

Indhold

Indhold	i
1 Hardwareimplementering	1
1.1 Version	1
1.2 Spændingsforsyning	1
1.3 Kodelås	1
1.4 Zerocross Detector	4
1.5 Carrier generator	5
1.6 Light module	8
1.7 Carrier Detector	10
2 Softwareimplementering	12
2.1 Version	12
2.2 PC	12
2.2.1 Scenario	12
2.2.2 Scenario constructor	12
2.2.3 <code>getAction</code>	12
2.2.4 <code>sortActions</code>	12
2.2.5 <code>operator ostream</code>	12
2.3 Action	13
2.3.1 Constructors	13
2.3.2 <code>setUnit</code>	13
2.3.3 <code>setCommand</code>	13
2.3.4 <code>setTime</code>	13
2.3.5 <code>getUnit</code>	13
2.3.6 <code>getCommand</code>	13
2.3.7 <code>getTime</code>	13
2.3.8 <code>ostream operator</code>	13
2.3.9 Less than operator	13
2.3.10 UI	14
2.3.11 PC UART	16
2.3.12 PC Main	18
2.4 Transmitter blokken	19
2.4.1 Action klassen	19
2.4.2 Codelock klassen	19
2.4.3 Time klassen	19
2.4.4 Transmitter klassen	19
2.4.5 Tx10 klassen	20
2.4.6 TxUART klassen	23
2.5 Receiver	25
2.5.1 Rx10 klassen	25
2.5.2 Receiver control klassen	27

1 Hardwareimplementering

1.1 Version

Dato	Version	Initialer	Ændring
03. december	1	HBJ	Første version

1.2 Spændingsforsyning

Spændingsforsyningen realiseres i første omgang på fumlebræt, hvorefter det måles med multimeter at modulet leverer hhv. $+5,0V$ og $-5,0V$.

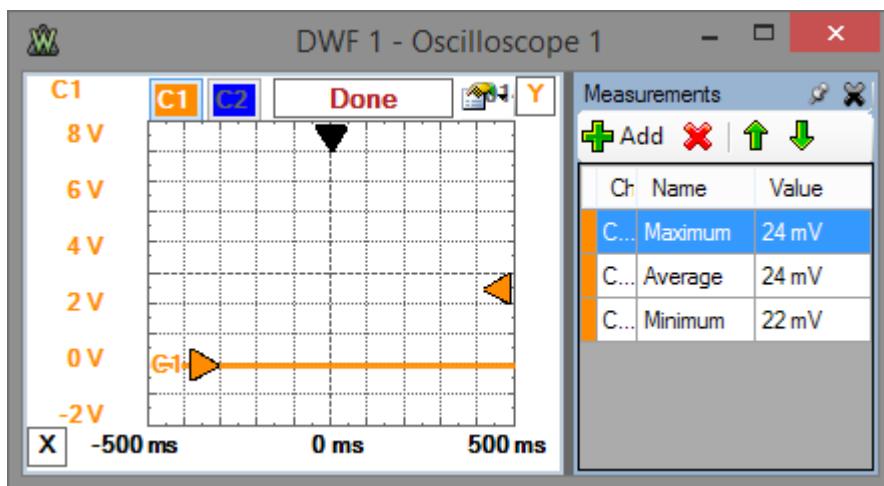
Som nævnt i designafsnittet er der et spændingsfald over spændingsregulatoren, hvorfor denne monteres med køleplade.

Herefter loddes spændingsforsyningen på veroboard.

1.3 Kodelås

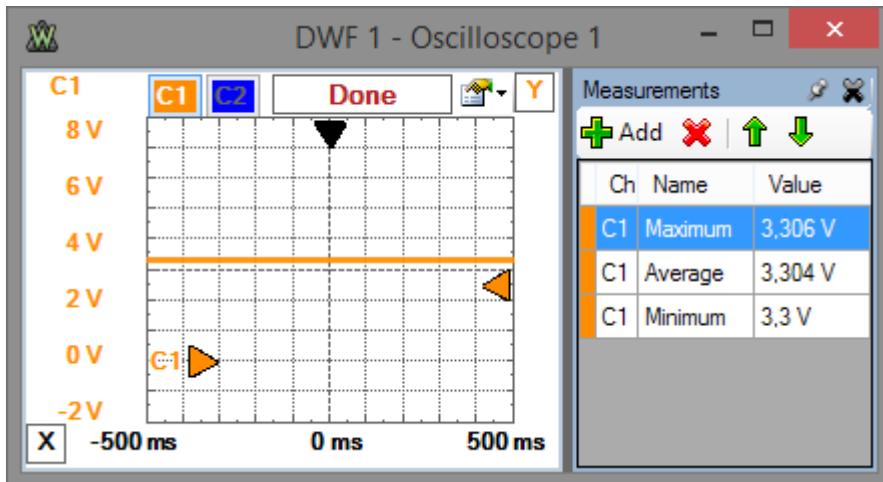
I dette afsnit testes hvorvidt kommunikationen mellem Altera DE2 boardet og Atmel STK500 kittet virker.

Vhdl filerne for kodelåsen(fremstillet i DSD exercise 7) downloades til DE2 boardet og der monteres et Analog discovery oscilloskop på DE2 boardets GPIO[1] pin 0 med en reference på GPIO[1] pin 11(GND).



Figur 1: Analog discovery - Oscilloskop: DE2 LOW output

På figur 1 og 2 kan det ses at et logisk HIGH output på DE2 boardet er målt til en spænding på $3.3V$ og en logisk LOW har en spænding på $0.0V$.



Figur 2: Analog discovery - Oscilloskop: DE2 HIGH output

For at teste om dette er en spænding der er høj nok til at STK500-kittet registrer det som HIGH, laves et lille testprogram. Testprogrammet sætter *PORTC* LOW, når *PORTA* pin 0 sættes HIGH og omvendt.

```

1 #include <avr/io.h>
2
3 void main( void )
4 {
5     DDRA = 0x00; // PORTA is input
6     DDRC = 0xFF; // PORTC is output
7
8     while(1)
9     {
10         if (PIN0==0b00000001) // Read PORT A0 and checks if it is HIGH
11         {
12             PORTC = 0x00; // LEDs turns on (PORTC is LOW)
13         }
14         else
15         {
16             PORTC = 0xFF; // LEDs turns off (PORTC is HIGH)
17         }
18     }
19 }
```

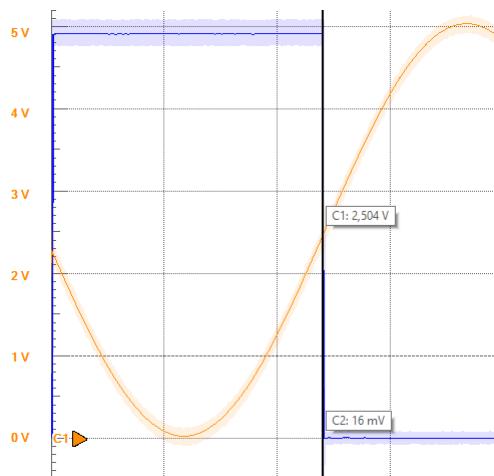
Med en Analog discovery's funktionsgenerator laves et sinussignal med en amplitude på 5V som sættes til *PORTA* pin 0. Med analog discovery's oscilloskop måles nu på et tilfældig ben på *PORTC*. Det ses på figur 3 at der skal påtrykkes en spænding på 2.3V for at STK500-kittet til at skifter fra at registrere HIGH til at registrere LOW.

Det ses på figur 4 at der skal påtrykkes en spænding på 2.5V for at STK500-kittet til at skifter fra at registrere LOW til at registrere HIGH.

For at kunne udføre en samlet modultest, tilsluttes STK500-kittet PORT A pin 0 til DE2 boards GPIO[1] pin 0 og et GND-ben på STK500-kittet tilsluttes GPIO[1] pin 11. Ydermere tilsluttes STK500-kittet PORT C til dets indbyggede LED'er.



