

# Bilag M - PSoC sensorkontrol

Jonas Gjørup Eriksen

1. juni 2023

# Indhold

<b>1 Indledning</b>	<b>2</b>
<b>2 Design valg</b>	<b>2</b>
2.1 Valg af mikrocontroller . . . . .	2
2.1.1 PSoC og Attiny . . . . .	2
2.1.2 Udelukkende PSoC . . . . .	2
2.1.3 Hvad valgte vi . . . . .	3
2.2 Valg af ADC . . . . .	3
2.2.1 SAR konverter . . . . .	3
2.2.2 Sigma-Delta konverter . . . . .	4
2.2.3 Hvad valgte vi . . . . .	4
2.3 Valg at datatype til videresending . . . . .	5
2.3.1 Et array af char . . . . .	5
2.3.2 Processeret data som float . . . . .	5
2.3.3 Hvad valgte vi . . . . .	5
<b>3 Implementering - færdige version</b>	<b>5</b>
3.1 ADC . . . . .	5
3.1.1 måling af data . . . . .	7
3.2 Pumpestyring . . . . .	8
3.3 Konvertering af data . . . . .	9
3.3.1 Lyssensor . . . . .	9
3.3.2 fugtighedssensor . . . . .	9
3.3.3 Vandniveau sensor . . . . .	10
3.3.4 Vægtsensor . . . . .	10
3.4 Sammensætning af WIFI og PSoC dele . . . . .	10
<b>4 Modultest</b>	<b>12</b>
4.1 test af ADC . . . . .	12

# 1 Indledning

I planten foretages der mange målinger fra forskellige sensorer. Disse sensorens signaler skal behandles, så vores RPI kan logge dataen. Der er derfor nogle vigtige valg, der skal træffes under udviklingen, som omfatter:

- Valg af mikrocontroller
- Valg af ADC
- Valg af datatype til videresending

## 2 Design valg

### 2.1 Valg af mikrocontroller

Ved valg af mikrocontroller er der primært to forskellige fremgangsmåder, som vi kan vælge imellem.

Den første metode er at bruge en PSoC i samarbejde med en ATtiny45, og den anden metode er udelukkende at bruge en PSoC.

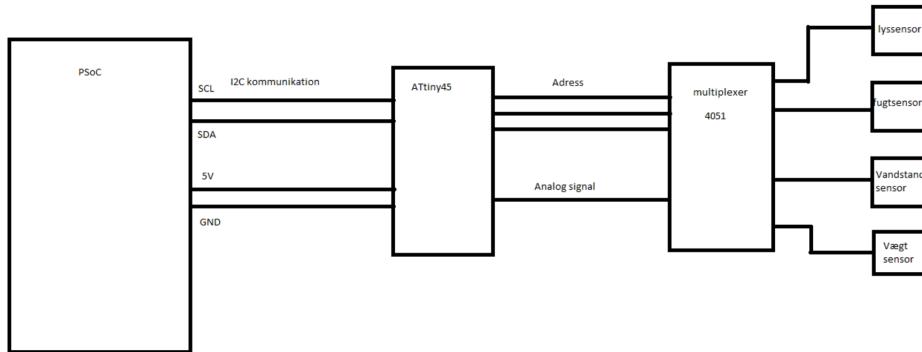
#### 2.1.1 PSoC og Attiny

Grundidéen bag dette koncept er at opdele opgaverne for vores mikrocontroller på to forskellige chips. Det, der skal ske, er en konvertering af analoge signaler til digitale samt en behandling af disse data, så de kan sendes videre til vores RPI.

I denne løsning ville konverteringen af data finde sted i ATtiny45 mikrocontrolleren, hvor den ved hjælp af I2C-kommunikation ville sende dataene videre til PSoC'en. PSoC'en ville agere som master, mens ATtiny45 ville agere som slave.

Der er både fordele og ulemper ved denne løsningsmetode. Fordelene ved dette er, at ATtiny45 er en meget lille chip, så det ville være nemmere at placere den tæt på vores sensorer, hvor den kan sende signalet til et sted med mere plads til vores PSoC. Ulemperne ved dette system er, at det tilføjer et ekstra kompleksitetsniveau ved at oprette denne kommunikation mellem PSoC og ATtiny, hvor der kan opstå fejl. Pludselig skal der kommunikeres mellem 3 mikrocontrollere i stedet for 2. En anden ulempe ved denne metode er, at vi bliver tvunget til at bruge en SAR-konverter, men mere om det senere.

En sketch af hvordan denne model ville komme til at se ud kan ses på figur 1:



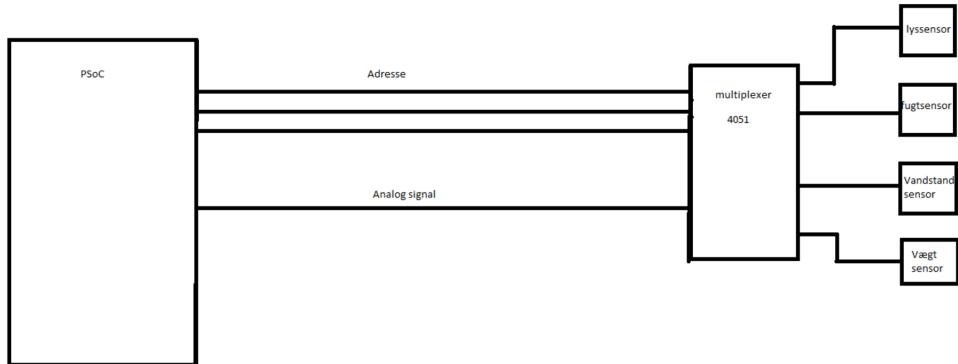
Figur 1: Sketch af kredsløb for ATtiny45 og PSoC med multiplexer

#### 2.1.2 Udelukkende PSoC

I modsætning til ideen om at dele det op, kan man også vælge at have alt på samme chip. Dette vil betyde, at det analoge signal kommer direkte ind på vores PSoC, og alt vil ske inde i den. Fordelene ved dette er, at vi har større valgmuligheder i beslutningen om, hvilke ADC der er bedst,

da PSoC'en kan implementere flere forskellige ADC'er. En anden fordel, som nævnt tidligere, er at der vil være færre kommunikationsveje, nemlig kun imellem PSoC og RPI. Der er dog bare det, at chippen fylder meget mere, og der vil skulle laves mere plads til den inde i potten.

