < 400; k++){
    for (double i = 0; i < ringSize; i++){

        uint8_t wavesin = (uint8_t)(pow(sin((i*0.1+1)+keyframe*(3.14/180.00)),2)*127);
        uint8_t wavecos = (uint8_t)(pow(cos((i*0.1+1)+keyframe*(3.14/180.00)),2)*127);

        ws2812_setRGB(i+ringSize, wavesin, 0x00, wavecos);

        keyframe+=0.1;
    }

    ws2812_update();
    CyDelay(3);
}
```

Da sinus funktionens output går fra -1 til 1 og vi vil have en farve kode fra 0 til 255, bruger vi først pow, så vi får 0 til 2. Derefter ganger vi med 127, så kommer resultatet til at være fra 0 til 254. Det er godt nok.

Det inderste for-loop's indeks går fra 0 til størrelsen af ringene og så ganger jeg det på sin og cos funktionerne og bruger bruger samme indeks til at sætte LED ring indekset. Dermed får hver LED deres egen lidt forskudte sin og cos vinkel og på den måde for man en glidende overgang mellem farver der langsomt skifter.

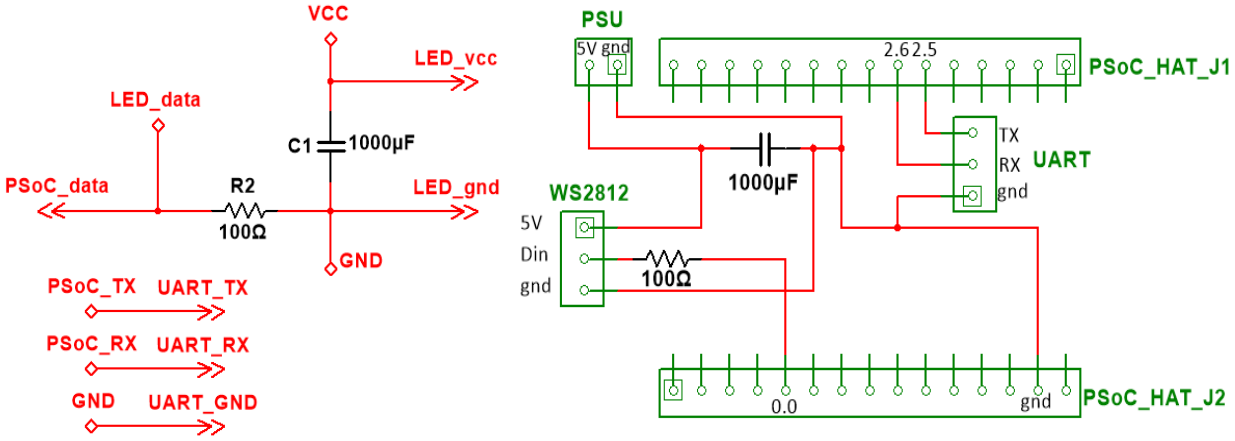
1.7 LED kommunikation optimering

I stedet for at have 4 funktioner med hver deres farvekode, kunne vi have en funktion med en byte. De første 4 bit ville enkeltvis bestemme hvilke af de 4 ringe der skal ændres og de sidste 4 bit kunne vælge imellem 16 forskellige farver. Det ville gøre så vi kunne ændre flere ringe på en gang. I virkeligheden bruger vores spil dog kun 4 farver til ringene; rød, grøn, gul og slukket. Hvis vi i stedet bruger 3 bit til 8 farver og så den sidste bit til at starte animationen, kan vi lave alle nuværende ring funktioner med denne ene byte. Det er klart at der skal markante ændringer til både LED driveren og spillogikken, hvilket er hvorfor vi ikke har valgt at lave nogen af disse ændringer. Lignende forbedringer kunne også laves til LED strippen.