Figur 3: Analog discovery - Oscilloskop: STK-500 Faling edge



Figur 4: Analog discovery - Oscilloskop: STK-500 Rising edge

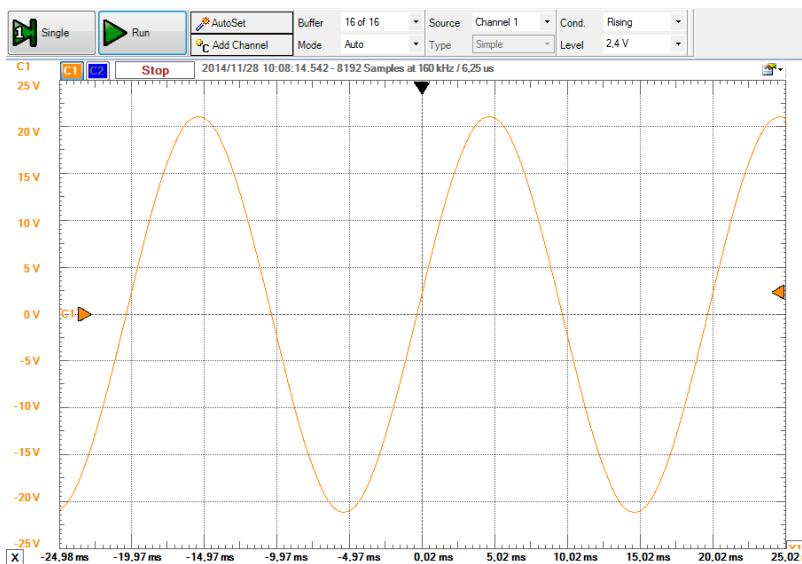
På både STK500-kittet og DE2 boardet ligger samme software som i de ovenstående test. Det kan nu ses at STK500-kittets LED'er tænder og slukker som forventet, alt efter kom koderne på DE2-boardet er indtastet korrekt.

Altså er kan det konkluderes at vi godt kan bruge DE2 boardet direkte til STK500-kittet.

1.4 Zerocross Detector

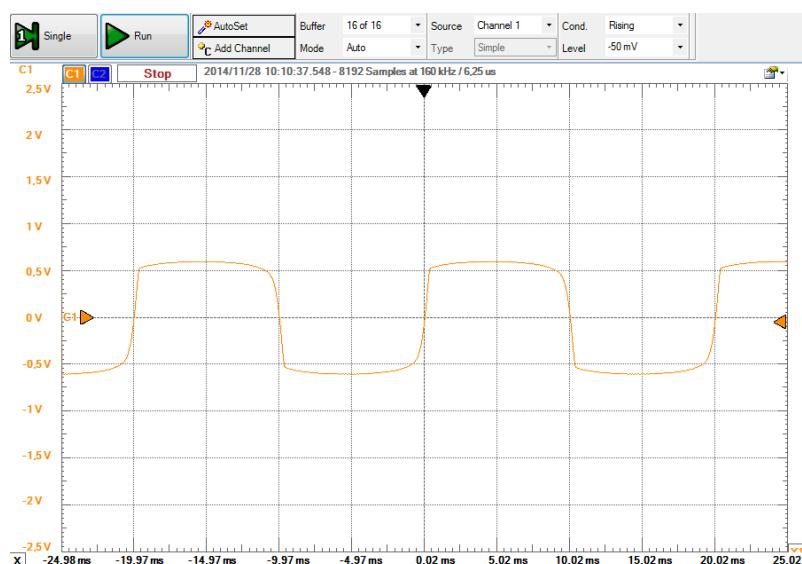
For at sikre at de enkelte dele af zerocross detectoren virker, realiseres kredsløbet på fumlebræt i flere etaper.

Først realiseres kredsløbets lavpasfilter, og det konstateres vha. oscilloskop, at det virker som forventet. Der er valgt en $33k\Omega$ modstand til filteret ($R1$), da dette er den nærmeste værdi, der er tilgængelig. Figur 5 viser en måling af signalet efter lavpasfilteret.



Figur 5: Ren sinus efter lavpasfilter

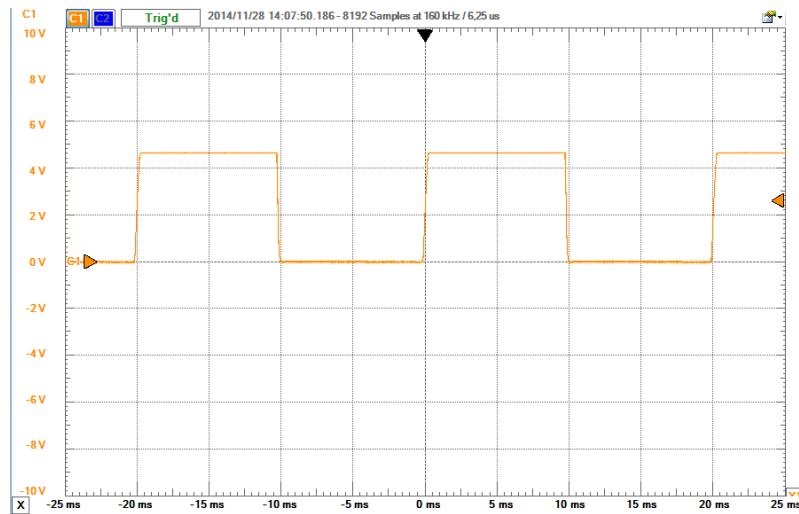
Herefter tilføjes de to dioder ($D1$ og $D2$) på fumlebrættet, og det ses at signalet begrænses som forventet. Signalet er vist på Figur 6.



Figur 6: Dæmpet Sinus efter dioder

Herefter tilføjes komparatoren, som sammenligner signalet fra Figur 6 med stel. R_2 og R_3 bevirket at der er en hysterese på $200mV$.

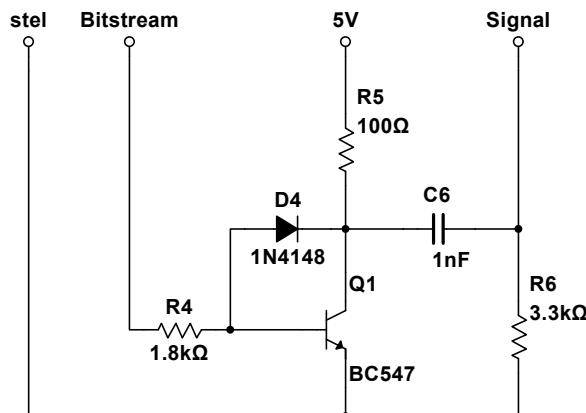
Indgange på ubrugte gates er sat til stel. Dioden efter operationsforstærkeren fungerer som forventet. Udgangssignalet er som ønsket (Figur 7).