En sketch af denne løsning kan ses på figur 2:



Figur 2: Sketch af kredsløb for udelukkende PSoC med multiplexer

### 2.1.3 Hvad valgte vi

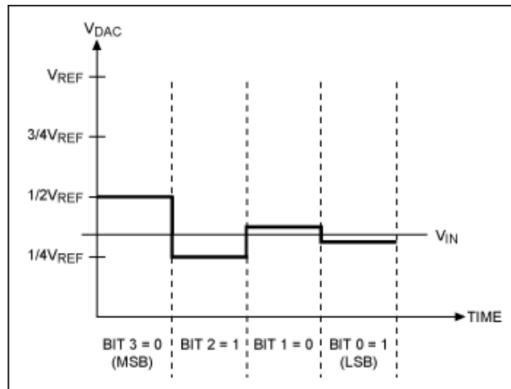
Originalt valgte vi den løsningsmetode, der involverer både PSoC og ATtiny45, primært af uddannelsesmæssige årsager. Vi ønskede at lære at arbejde med mindre mikrocontrollere og få kendskab til programmering af dem. Dog stødte vi på problemer under udviklingen af i2c-kommunikationen mellem PSoC og ATtiny45, hvilket gjorde at vi blev nødsaget til at vælge løsningsmetoden med udelukkende en PSoC som mikrocontroller.

## 2.2 Valg af ADC

Vores ADC er en rigtig væsentlig del af vores produkt. Vores ADC er den der afgør både præcision og hastighed på aflæsningen af sensorerne. Vi havde primært 2 forskellige typer ADC i diskussionerne om hvad vi skulle vælge. Dette er ADC'erne der bliver kaldt SAR og sigma-delta.

### 2.2.1 SAR konverter

SAR står for ”successive-approximation-register” hvilket betyder at en SAR konverter løbende approximere sig tættere på svaret. For at forklare dette dybere kan man se på figur 3.



Figur 3: Demonstration af SAR ADC [1]

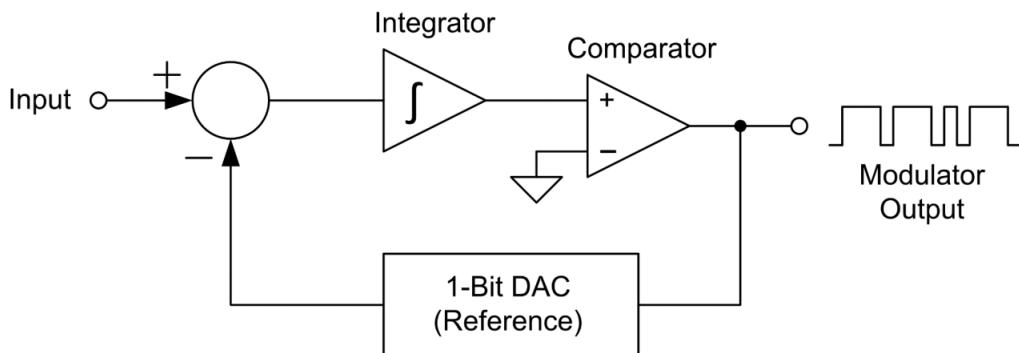
Her kan man observere, hvordan SAR ADC kontinuerligt nærmer sig den korrekte værdi ved at sætte og nulstille bit. Først sætter den det mest signifikante bit og kontrollerer, om det analoge signal er over eller under denne værdi. Hvis signalet er over, forbliver bit'et sat til 1, ellers sættes

det til 0. Derefter går ADC'en videre til næste bit og bruger den tidligere gemte værdi til at fortsætte processen. Antallet af bits repræsenterer opløsningen, dvs. hvor mange gange denne proces gentages. Så i eksemplet på figur 3 kan man se, at opløsningen er 4 bit.

Det gode ved SAR er, at man altid er sikker på, hvor lang tid der går mellem hvert modtagne signal, da ADC'en altid gennemgår den samme proces. SAR er også i stand til at aflæse signaler, der er i bevægelse eller forandring. Dog er der også nogle ulemper ved SAR. En af ulemperne ved SAR ADC'en er, at selvom den tilbyder en relativt høj opløsning, når den ikke tæt på den næste ADC, vi vil tale om, nemlig sigma-delta ADC'en.

### 2.2.2 Sigma-Delta konverter

Sigma-Delta ADC fungerer på en meget anderledes måde end SAR ADC'en, men er på en måde baseret på de samme principper. Ved sigma-delta ADC har man et output-signal, der skifter mellem 1 og 0, og ved at tælle antallet af 1'ere kan man beregne den tilsvarende spænding på inputtet. For at forklare dette nærmere, henvises til figur 4.



Figur 4: Demonstration af sigma-delta ADC[1]

Måden, en sigma-delta ADC fungerer på, er ved at have et input, der tilføjes integratoren, som starter ved 0. Hvis inputtet er 1 V, vil integratoren stige til 1. Da denne værdi er over nul, bliver komparatoren sat høj, hvilket betyder, at vi trækker reference-spændingen fra. Hvis vi i dette eksempel har en reference-spænding på 2,5 V, vil dette blive trukket fra inputtet, hvilket giver -1,5 V. Dette bliver yderligere trukket fra komparatoren, der bliver -0,5 V. Da dette er under nul, vil outputtet være 0, og dette betyder, at reference-spændingen bliver lagt til igen. Denne proces fortsætter, indtil vi har en lang strøm af tal.

Med disse værdier, dvs. input på 1 V og reference på +/-2,5 V, ville outputtet være 10111011011011101101. Her udgør 70% af tallene 1, og måden, man kommer frem til 1 V ud fra dette, er ved at tage 70% af 5 V, som er spændingsintervallet fra -reference til +reference, hvilket giver 1,5 V, og trække dette fra +reference.

Denne ADC er lidt mere kompleks, men den fungerer rigtig godt til signaler, der er meget stillestående, da den tilbyder en meget høj opløsning, selvom den ikke er særlig hurtig sammenlignet med andre ADC'er.