bit	funktion
0	Ring
1	Ring
2	Ring
3	Ring
4	Farve
5	Farve
6	Farve
7	Animation

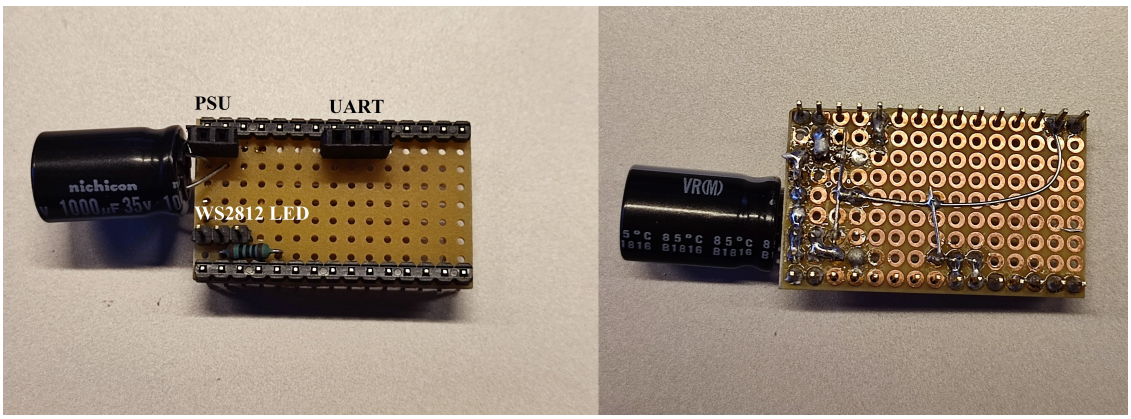
1.8 PSoC LED hat

For at sende data og strøm til LED strippen valgte vi at bygge en hat til PSoC'en. Det gør det nemmere at forbinde LED'en og strøm uden at kortslutte noget ved en fejl. Derudover kan UART kommunikationen til spillogikken også sættes på med det samme. Herunder ses designet. Det er et meget simpelt design med en enkelt modstand for at beskytte PSoC'en og en 1000 µF kondensator for at beskytte LED'erne. Det kan ses på figuren hvilke PIN på PSoC'en vi bruger, men det er ikke så vigtigt da de selvfølgeligt nemt kan ændres i koden.

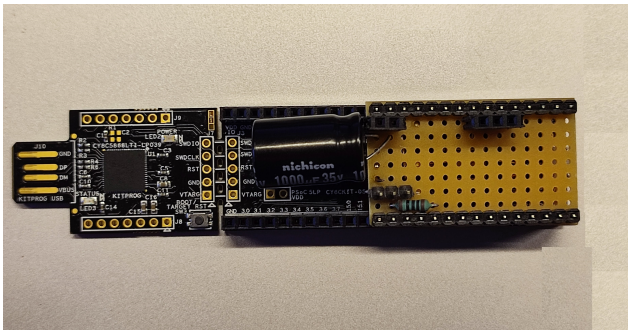


(a) PSoC hat elektrisk design

(b) PSoC hat pins



Figur 11: PSoC hat



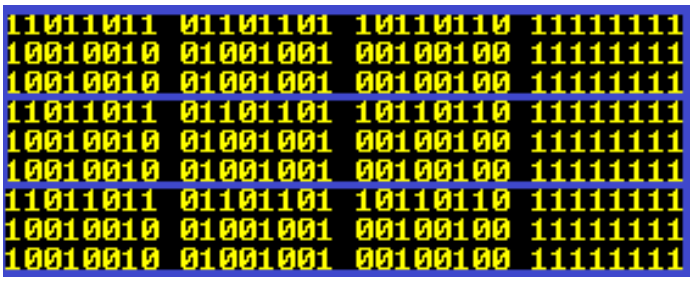
Figur 12: PSoC med hat på

2 Lys modultest

Normalt når man tester noget som ingeniør kan man bruge en LED for at se om det rigtige signal kommer igennem eller om der overhovedet er strøm. Vi er så heldige at det præcist er det vi skal teste. Derfor vælger vi at bruge en UART på computeren der skal opdatere LED drivers framebuffer og selve LED'erne for at se om de gør som forventet.