Figur 7: ZeroCrossDetect signal efter comparator

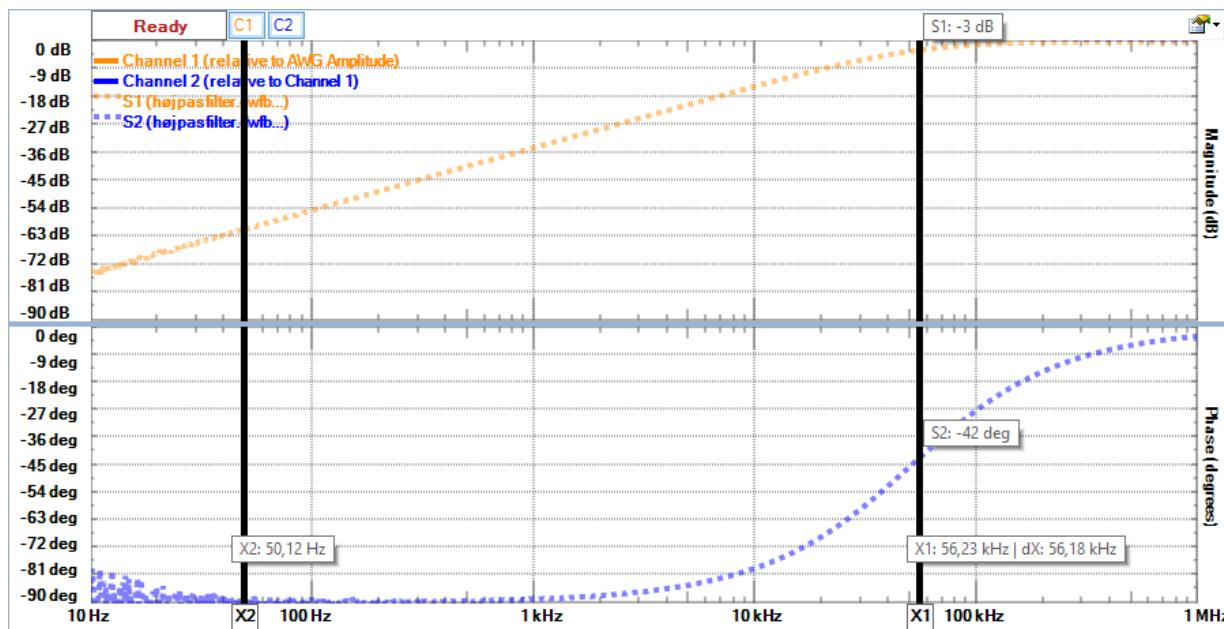
1.5 Carrier generator

Først opbygges kredsløbet på fumlebræt, hvor de nedensående del-elementer testes. Når del-elementerne, er testet med succes, laves en samlet modultest for Carrier generatoren. Efter fuldendt modultest, loddes et veroboard med de eventuelle ændringer til designet, der er foretaget.



Figur 8: Carrier generator

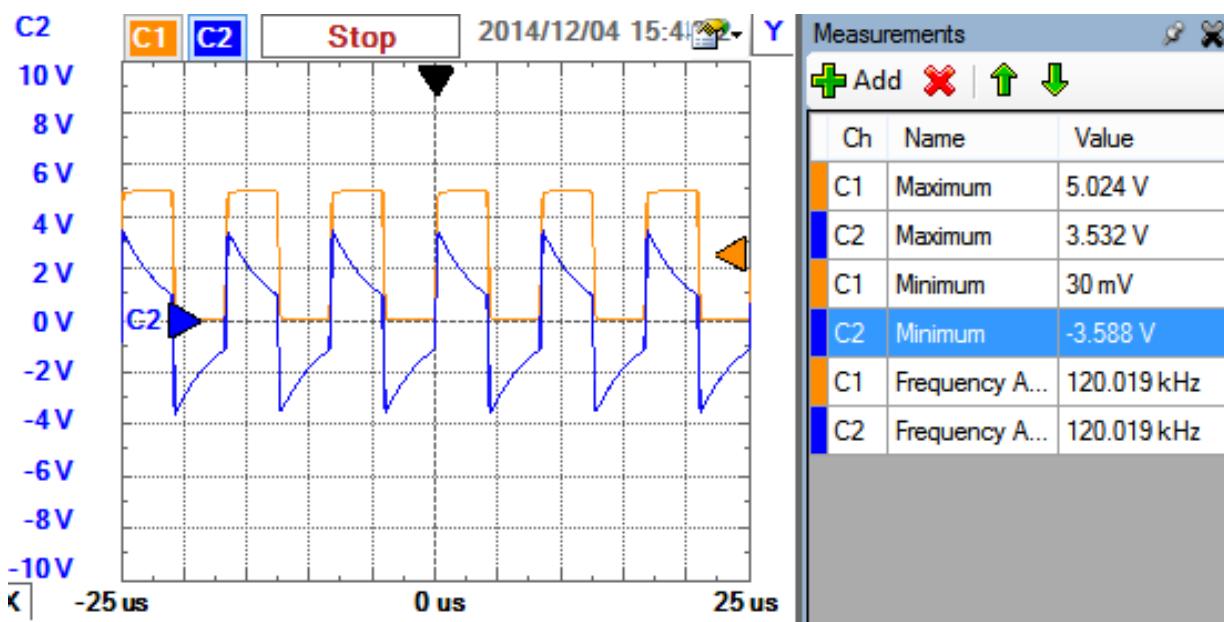
Der laves en del-test af højpasfilteret, som er består af C_6 og R_6 . Testen skal sikre at filteret opfører sig jf. bodeplotet i design-dokumentet, Figur 9.



Figur 9: Analog discovery - Network Analyzer: Bodeplot af højpasfilter

Med *Network Analyzer* funktionen på *Analog discovery* er der konstrueret et bodeplot, der viser frekvenskarakteristikken for højpasfilteret. Se Figur 9. Det ses at knækfrekvensen ligger som forventet tæt på 50kHz , svarene til $100\pi \cdot 10^3\text{rad/s}$. Ydermere kan det ses at de 50Hz bliver dæmpet med omkring 60dB , hvilket er en tilstrækkelig dæmpning. Dæmpningen af 120kHz signalet er tæt på 0dB . Overordnet ser det ud som forventet og beskrevet i designdokumentet.

Ved at sende et 120kHz firkantsignal gennem filteret, får vi et signal der viser opladning og afladning af kondensatoren.



Figur 10: Måling af firkantsignal efter passering af højpasfilter.

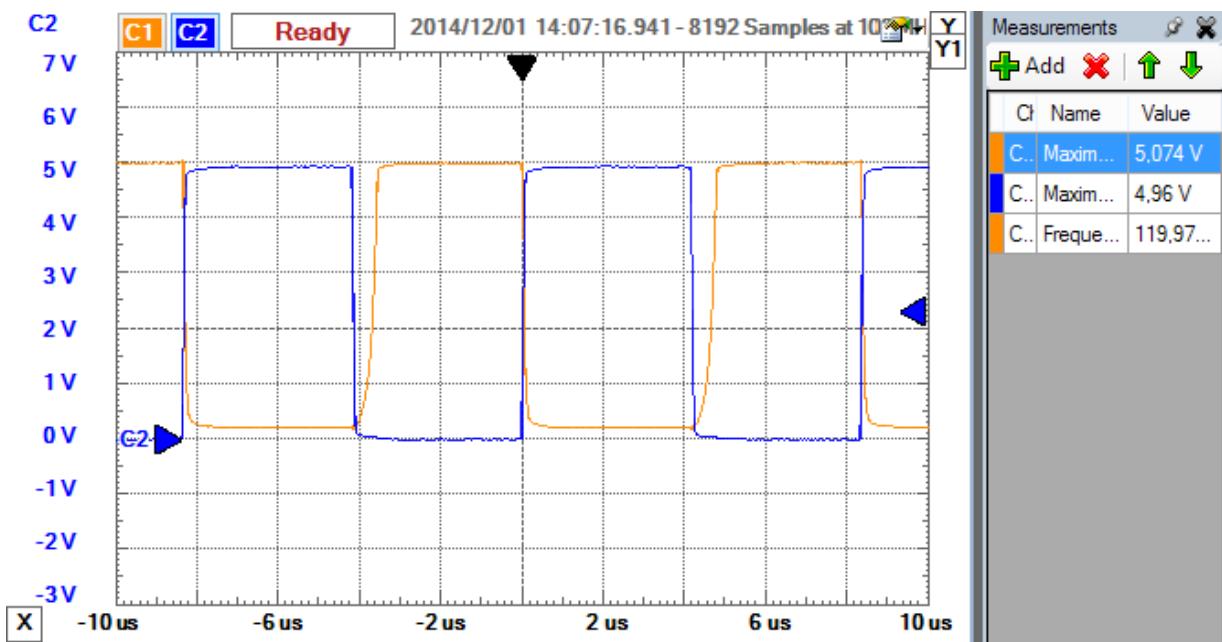
Outputtet kan ses på Figur 10. Den orange kanal viser et 120kHz firkantsignal, mens den blå viser firkantsignalet efter passage gennem højpasfilteret.

Oprindeligt blev kredsløbet designet med en *BD139* transistor, men det viste sig at *Bitstream*, ($0V - 5V$) firkantsignal, ikke kunne passere igennem fra transistorens kollektør til emitter. Efter en fejlsøgning kan det konkluderes, at dette skyldes en positiv opladning, når signalet var HIGH, som transistoren ikke kan nå at aflade inden næste periode.