### 2.2.3 Hvad valgte vi

Til at starte med valgte vi SAR ADC'en, da den var den eneste tilgængelige i ATtiny45. Men efter jeg ikke kunne få den til at fungere, skiftede vi til sigma-delta ADC'en, som fungerer perfekt til vores applikation. For vores signaler er det afgørende, at vi kan få så præcise målinger som muligt, f.eks. for at sikre præcisionen af vægten. Samtidig er vores signaler meget stillestående, hvilket er ideelt for sigma-delta ADC'en.

## 2.3 Valg at datatype til videresending

Efter vi har modtaget og aflæst vores data igennem vores ADC. Skal vi have konverteret det for at kunne sende det videre igennem wifi kommunikationen. Meget af dette handler om hvor vi vil omdanne den rå data til værdier der giver mening for brugeren.

### 2.3.1 Et array af char

Denne løsning er baseret på ideen om at sende den samme bit størrelse hver gang der bliver sendt data. Dette betyder at den rå data vil blive konverteret til 2 char som er 16 bit og lagt i et char array med de andre sensorer. Dette betyder at for de fire sensorer vil der hver gang blive sendt 64 bit. Dette gør det nemmere at sende dataene gennem internettet da modtageren altid ved hvor meget data der kommer tilbage. Dog gør det at vi ikke er i stand til på PSoCen at konvertere dataene til noget brugbart for eksempel at vægtens data er i gram.

### 2.3.2 Processeret data som float

Denne metode fungere lidt modsat af den tidligere. Her omdanner vi værdierne på PSoC siden og sender den som en float igennem internet modulet. Fordelen ved dette er at der er færre personer indblandet i processeringen af data da det hele bare kan laves fra PSoCen men derimod bliver det sværere at sende signalet igennem internettet eftersom datalængderne kan variere.

### 2.3.3 Hvad valgte vi

Til dette valgte vi at gå med den metode der nemmest tillod os at sende data igennem internet-modulet. Denne metode var at konvertere dataene til et array af chars også behandle værdierne på RPI siden af kredsløbet.

## 3 Implementering - færdige version

Til udviklingen af denne del af produktet er der blevet brugt programmet PSoC creator 4.4 lavet af infineon. Dette program er lavet til at understøtte PSoCen så det var et nemt valg at bruge dette program.

### 3.1 ADC

I PSoC creator er der indbygget en sigma-delta ADC som jeg har valgt at bruge. Når man bruger denne indbyggede sigma-delta ADC er der nogle ting man skal tage højde for.

Det første er hvilket mode man vi kører i. Der er 4 forskellige modes at vælge mellem:

- Single Sample mode

Single Sample mode tillader at sætte en konvertering i kø. Hvis der anmodes om en prøve enten ved at kalde StartConvert() funktionen eller ved at bruge inputtet "soc" før den nuværende konvertering er fuldført, vil en anden konvertering automatisk blive startet efter at den nuværende konvertering er fuldført.

- Multi-sample mode

Multi-sample mode optager enkelte prøver efter hinanden og nulstiller sig selv og modulatoren mellem hver prøve automatisk. Denne tilstand er nyttig, når indgangen skifter mellem flere signaler. Filterne tømmes mellem hver prøve, så tidlige prøver ikke påvirker den aktuelle konvertering.

- Continous mode

Continuous sample mode fungerer som en normal delta-sigma-konverter. Brug denne tilstand, når du mäter et enkelt indgangssignal. Der er en forsinkelse på tre konverteringstider, før det første konverteringsresultat er tilgængeligt. Dette er den tid, der er nødvendig for at klargøre det interne filter. Efter det første resultat vil en konvertering være tilgængelig ved den valgte samplingsfrekvens.

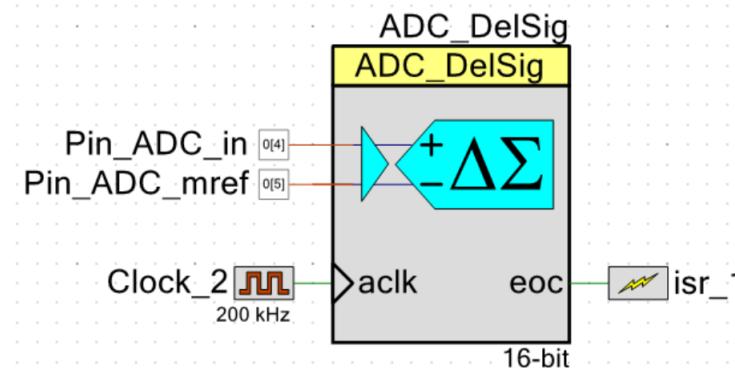
- Multi-sample (turbo) mode

Multi-Sample (Turbo) tilstand fungerer på samme måde som Multi-Sample tilstand for opløsninger fra 8 til 16 bit. For opløsninger fra 17 til 20 bit er ydelsen i denne tilstand cirka fire gange hurtigere end Multi-Sample tilstand.

Disse informationer er fundet i datasheet for sigma-delta ADCen [2]

Her valgte vi at arbejde med 16 bit hvilket betyder at multi-sample mode ville fungere rigtig godt til vores applikation. Vi ville ikke kunne bruge simple og continuous mode på samme måde da vi har flere forskellige sensorer vi skal måle fra og vi skal tage flere målinger lige i streg.

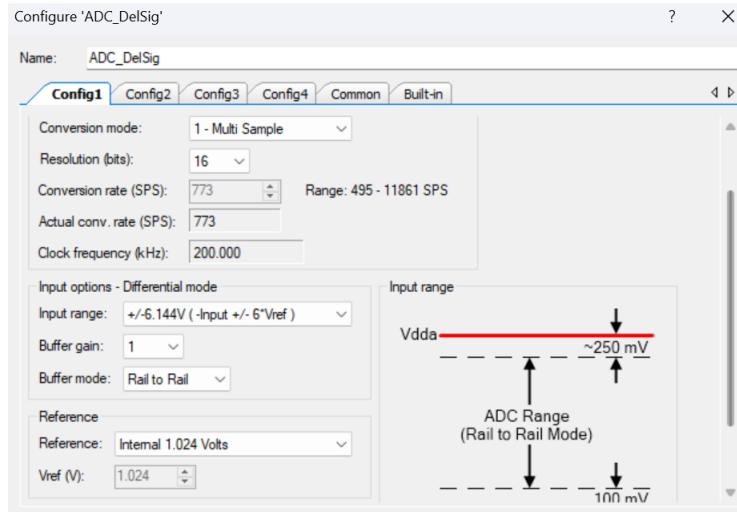
Med dette kom vores top level design til at se således ud: Her er vores Pin\_ADC\_in vores ana-



Figur 5: toplevel design af sigma-delta ADC

log signal og Pin\_ADC\_mref sat til ground. Inde i denne blok bliver indstillingerne sat således.