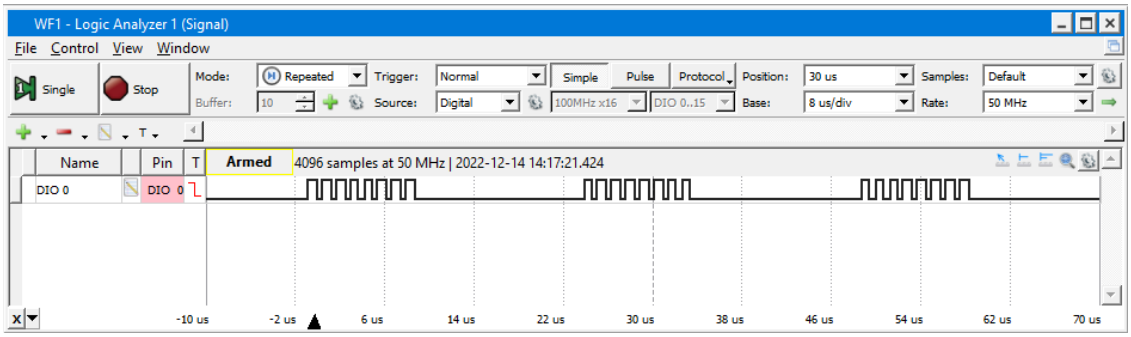
2.1 Målinger (UART & oscilloskop)

Først vil vi lige check at det array vi sender med SPI indeholder den korrekte data. Jeg udskriver den direkte til en konsol med den indbyggede UART. Da koden fungere ved at sende enten 110 eller 100 for hhv. 1 og 0, kan man se på figuren herunder at framebufferen er sat op korrekt. (Her skriver jeg 100% grøn til LED'erne)



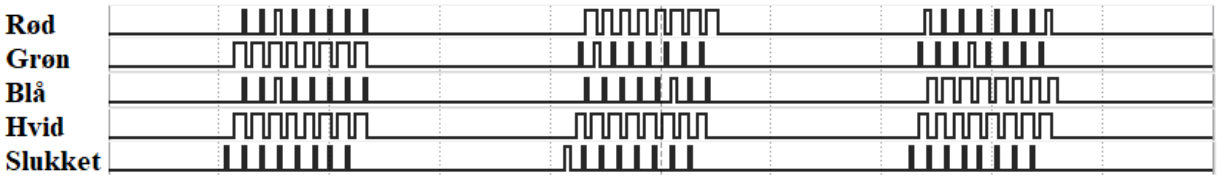
Figur 13: ws2812 framebuffer for 3 LED'er(GRB)

Med et oscilloskop kan vi også se om selve signalet der bliver sendt ud af SPI ser rigtigt ud. Herunder ses en enkelt måling når man sender 100% RGB. Læg mærke til at der ved hver data overførsel bliver sendt 24 bit, selvom det ligner 8. Da det er 3 overførsler med 8 bit hver med SPI var lidt nervøse for at der ville være et delay imellem hver overførsel, men det kan vi se med oscilloskopet at der ikke er.



Figur 14: Oscilloskop skærmklip

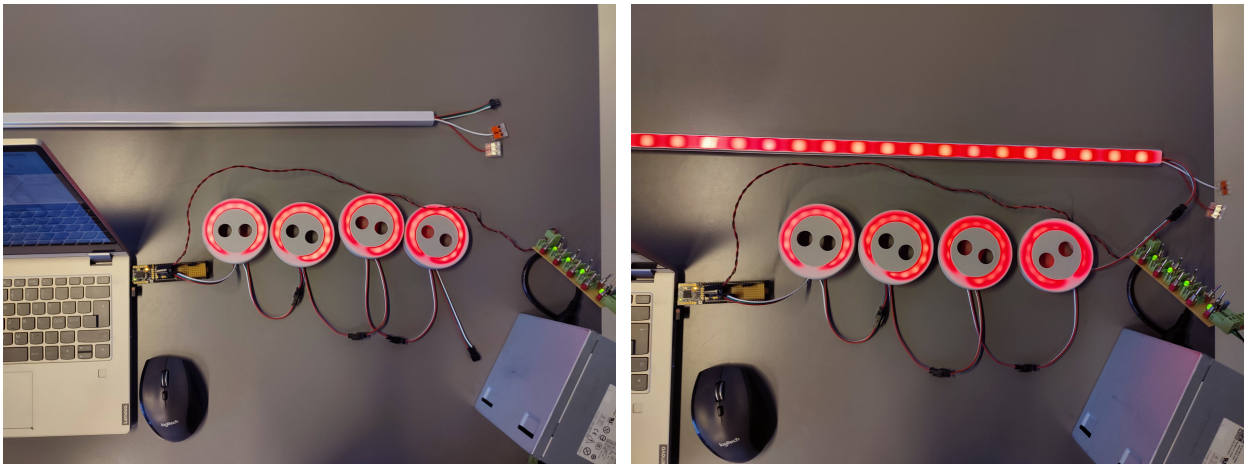
Herunder ses nogle flere målinger. Det forventede bliver sendt ud med SPI.
Igen så arbejder ws2812 med GRB og ikke RGB.



Figur 15: Oscilloskop output fra ws2812 datalinje

2.2 Lys test

Vi tester LED'erne og alt virker som det skal. De er sluttet til vores strømforsyning da de trækker meget strøm.