Hvis transistoren skal anvendes, er det nødvendigt at fjerne den positive ladning der skabes, hurtigere end den selv kan aflade. Problemet kan løses ved at indsætte en diode (D_4) parallelt med kollektør-base benene med spærretning mod kollektor (Baker clamp). Dette vil hjælpe transistoren til at aflade den positive spænding hurtigere.

En anden løsning er, at skifte transistoren *BD139* ud med en anden transistor, der har et mindre storage delay, fx *BC547*. Konsekvensen heraf vil dog være at transistoren kun kan trække $100mA$ middelstrøm.

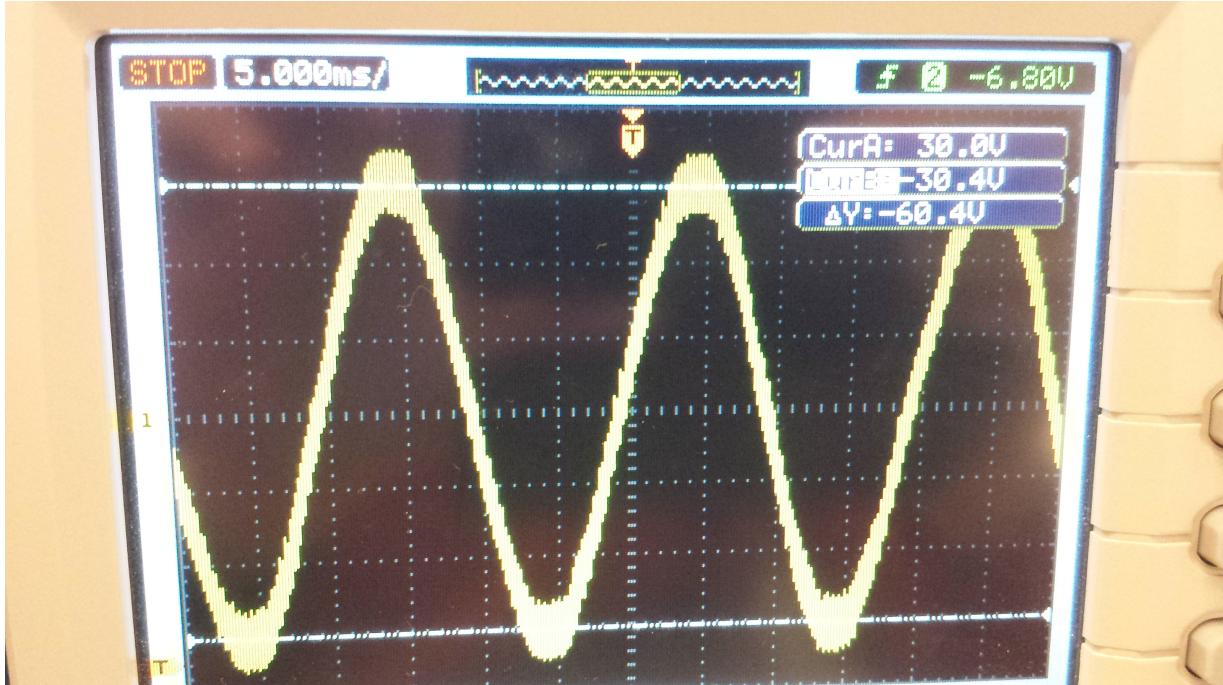
Der vælges en blanding af de to løsninger, således transistoren nu er en *BC547* og vi indsætter en diode parallelt med kollektør-base benene.



Figur 11: Analog discovery - Oscilloskop: Transistor-kredsløb

På figur 11 ses en oscilloskop-måling af transistor-kredsløbet. Her ses at udgangen *Signal* (den orange), er inverteret i forhold til indgangen *Bitstream* (den blå). Det kan ligeledes ses at der er en lille opladnings- og afladningstid på udgangen, der skyldes opladningen af transistoren. Dette har dog ingen væsentlig betydning i forhold til anvendelsen af signalet.

En samlet modultest af Carrier generator udføres med et oscilloskop på udgangen af carrier generatoren (*Signal*).



Figur 12: Oscilloskop: Carrier generator modultest

Figur 13 viser samme signal som figur 12. Der er blot zoomet ind så der kun er $2\mu s$ mod $5ms$ pr. streg på tidsakssen. Testen viser at der bliver påtvunget et $120kHz$ signal på udgangen *Signal*, der i forvejen indeholder en $18VAC - 50Hz$ frekvens.

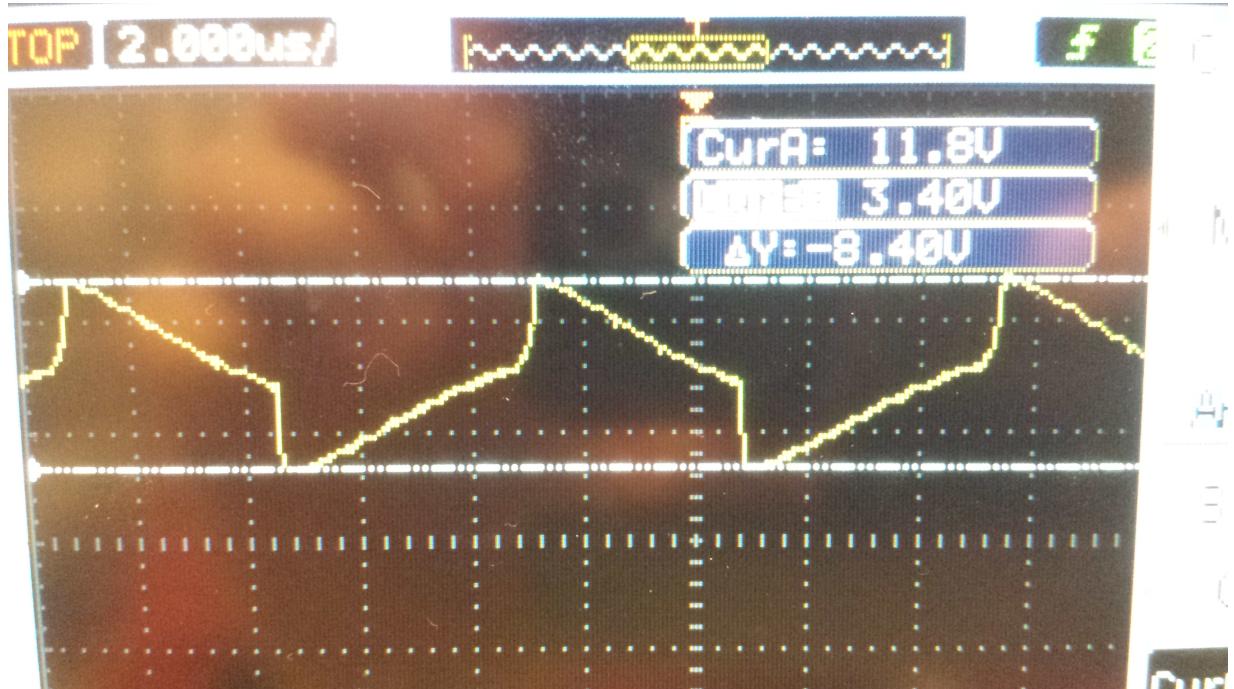
Det bemærkes at signalet har en peak to peak spænding på $3.4V$, hvilket er naturligt, da det har været udsat for en spændingsdeler.

1.6 Light module

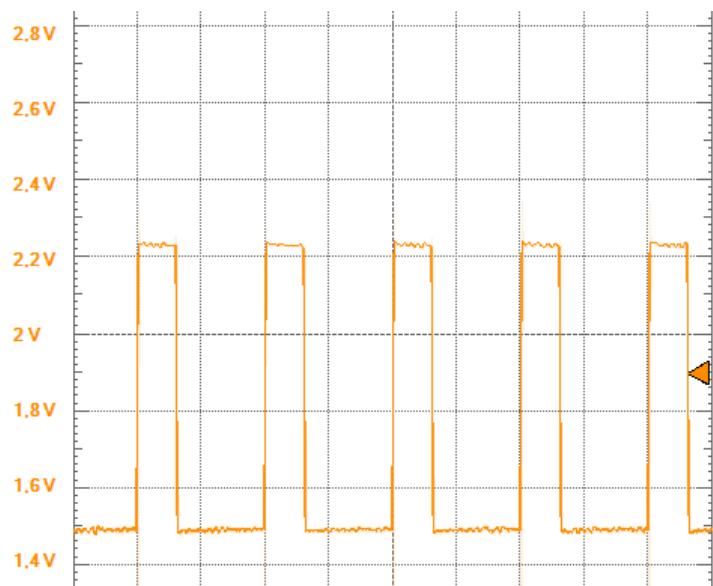
Light module er opbygget på fumlebræt før det blev loddet på et veroboard.