Her er det vigtigste at se at input range er mellem +/-6.144V og at det er multi-sample mo-



Figur 6: Indstillinger inde i sigma-delta ADCen

de vi kører med.

Nu med det hele sat op kommer vi så til selve koden, den første vi skal gøre er vores setup. Her bruger jeg funktionen ADC\_start(). Denne funktion konfigurerer og tænder ADCen men starter ikke en konversion. Herefter er det at bruge vores ADC. Måden dette fungere på kan ses på figur 7.

```

if (ADC_DelSig_IsEndConversion(ADC_DelSig_WAIT_FOR_RESULT))
{
    buffer[count] = ADC_DelSig_GetResult16();
    count++;
}

```

Figur 7: Brug af sigma-delta ADC i kode

Her kan man se at jeg bruger 2 funktioner til at få det til at fungere. Den første er ADC\_IsEndConversion(ADC\_WAIT\_FOR\_RESULT). Denne funktion giver brugeren 2 valg alt efter hvilken parameter man giver den. Man kan vælge at den omgående sender en status på konverteringen tilbage og man kan vælge at den venter til konverteringen er færdig for at få svaret tilbage. Her har vi brugt parameteren ADC\_WAIT\_FOR\_RESULT som betyder at den venter til konverteringen er færdig og sender status tilbage. Dette gør at vi ikke misser nogle konverteringer.

Efter dette kommer vi til selve målingen hvor jeg bruger funktionen ADC\_GetResult16(). Denne funktion starter en konvertering og venter til denne konvertering er slut hvorefter den returnerer en 16 bit værdi som vi kan gemme i en uint16\_t.

### 3.1.1 måling af data

Nu med implementeringen af vores sigma-delta ADC er vi i stand til at skaffe den nødvendige data. Men for at holde styr på hvilke sensor vi læser fra og have en mere præcis observering er der nogle ting som der mangler at blive implementeret.

Man kan se på figur 2 kan man se at kredsløbet bruger en multiplexer. Måden denne fungere på er ved at den modtager en adresse hvorefter den sender sensorens på den pågældende adresses analoge signal igennem. Til dette har jeg valgt at lave en funktion til de pågældende 4 sensorer hvor jeg inde i disse funktioner kalder den korrekte adresse og kalder funktionen readADC. En af disse funktioner ser således ud:

```

28  uint16_t readLight()
29 {
30
31     // address for light sensor
32     Pin_bit0_Write(1);
33     Pin_bit1_Write(0);
34     Pin_bit2_Write(1);
35
36     return readADC();
37
38 }

```

Figur 8: Kode eksempel fra readLight() funktionen

Her kan man se at vores lyssensor har adressen 0b101 eller 5. Udover dette har fugtighedssensoren adressen 0b110 eller 6. Vandstand sensoren adresse 0b000 eller 0 og vægtsensoren 0b010 eller 2.

Det næste der så sker er inde i funktionen readADC().

```

71  uint16_t readADC()
72  {
73      uint8_t BUFFER_SIZE = 200;
74      uint16_t buffer[BUFFER_SIZE];
75      uint32_t count = 0;
76      int sum = 0;
77      //static char outputBuffer[256];
78
79      ADC_Delsig_StartConvert();
80      while (count < BUFFER_SIZE)
81      {
82          if (ADC_Delsig_IsEndConversion(ADC_Delsig_WAIT_FOR_RESULT))
83          {
84              buffer[count] = ADC_Delsig_GetResult16();
85              count++;
86          }
87      }
88      ADC_Delsig_StopConvert();
89      for (int i = 0; i<BUFFER_SIZE; i++)
90      {
91          sum = sum + buffer[i];
92      }
93      uint16_t gennemsnit = sum/BUFFER_SIZE;
94
95
96      if (gennemsnit > 50000)
97          gennemsnit = 0;
98      return gennemsnit;
99  }

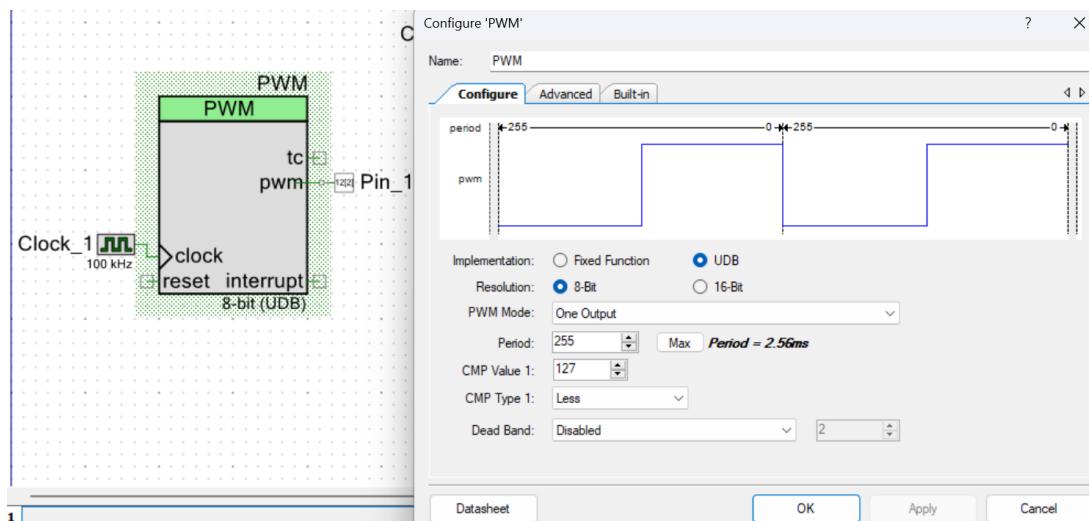
```

Figur 9: kode eksempel fra readADC() funktionen

Ved denne funktion har det været vigtigt for os at sikre den data vi læser så meget som muligt. Derfor har jeg valgt at implementere en buffer hvor vi gemmer 200 værdier i den og finder gennemsnittet af disse værdier. Dette sikrer os imod tilfældige spikes og støjsignaler der ville kunne komme på det analoge input.