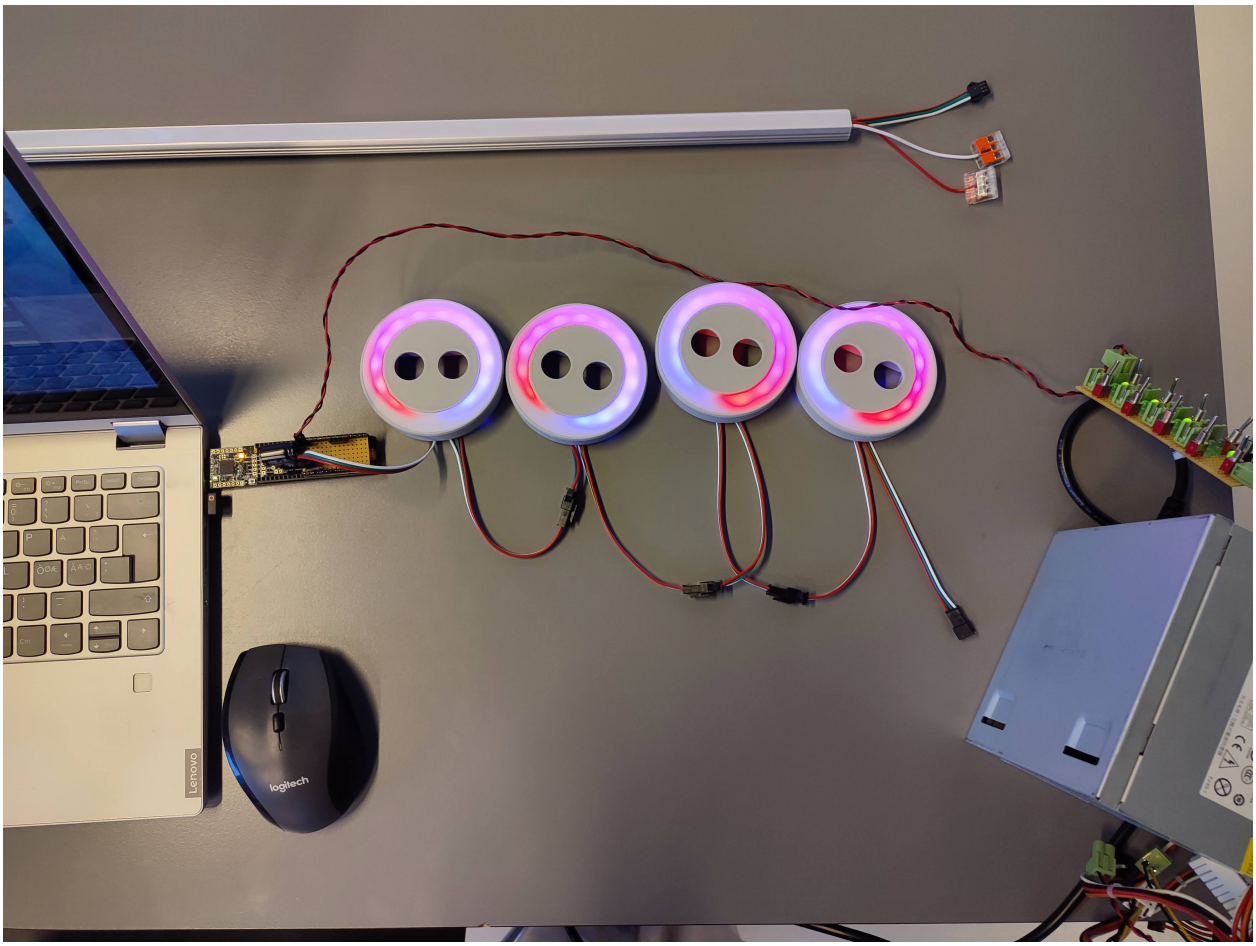


(a) Oplyste LED ringe

Figur 16: LED lys tests

(b) Oplyst LED strip

Herunder ses en enkelt udsnit af animationen. Man kan se at de skifter fra rød ti blå i en glidende overgang.



Figur 17: Oplyste LED ringe med animation

Litteraturliste

[1] Bilag O -. „Source kode“.

[2] Alan Hawse. *PSoC 6, DMA & WS2812 LEDs*. Last visited: 12-12-2022. iotexpert. `https://iotexpert.com/psoc-6-dma-ws2812-leds/`.

[3] Worldsemi. *WS2812 Intelligent control LED integrated light source*. Last visited: 12-12-2022. worldsemi. `https://cdn-shop.adafruit.com/datasheets/WS2812.pdf`.