Frekvensen er valgt til $1kHz$, jf. signalbeskrivelsen, eftersom en mindre frekvens vil give et synligt blinkende lys. Duty-cycle blev testet fra 0% til 100% i spring af 10% , og viste et tydeligt skift i lysstyrke.

Efter test på fumlebræt blev modulet implementeret direkte på veroboard, grundet det simple design.



Figur 13: Oscilloskop: Carrier generator modultest zoom

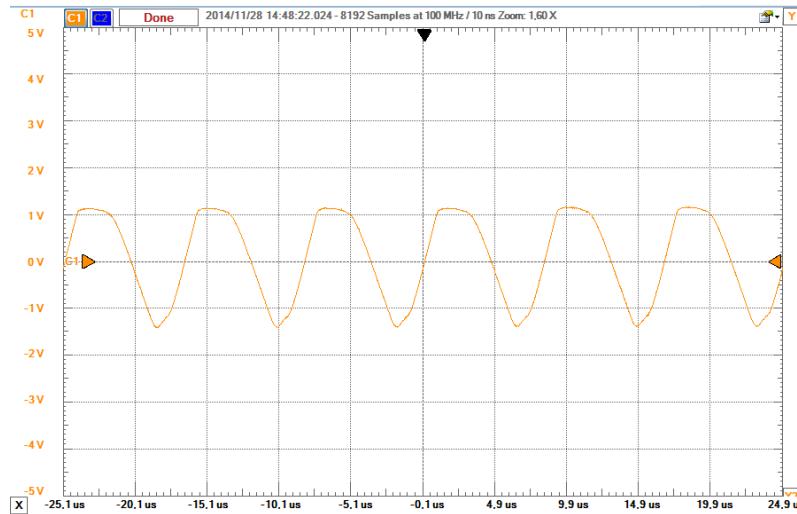


Figur 14: Spændng over lysdiode, ved et 0 – 5V firkantsignal, med 30% duty cycle

1.7 Carrier Detector

For at sikre at alle dele af kredsløbet fungerer, deles det op i mindre blokke, og implementeres på fumlebræt.

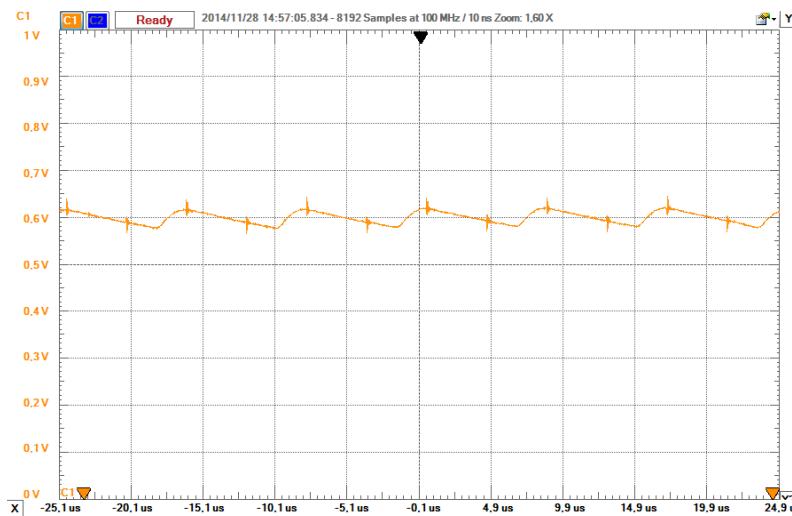
Først realiseres båndpasfilteret for at konstatere, at $18VAC\ 50Hz$ delen er dæmpet, men $120kHz$ signalet fortsat er til stede.



Figur 15: Signal efter båndpas

Det ses på Figur 15, at båndpasfilteret virker som forventet. $18VAC\ 50Hz$ signalet er væk. $120kHz$ signalet ser noget anderledes ud, men det har ingen betydning.

Herefter realiseres selve envelope detektoren ($D5$, $R11$ og $C9$), og der måles med oscilloskopet på udgangen. Signalet er vist på Figur 16.

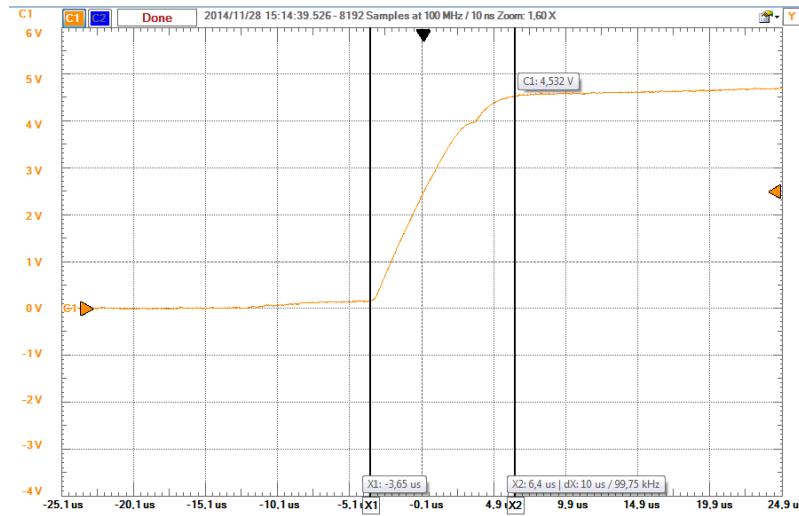


Figur 16: Signal efter envelope detector med $120kHz$

Det ses på Figur 16, envelopedetektoren fungerer. 120kHz signalet ses tydeligt på takkede signal, og man kan desuden se at forholdet mellem $R11$ og $C9$ er passende; afladningstiden er lang nok. Det testedes desuden at udgangen på envelope detektoren gik til 0V , når 120kHz signalet fjernes.

Signalet efter envelope detektoren ligger mellem 480mV og 600mV , og den efterfølgende komparator har derfor en reference på 318mV på den inverterende indgang. Komparatoren $U4B$ og dioden $D6$ samt pull-down modstanden $R14$ realiseres på fumlebrættet, og spændingen på udgangen måles. Uden detektion af 120kHz signal måles 0.0V og med detektion af 120kHz signal måles 4.7V , hvilket ligger inden for tolerancen jf. signalbeskrivelsen.

For at sikre at rise time på bitstream er hurtig nok, måles dette med oscilloskopet; resultatet ses på Figur 17.



Figur 17: Rising edge på bitstream

Det ses på Figur 17, at signalet er ca. $10\mu\text{s}$ om gå fra lav til høj, hvilket er tilfredsstillende. Det ses desuden, at der ikke er prel på signalet.

2 Softwareimplementering

2.1 Version

Dato	Version	Initialer	Ændring
27. november	1	LS	Første udkast af dokumentet

2.2 PC

2.2.1 Scenario

2.2.2 Scenario constructor

Scenario klassen bruger STLs vector container format istedet for et array, siden den tillader brug af iterators. Constructoren bruger stl's assign funktion til at oprette 20 aktion objekter med default værdier.

2.2.3 getAction

Methoden returner en reference til et aktion objekt i vectoren på baggrund af pladsen i den.

2.2.4 sortActions

sortActions bruger STL's sorterings funktionalitet til at sætte de aktions objekter i numerisk rækkefølge set i forhold til deres tids værdi. For at vi kan sammenligne aktion objekter, har aktion klassen fået overridet dens «"operator. sortAction har ikke længere en returværdi, men muterer et scenarios vector af action objekter.

```
1 void Scenario :: sortActions ()  
2 {  
3     sort (scen_. begin () , scen_. end ()) ;  
4 }
```

2.2.5 operator ostream

Scenario klassens ostream operator er også blevet overridet til brug i UI klassen. «<"operatoren udskriver nu alle aktion objekters værdier som: Tid, kommando og enhed.