### 3.2 Pumpestyring

Måden vi vander vores planter er ved at tænde en pumpe der er styret af en motor. Måden denne pumpe skal tendes er ved hjælp af at sende et PWM signal. Måden vi har valgt at implementere vores PWM signal i vores top level design kan ses på figur 10



Figur 10: Implementering af PWM signal i top level design

Her der det vigtigste at poientere at vi har en clock på 100 kHz hvilket og at vi har en periode på 255 clock cycles.

Måden vi så bruger PWM signalet i koden er ved at skrive dette:

```
PWM_Start();  
PWM_WriteCompare(255);  
CyDelay(10000);  
PWM_Stop();
```

Figur 11: Brug af PWM signal i kode

Den første funktion jeg bruger er PWM\_start(). Denne funktion er designet til at starte komponentens drift. PWM\_Start() sætter variablen initVar, kalder PWM\_Init-funktionen og derefter PWM\_Enable-funktionen.

Herefter bruger vi funktionen PWM\_WriteCompare(uint8). Det er denne funktion der bestemmer duty cycle på vores PWM signal. I dette eksempel sætter vi vores duty cycle til at være 100% at kører pumpen så hurtigt som muligt.

Herefter kommer der et delay der styrer hvor lang tid pumpen skal kører. Her har vi besluttet at pumpen skal pumpe 10 ml vand og igennem test er det blevet konkluderet at pumpen med 100% duty cycle pumper 1 ml i sekundet hvilket betyder at den skal kører i 10 sekunder. Hvorefter vi stopper PWM signalet ved hjælp af funktionen PWM\_Stop().

Informationen om disse funktioner er fundet i datasheetet for PWM modulet som infineon har lavet [3]

Men hvornår skal pumpen så kører. Pumpen skal kører i det tilfælde at vores fugtighedsmåler måler en værdi som vi vurdere til at være tør jord. Denne værdi er fundet ved hjælp at modultesten for fugtighedsensoren hvor de har defineret de forskellige intervaller. (((((())))) her mangler der noget tekst om hvilket interval

### 3.3 Konvertering af data

Til dette kredsløb har vi 4 sensorer der hvis man ikke gør noget med de signaler man modtager er ret ubrugelige. Dette betyder at vi skal have lavet nogle formler og intervaller som kan bruges og fortolkes af mennesker således loggen giver bedre mening.

Her er det værd at pointere at meget af arbejdet er lavet ved de forskellige elektroniske kredsløbers modultest. Her bliver der blot forklaret hvordan de spændinger der kommer ind på vores sigma-delta ADC bliver håndteret.

#### 3.3.1 Lyssensor

Ved lyssensoren vil vi gerne have en værdi der kommer ud i procent. Dette betyder at når lyssensoren modtager maksimalt lys i forhold til hvad den kan klare skal den skrive 100% og når der er helt mørk skulle den gerne skrive 0%.

Ved helt mørke målte vi en værdi på 596 og ved 100% lys målte vi en værdi på 28454. Herefter er det bare at konvertere dette til en ligning  $(x - 596) / 278.58$ .

#### 3.3.2 fugtighedssensor

Ved fugtighedssensoren vi vi gerne have den inddelt i forskellige intervaller. Disse intervaller skal være en indikator til de forskellige niveauer: tør, optimal, våd og druknet. Disse værdier endte med at være 18125, 17365, 17000, 0 efter konvertering med vores sigma-delta ADC. Her har vi så valgt at indele dem i intervallerne 0%, 33%, 66% og 100% for at man kan se på loggen hvad fugtigheden er.

### 3.3.3 Vandniveau sensor

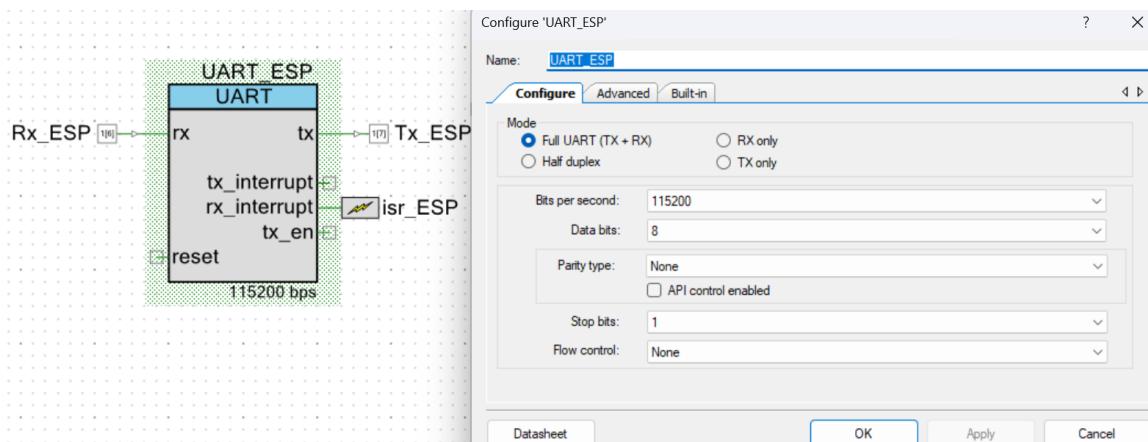
Ved vandniveau har vi også valgt at inddøle de forskellige værdier i intervaller. Har har vi så intervaller er indikere forskellige mængder vand målt fra sensoren. Så vi har en værdi for 0 ml, 100 ml, 200 ml, 300 ml, 400 ml og 500 ml hvor 500 ml er det maksimale der kan være i vores vandtank. Disse værdier endte med at være 20835, 20200, 18700, 18030, 17740 og 17230 hvor 20200 er for 100 ml. Her er det vigtigt at pointere at ved alle intervallerne bliver der rundet ned. så hvis der er over 20200 vil der altid stå 0 ml og hvis nu man havde en værdi på 18701 vil systemet sige 100 ml.