2.3 Action

2.3.1 Constructors

Aktion klassen indeholder to constructors: En default constructor, der kaldes 20 gange når Scenario først oprettes, og en explicit constructor til de prædefinere scenarier.

2.3.2 setUnit

set-metode til at sætte aktionens enhedsværdi.

2.3.3 setCommand

Set-metode til aktionens kommandoværdi. Funktionen oversætter kommandoens talværdi til en char, der vil blive brugt i UART klassen.

2.3.4 setTime

Set-metode til aktionens tidsværdi.

2.3.5 getUnit

Get-metode til aktionens enhedsværdi.

2.3.6 getCommand

Get-metode til aktionens kommandoværdi.

2.3.7 getTime

Set-metode til aktionens tidsværdi

2.3.8 ostream operator

Dette skyldes override funktionen for `outstream` operatoren er implementeret som en fri funktion, og kan derfor ikke direkte tilgå de private members.

???++

2.3.9 Less than operator

Yderligere er der tilføjet en overload af operatoren `<`, så et Action objekt kan sammenlignes direkte på dens tids værdi.

2.3.10 UI

UI'en er en klasse der består af en række metode-kald. Hver af metoderne clearer skærmen for tidlige re menu, og tegner den nye menu alt efter hvilken menu, der bliver kaldt. Skærmen bliver clearet ved brug af kommando'en `system("cls")`. som forekommer i starten af alle metode-kaldene. En metode afsluttes ved at returnere en integere i alle tilfælde, men untagelse af funktionen `drawStopPrompt` som returnere en boolean.

drawMainMenu

drawMainMenu metoden udskriver en liste over mulige menuer på skærmen hvorefter brugerens får mulighed for at indtaste, hvilken menu der ønskes. ved brug af `Cin` til at læse input fra brugerens, kan brugerens også indtaste forkerte inputs som f.eks. "AA3". Til at forhindre at forkerte inputs bliver indlæst, bliver der fortaget en validering af den indtastede int. Det tjekkes at inputtet er imellem 1 og 3, og alle andre inputs resulterer i en fejlmeddelse på skærmen, og beder brugerens om at prøve igen.

```
1 cin >> pick;
2     if (pick >= 1 && pick <= 3) // tjekker at input værdien er indenfor gyldige
3         grænser
4         return pick;
5     else
6         cout << "ugyldig menu, pr\x9Bv igen: ";
```

Ved inputs som f.eks. "AA3", vil inputtet ikke kunne valideres ordenligt og resultere i en uendelig løkke af fejlmeddelser. Ved at flush keyboardets buffer, sørges der for at uanset hvor lang en tekst brugerens indtaster, resultere det kun i en enkelt fejlmeddelse.

```
1 cin . clear ();
2 fflush (stdin);
```

drawScenList

Metoden `drawScenList` udskriver en liste med beskrivelsen af 3 forudindstillede scenarier. hvorefter brugerens kan vælge en af de forudindstillede scenarier. Validering af input og valg fungere på samme måde som `drawMainMenu` metoden.

drawScenario

Metoden `drawScenario` får en reference til et scenarie ind som parameter. Metoden bruger Scenariet ostream operator til at udskrive de 20 aktioner, der ligger i scenariet ud på skærmen.

```
1 int drawScenario ( Scenario & Scen )
2 {
3     ...
4     cout << Scen ;
```

Brugerens kan herefter vælge en af de 20 aktioner, ved at indtaste nummeret på aktionen. valg og validering af aktion fungerer på samme måde som `drawMainMenu`, dog er validering af input sat imellem 0 og 19, istedet for 1 og 3.

```
1 if ( pick >= 0 && pick <= 19 )
```

drawAskUnit

metoden **drawAskUnit** udskriver en liste over Units der kan vælges til udførsel af en kommando. Brugeren kan vælge mellem 2 lamper, et TV og en radio. valg og validering af input fungere på samme måde som **drawMainMenu**, dog er validering af input sat imellem 1 og 2, som er de 2 lamper. TV og Radio kan ikke vælges, da de ikke implementeres i systemet.

drawAskCommand

metoden **drawAskUnit** udskriver en liste over kommandoer der kan udføres af den givne Unit. brugeren kan vælge mellem at tænde, slukke og dimme. Valg fungere som i **drawMainMenu** metoden. Validering af input tjekker her for om det er enten tænd, sluk eller dim der er valgt. Ved tænd og sluk returnere koderne for det valgte, men hvis det valgte er dim, får brugeren mulighed for at vælge styrken af dimming. Styrken af dimming valideres og returneres. Hvis brugeren indtaster en værdi uden for det gyldige område, bliver metoden kørt fra begyndelsen, hvor tænd, sluk eller dimming skal vælges igen.

drawAskTime

metoden **AskTime** fungere ved at brugeren bliver bedt om at indtaste hvilket time-tal på dagen den ønskede kommando skal udføres på. Herefter bliver brugeren bedt om at indtaste det ønskede minut-tal på den valgte time. Tiden valideres, og hvis både time-tal og minut-tal er indenfor de gyldige grænser returneres tiden siden midnat i minutter. *timer * 60 + minutter* Hvis enten time-tal eller minut-tal er udenfor de gyldige grænser, får brugeren besked om at den indtastede tid er ugyldig og keyboard-bufferen tømmes. Når en knap trykkes ved fortsættes programmet, og metoden bliver startet forfra. Til at vente på en knap trykkes er brugt **_getch()** som kan bruges til at hente hvilken knap på keyboardet er trykket. I dette tilfælde er funktionen brugt til at tjekke om en knap bliver trykket på, og ikke for at gemme inputtet af hvad der er trykket på.

```
1 if (hour >= 0 && hour <= 23 && min >= 0 && min <= 59)
2     return (hour * 60 + min); // returnere tiden siden kl 00
3 else
4     cout << "ugyldig tid, pr\x9Bv igen";
5
6 cin.clear();
7 fflush(stdin); // clear og flusher alt gemt i keyboard bufferen
8 _getch(); // venter paa hvilket somhelst tryk paa keyboard
```

drawStopPromt

drawStopPromt metoden udskriver på skærmen en besked om brugeren er sikker på at afviklingen af scenarie skal afsluttes. Ved at trykke på knappen "Y/y" på keyboardet, bekræftes det at det skal afsluttes og der returnes **true**, en hver anden knap resultere i, at der returneres **false**

```
1 get = _getch();
2 if (get == 89 || get == 121) // ascii værdi for y og Y
3     return true;
4 else
5     return false;
```

2.3.11 PC UART

PC'ens UART har til formål at oversætte de data, der er i *Scenario*-objektet, og transmittere dem over på transmitteren. Dette gøres ved brug af et open-source bibliotek kaldet RS232, der følger den protokol det ønskes at transmittere data med.

For at sende data ud til Transmitteren, skal der oprettes en forbindelse gennem en USB port. her bruges **UART_connect** til formålet.

Systemet kræver også at kunne få status på kodelåsen, hvilket metoden **getLockStatus** står for. Kravet for at bruge **getLockstatus** er, at **UART_connect** har været kørt, for at finde en gyldig port at sende og modtage på.

Når data skal transmitteres bruges **sendScen**, der tager en reference til et scenarie som parameter, og får oversat dataene i scenariet til at følge UART protokollen, hvorefter den sender dem over til transmitteren.

UART_connect

Metoden **UART_connect** bruges til at finde en gyldig COM-port på computeren. Metoden kører igennem alle COM-porte på PC'en ved at bruge RS232-bibliotekets funktion **RS232_OpenComport**, til at tjekke om der sidder et RS232 stik i den givne COM-port. Så længe den testede COM-port ikke har et RS232 stik i, tæller metoden **Cport_nr** en op, og derved tester den næste COM-port, indtil en gyldig COM-port er fundet.

```
1 while (RS232_OpenComport(cport_nr, bdrate, mode))  
2 {  
3     printf("ugyldig Com-port\n");  
4     cport_nr++;  
5 }
```

getLockStatus

Metoden **getLockStatus** finder ud af kodelåsens status ved at sende et L ud på COM-porten. Dette sker ved at bruge RS232-bibliotekets funktion **RS232_cputs()**. Parametrene for funktionen er COM-port nummeret **cport_nr** og den mængde af data i form af chars, der skal sendes.

Da det kræver COM-port nummeret, er det derfor også nødvendigt, at metoden **UART_connect** har været kørt, før det er muligt at bruge **getLockStatus**.

```
1 RS232_cputs(cport_nr, "L");
```

Når beskeden om at tjekke status er blevet sendt til transmitteren, begynder metoden at læse på porten og venter på et svar tilbage fra transmitteren. Svaret vil indeholde et L eller et U, afhængig af om kodelåsen er hhv. låst eller oplåst. Til aflæsning på COM-porten bruges RS232-bibliotekets funktion **RS232_PollComport()**. Hvis det aflæste er et L returneres **true**, for at kodelåsen er låst. Hvis et U aflæses returneres **false**, hvilket betyder kodelåsen er låst op.

sendScen

metoden **Send Scen** har til opgave at oversætte data fra scenariet til chars som følger UART protokolen. til at starte med kører metoden koden nedenfor:

```
1 RS232_cputs(cport_nr, "N");
```

N'et betyder at der transmitteren skal være klar på at der nu kommer et ny datastrøm fra PCen. metoden henter herefter den første aktion, så `get` funktionen kan bruges til at hente data ud af aktionen. Kommandoen bliver sat direkte ind, da `get` funktionen for den returnere en char, derfor sker ingenting med den. Tiden skal regnes om, så transmitteren ved hvor længe der skal gå, før den skal udføre en kommando. Dette sker i koden set nedenfor:

```

1 //udregner tiden til den gældne commando skal ekseveres
2 int timeTillExecution;
3 if (action.getTime() >= Ctime())
4 {
5     timeTillExecution = action.getTime() - Ctime();
6 }
7 else
8 {
9     timeTillExecution = 1440 - (Ctime()-action.getTime());
10 }
11 //tager sig af at omregne tiden fra int til char og dele tiden op i 2 chars,
12 //da den kan ske at fylde mere end 8 bit
13 int timeL1 = timeTillExecution % 16;
14 int timeL2 = int((floor(timeTillExecution / 16) % 16)
15 int timeH = int((floor(timeTillExecution / 256)));
16
17 TimeLow1 = ((char)timeL1 +48);
18 TimeLow2 = ((char)timeL2+48)
      TimeHigh = ((char)timeH)+48);

```

Integeren `timeTillExecution` bliver udregnet ud fra en af 2 udregning. Den valgte udregning afhænger af om tidspunktet allerede har passeret systems klokkeslet. Hvis dette er tilfældet køres udregningen i linje 9. Denne udregning tager hensyn til hvor mange minutter der er tilbage på dagen og kører derfor på det angivne klokkeslet næste dag. udregningen i linje 5 sker hvis klokkeslettet endnu ikke har passeret tiden, hvor kommandoen skal udføres. Den udregner hvor mange minutter der er fra det gældende klokkeslet og frem til kommandoen skal udføres. `Ctime()` funktionen som fremgår flere steder er en hjælpefunktionen som returnere en integer med klokkeslettet på PCen. Når `timeTillExecution` er udregnet skal den omsættes til Chars. Da den maksimale værdi der vil fremkomme er 1440(antallet af minutter på et døgn), skal der bruges 2 chars til at gemme dataene i.

```

1 int timeL1 = timeTillExecution % 16;
2 int timeL2 = int((floor(timeTillExecution / 16) % 16)
3 int timeH = int((floor(timeTillExecution / 256)));

```

`timeL1` er hvor de 4 første bit gemmes, derfor tages modulus til 16 for at finde resten, og altså den værdi der skal gemmens i `timeL1`. `TimeL2` er hvor de midterste bit gemmes, derfor nedrundes der, og denne omskrives til en int. Derefter tages modulus til 16. `timeH` er hvor de 4 største bit gemmes. For at finde værdien der skal gemmes heri, divideres `timeTillExecution` med 256, og rundes ned, for præcision. Da nedrundingen sker som en double omskrives det til en int, efter nedrundingen. `timeL1`, `timeL2` og `timeH` har efter udregningen en maksimalstørrelse på 16, og kan derfor laves om til en char, med ASCII kode mellem 48 og 64

```

1 TimeLow1 = ((char)timeL1 +48);
2 TimeLow2 = ((char)timeL2+48)
3 TimeHigh = ((char)timeH)+48);

```

Unit oversættes ved brug af en switch set i koden nedenfor:

```

1 switch (action.getUnit()) // tager sig af at oversætte unit til at følge UART
2     {
3         case 1:
4             unit = 'A';
5             break;
6         case 2:
7             unit = 'C';
8             break;
9         case 3:
10            unit = 'P';
11            break;
12        case 4:
13            unit = 'N';
14            break;
15        default:
16            unit = 'O';
17            break;
18    }
19

```

Switchen tager uniten ind fra aktionen, og alt efter værdien sættes unit til den char der passer en den gældende unit.

UARTen sender nu dataene fra aktionen ud på COM-porten ved brug af funktionen RS232_cput

```

1 char data[6] = { unit , command, TimeHigh , TimeLow2, TimeLow1 };// data der skal
2     sendes over
3     strcpy(str [c] , data);
4     RS232_cputs(cport_nr , str [c]);

```

når dataene er sendt på på porten, venter UARTen 100milisekunder, hvorefter den begynder forfra med aktion nr.2. Dette sker til alle 20 aktioner er overført til transmitteren, hvorefter metoden afslutter.

stopAll

metoden stopAll er til at slukke alle enheder på system. Dette gør metoden ved at bruge funktionen RS232_cput til at sende et S til transmitteren. Transmitteren ved ifølge protokolen at et S betyder den skal slukke alt.

```
1 RS232_cputs(cport_nr , "S");
```

2.3.12 PC Main

PC'ens main har til formål at stykke alle PC'ens klasser sammen, og styrer alt fra input til output. Den er kodet således at UC1, UC2 og UC3 er opfyldt. Basalt set er dette en main fil som bruger alle funktioner fra PC'ens klasser: Action, PC UART, Scenario og UI.

Ud over dette er det under denne fil at koden til de predefinerede scenarier ligger.

2.4 Transmitter blokken

2.4.1 Action klassen

Selve Action-klassen består af en række get'er metoder og én set metode, som validerer data og genererer houseCode ud fra unitCode parametren. Der er ikke foretaget nogle videre interessante algoritmer eller yderligere hjælpefunktioner uddover dem, der forefindes i kapitlet Software Design.

2.4.2 Codelock klassen

Implementeringen af denne klasse er blot én metode `locked()`, som returnerer true, hvis kodelåsen er låst (benet er højt).

```
1 bool Codelock::unlocked() {
2     //TRUE if unlocked , FALSE is locked .
3     //Signal is LOW when code is correct , HIGH when not correct .
4     return !(PINA & 0b00000001);
5 }
```

2.4.3 Time klassen

Time klassens roller er at give TransmitterCtrl klassen en opdateret værdi i minutter hver 5 sekund. Dette tager højde for flere events, som vil køre på samme tid og giver Tx10 klassen mulighed for at sende hele aktionen før næste aktion påbegyndes. Klassen er blevet implementeret med en compare interrupt på Timer1, hvor der sker et overflow hvert sekund, som inkrementer en `seconds` variabel. Når denne `seconds` er modulo 0 med 5, sendes `minutes` til ctrl klassen, ellers hvis `second` overstiger 59 bliver den sat til 0 og `minutes` tillægges 1.

```
1 ISR(TIMER1_OVF_vect) {
2
3     //increments the time by 1 sec
4     myTime.seconds++;
5
6     //every 60 seconds updates Minutes value .
7     if (myTime.seconds > 58){
8         myTime.seconds = 0;
9         myTime.minutes++;
10    }
11
12    //every 5 seconds updates transmitters next action time .
13    if (myTime.seconds % 5 == 0){
14        myTime.myTransPtr->checkTime( myTime.minutes );
15    }
16 }
```