### 3.3.4 Vægtsensor

Meget af den data som der bliver behandlet her kan findes i bilag H for vægten. Men det grundlæggende er at der bliver målt en flere prædefinerede vægte som bliver plottet og fundet en tendenslinje for. Vi har valgt at måle værdierne: 0g, 1000g 2000g, 3000g, 4000g og 4500g. Disse vægte har givet os værdierne 1860, 5580, 9430, 13327, 17221 og 19101. Med dette ender vi med at få ligningen: vægt =  $0,2599x - 465,08$ .

## 3.4 Sammensætning af WIFI og PSoC dele

For at få ESP modulet til at fungere med vores PSoC skulle vi opsætte en UART da det er den måde man kommunikere med den på. Måden denne uart bliver sat op i toplevel design ser følgende ud:



Figur 12: UART for ESP i toplevel design

her kan man se at vi har opstat en rx(reciever) og en tx(transmitter). Det er disse vi bruger til at modtage data fra ESPen og sende data videre ud igennem ESPen. Derudover har vi også opsat en isr (interrupt service rutine) som gør at vi kan kalde et interrupt hver gang vi modtager data fra ESPen. Udenfor dette kan man se at vi kører med en baud rate på 115200 og vi kører med data bits som er 8 lange.

I main er det første vi skal have gjort at initiere vores uart samt initiere vores isr. Dette gøres på følgende måde:

```
88 CyGlobalIntEnable; /* Enable global interrupts. */
89
90 UART_ESP_Start();
  isr_ESP_StartEx(ISR_ESP_Handler);
```

Figur 13: initiering af UART i main kode

Her bruger vi CyGlobalIntEnable som muliggøre interrupts. Herefter bruger vi funktionen UART\_Start() som starter komponentens drift. UART\_Start() sætter initVar-variablen, kalder funktionen UART\_Init() og derefter funktionen UART\_Enable(). Til sidst bliver isr\_startEx(ISR\_ESP\_Handler) kaldt som initiere at funktionen ISR\_ESP\_Handler til at kører når der kommer et interrupt på rx pinnen.

```

44
45 CY_ISR(ISR_ESP_Handler)
46 {
47     char readChar = UART_ESP_GetChar();
48
49     if (readChar == 'R')
50     {
51         dataReqFlag = true;
52     }
53 }
54

```

Figur 14: ISR for ESP modulet

I denne isr sker der egentlig ikke særligt meget men det er en meget vigtig del for koden. Når der bliver læst et R på vores rx for ESPen sætter vi dataReqFlag til at være true hvilket kører en kode nede i vores main løkke. Denne kode der bliver kørt kan ses her:

```

83     if (dataReqFlag)
84     {
85         uint16_t light = readLight();
86         uint16_t humid = readHumid();
87         uint16_t weight = readWeight();
88         uint16_t water = readWater();
89
90         UART_PC_PutString(uartBuffer);
91         sprintf(uartBuffer, sizeof(uartBuffer), "%c%c%c%c%c%c%c%c",
92                 water>>8,water,
93                 humid>>8,humid,
94                 light>>8,light,
95                 weight>>8,weight);
96
97
98         for (int i = 0; i<8 ;i++)
99         {
100             UART_ESP_PutChar(uartBuffer[i]);
101         }
102
103         if (humid > 17900)//value to be determined
104         {
105             PWM_Start();
106             PWM_WriteCompare(255);
107             CyDelay(10000);
108             PWM_Stop();
109         }
110
111         sprintf(uartDebugBuffer, sizeof(uartDebugBuffer), "%d %d %d %d\n",
112
113         dataReqFlag = false;
114     }
115
116

```

Figur 15: Kode der kører efter en interrupt

I denne kode er det her vi indsamler de forskellige data hvorefter vi sender dem ud til ESPen igennem UARTEn. I starten er det der hvor vi kalder alle read funktionerne. Disse værdier bli-

ver gemt og konverteret til chars ved hjælp af sprintf funktionen. Derefter bruger vi funktion UART.PutChar til at sende en char af gangen ud igennem UARTEn.

Al den information jeg bruger om UART kan findes i datasheetet infineon har udgivet om den [4]

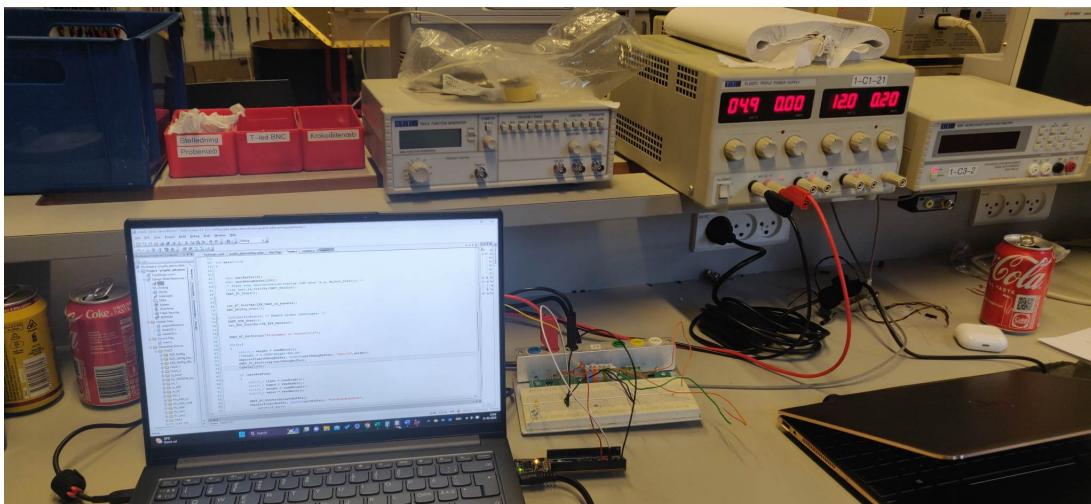
Efter dette kommer så den del det tænder for pumpen hvis det vi læser på fugtighedssensoren går inden for det interval der er defineret tørt. Hvor vi til sidst sætter dataReqFlag tilbage til at være false og venter på næste gang vi får et interrupt.

## 4 Modultest

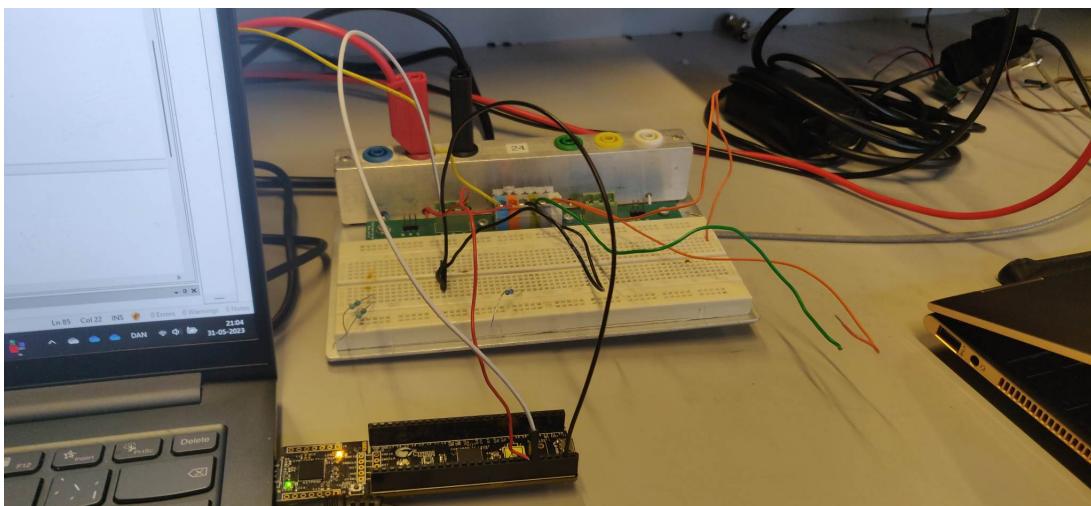
Ved PSoC modulet er der ikke særligt meget der skal testes for sig selv. Dette er fordi meget af det PSoCen handler om er at danne forbindelse mellem hardware og software. Så meget testen af dette modul kommer ved integrationstesten. Dog skal sigma-delta enheden testes om den er lineær med de værdier den giver.

### 4.1 test af ADC

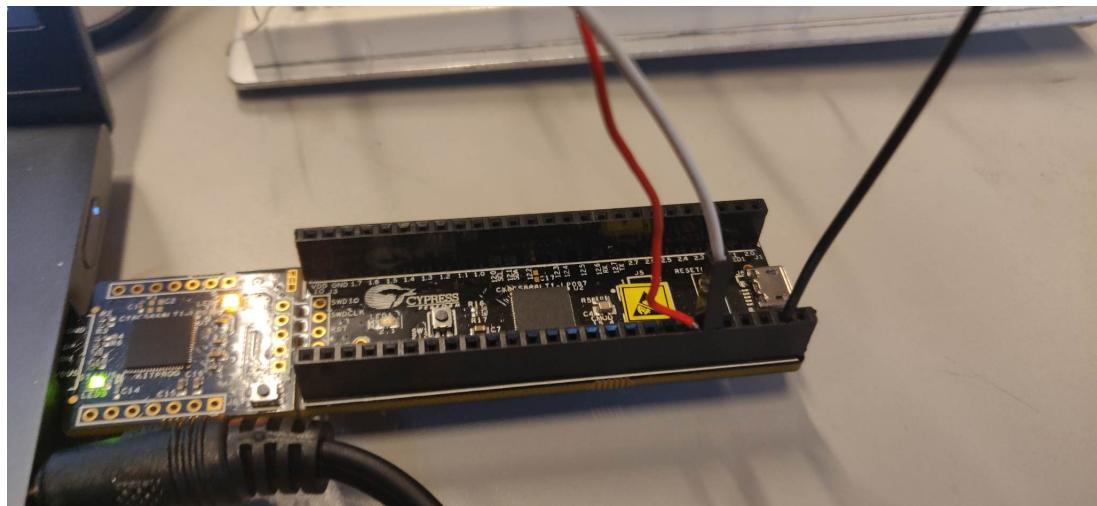
Ved test af sigma-delta ADC laver vi en opstilling hvor vi har en strømforsyning komblet til ADCen. Her kan vi så kontrollere spændingen der kommer ind på input pinnen. Forsøgsopstillingen til dette så således ud:



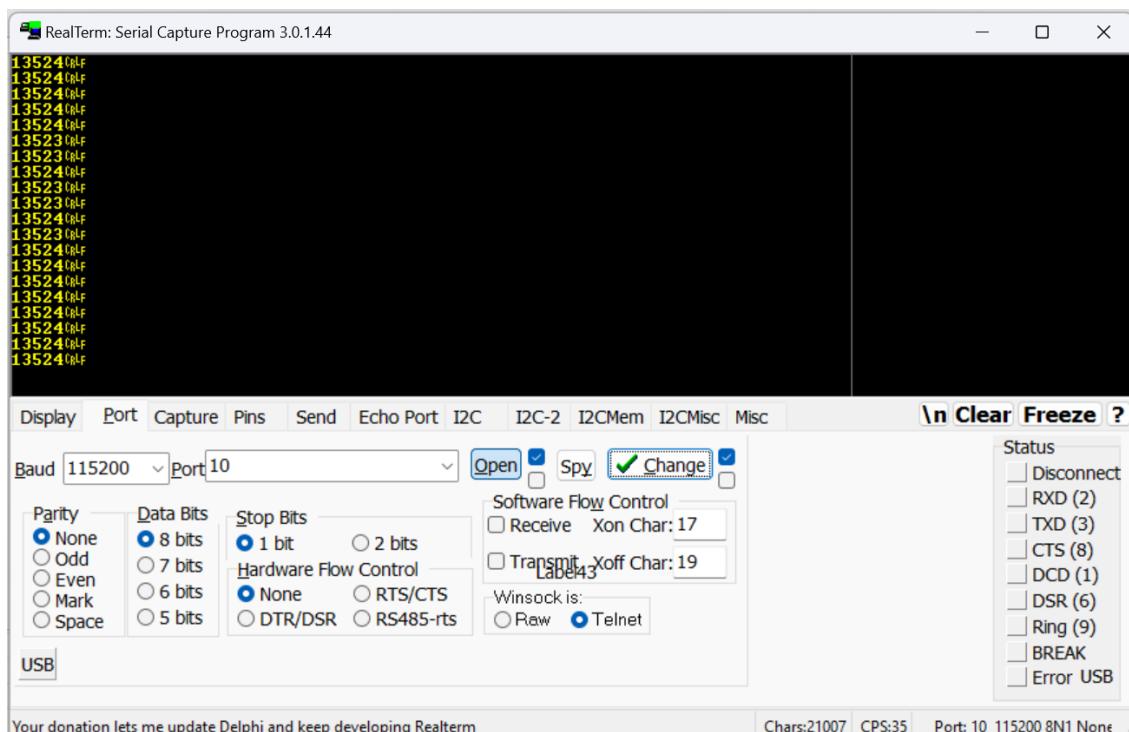
Figur 16: overbliks billede af modultest



Figur 17: billede over ledninger ved modultest



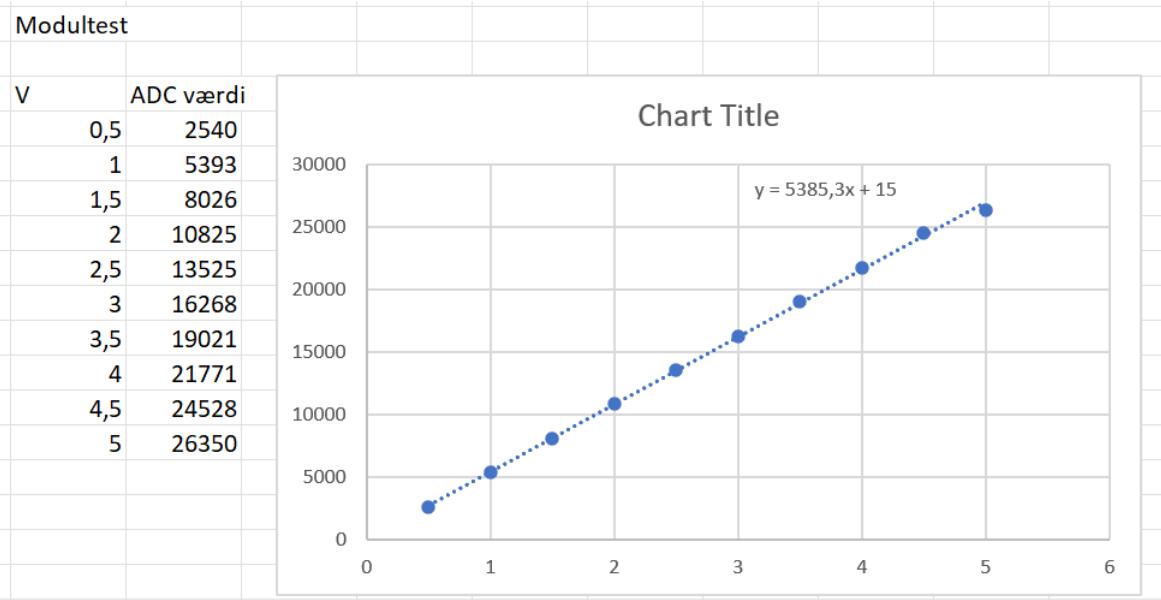
Figur 18: billede over PSoC ved modultest



Figur 19: terminalvindue under modultest ved 2.5V

på figur 18 kan man se den røde ledning der er input til adc, den hvide ledning der er referencen og den sorte der er fælles ground.

Ud fra dette forsøg har vi fået dataen:



Figur 20: Data for modultest

Her kan man se at vores ADC er linear hvilket er det den skal være. Man kan dog også se at der er nogle få usikkerheder. Men dette skyldes højest sandsynligt fordi strømforsyningen kun bruger 2 betydende cifre når man arbejder mellem 0V og 5V så det kan skabe nogle store spænd som ADC'en kan måle.

Så ud fra denne modultest kan vi konkludere at vores ADC virker som tiltænkt.

## Litteratur

- [1] Søren Forchhammer. *AD and DA Conversion*. PowerPoint presentation. 2021. URL: <https://brightspace.au.dk//content/enforced/90355-LR19205/10%20ADDA/AD%20and%20DA%20conversion.pdf>.
- [2] Infineon Technologies AG. *Infineon Component Delta Sigma ADC (ADC DelSig) V2.20 Software Module Datasheets*. 27-10-2011. URL: [https://www.infineon.com/dgdl/Infineon-Component%5C\\_Delta%5C\\_Sigma%5C\\_ADC%5C\\_\(ADC%5C\\_DelSig\)%5C\\_V2.20-Software%5C%20Module%5C%20Datasheets-v03\\_03-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e7ca0b90e6d](https://www.infineon.com/dgdl/Infineon-Component%5C_Delta%5C_Sigma%5C_ADC%5C_(ADC%5C_DelSig)%5C_V2.20-Software%5C%20Module%5C%20Datasheets-v03_03-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e7ca0b90e6d).
- [3] Infineon Technologies AG. *Infineon Component PWM V2.20 Software Module Datasheets*. 20-06-2012. URL: [https://www.infineon.com/dgdl/Infineon-Component%5C\\_PWM\\_V2.20-Software%5C%20Module%5C%20Datasheets-v03%5C\\_03-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e801c7611bd](https://www.infineon.com/dgdl/Infineon-Component%5C_PWM_V2.20-Software%5C%20Module%5C%20Datasheets-v03%5C_03-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e801c7611bd).
- [4] Infineon Technologies AG. *Infineon Component UART V2.30 Software Module Datasheets*. 8-02-2016. URL: [https://www.infineon.com/dgdl/Infineon-Component%5C\\_UART%5C\\_V2.30-Software%5C%20Module%5C%20Datasheets-v02\\_05-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e7fa4db1163](https://www.infineon.com/dgdl/Infineon-Component%5C_UART%5C_V2.30-Software%5C%20Module%5C%20Datasheets-v02_05-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e7fa4db1163).