2.4.4 Transmitter klassen

Constructor

Til implementering af Transmitter klassen er der lavet en constructor, som initierer de to variabler `charCounter` og `scenCounter`. Uddover dette opretter den et array på 20 pladser, som den fylder med tomme `Action` objekter.

scenData(char input)

Denne metode er implementeret, så den tjekker hvor mange dele af en **Action**, der er gemt og når et helt **Action** objekt er overført, gemmes dette i objekt nummer **scenCounter**, som holder styr på hvor mange **Action** objekter der er overført. Når data gemmes oversættes den samtidigt til de kodemønstre, som X10 protokollen bruger.

checkTime(unsigned int theTime)

Denne metode er implementeres således at den kaldes sig selv rekursivt for at tjekke om der er flere aktioner oprettet på samme tidspunkt. Hvis dette er tilfældet fortsætter den med rekursive kald, indtil der ikke er flere aktioner til tidspunktet eller at alle aktioner i hele scenariet er blevet udført (i det tilfælde, hvor alle aktioner i et scenario ligger samtidigt).

```
1 void Transmitter::checkTime(unsigned int theTime){  
2     if (myScenario[nextAction].getTime() == theTime){ //Check to see if the time has  
3         past the next action to be sent  
4         if (firstLoop){ //Check if this is the first recursive call  
5             breakAt = nextAction; //Set the breaking point for recursive calls  
6             firstLoop = false;  
7         }  
8         else{  
9             if (breakAt == nextAction){ //Check if the breaking point has been reached  
10                 firstLoop = true;  
11                 return; //Break out of the recursive calls , if all the actions in the  
12                 Scenario have been sent simultaniously  
13             }  
14         }  
15         myTx10Ptr->sendAction(myScenario[nextAction]);  
16         nextAction++;  
17         if (nextAction == 20){ //If the last action has been sent , queue the first action  
18             nextAction = 0;  
19         }  
20         checkTime(theTime); //Recursive call , to check if several actions are on the  
21             same time.  
22     }  
23     else  
24         firstLoop = true;  
25 }
```

2.4.5 Tx10 klassen

Klassen fungerer ved at der hele tiden kommer interrupts fra **zeroCrossDetected** signalet, 100 interrupts i sekundet. Datamedlemmet **bool start** bestemmer om Tx10 klassen, skal sende noget ud på nettet eller ej.

Constructor

Constructoren sørger for at initialisere samtlige variabler til værdier, som giver mening. Den initialiserer Timer0, Timer2 og INT0 samt PORTC til debugging med LED-lysene, som sidder på STK500.

Timer0 bruges til at danne et 120 kHz clocksignal, til at beregne de initierede værdier er følgende udregninger lavet. Dette er når $f_{osc} = 3.6864\text{MHz}$, $N = 1$ og $OCR0 = 14$.

$$f = \frac{f_{osc}}{2N(1 + OCR0)} = 122.88\text{kHz}$$

unsigned char Translate(unsigned char bitCode)

Hjælpemetode, som oversætter 4-bit tal til en char indeholdende 8 nulgennemgange. Denne er implementeret med **if**-sætninger, som tjekker på de enkelte bits i **bitCode**.

```

1 unsigned char Tx10::translate( unsigned char bitCode ){
2     unsigned char temp = 0;
3     if (bitCode & 0b00001000){
4         temp |= 0b10000000;
5     }
6     else{
7         temp |= 0b01000000;
8     }
9     if (bitCode & 0b00000100){
10        temp |= 0b00100000;
11    }
12    else{
13        temp |= 0b00010000;
14    }
15    if (bitCode & 0b00000010){
16        temp |= 0b00001000;
17    }
18    else{
19        temp |= 0b00000100;
20    }
21    if (bitCode & 0b00000001){
22        temp |= 0b00000010;
23    }
24    else{
25        temp |= 0b00000001;
26    }
27    return temp;
28 }
```

sendAction(Action &)

Metoden starter med at gemme **Housecode**, **Unitcode** og **Command** fra aktionen i **Tx10** klassens egne variabler ved hjælp af **Translate()** og sætter **start** til TRUE.

sendCrossing(unsigned char send)

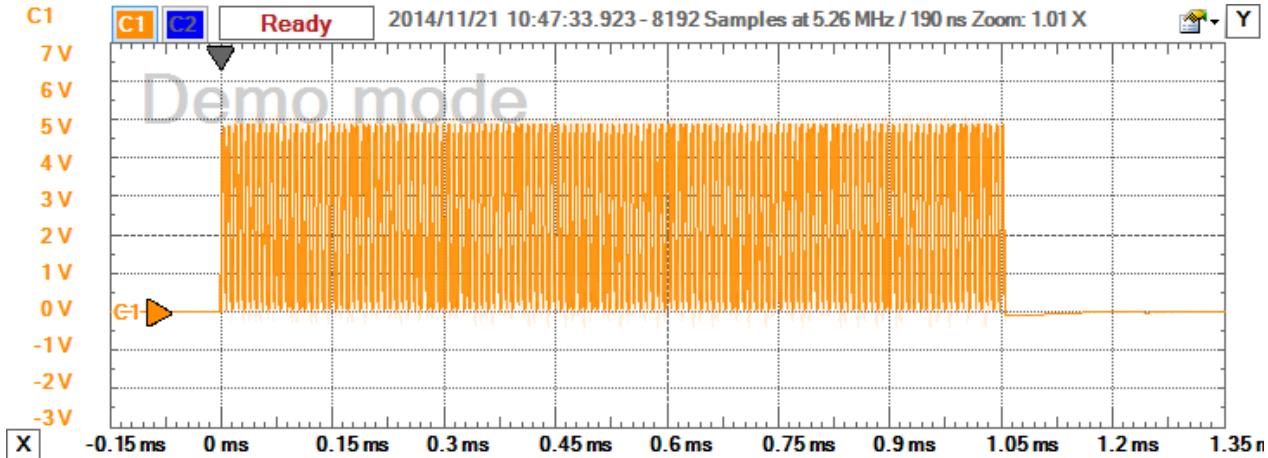
Hjælpemetode, som åbner op for 120 kHz firkantsignal, hvis send er forskellige fra 0.

Metoden er testet til at give outputet i Figur 18 på STK500 når den kaldes med en char forskellige fra 0.

Frekvensen på firkantsignalet er med Analog Discovery målt til 122 kHz.

waitMs()

Hjælpemetode, som starter timer2 på 1 ms og venter til denne er færdig inden metoden forlades. Tiden er regnet ud ud fra følgende formel:



Figur 18: Output på x10Data når en der skal sendes 1 i en nulgennemgang.

$$t = \frac{1}{f_{osc}} N(C - 1) = 1.00ms$$

Hvor N er sat til 64 og C er sat til 59. N er prescaleren på clockfrekvensen for ATMega32 og C er antallet af tællinger der skal til. Dvs at der skal skrives $255 - C$ til OCR2 registeret.

intHandler()

Hjælpemetode, som håndterer interrupts fra `zeroCrossDetected` benet.

Starter med at tjekke om `start` er TRUE, hvis den er det fyldes `buffer` variablen med de næste nulgennemgange, som skal sendes.

Starter med at sende MSB fra bufferen og skifte bufferen til venstre, indtil at bufferen er tom (`buffCount == 0`). Afhængigt af hvor mange mønstre (`pattCount`), som er sendt fylder den det næste mønster i bufferen. Når alle mønstre er sendt sætter den `start` til false.

```

1 void Tx10::intHandler() {
2     if (start){ //check if the Tx is supposed to transmit
3         if (buffCount == 0){
4             if (pattCount == 0){
5                 buffer = 0b11101111;
6                 buffCount = 4;
7             }
8             else if (pattCount == 1){
9                 buffer = house;
10                buffCount = 8;
11            }
12            else if (pattCount == 2){
13                buffer = unit;
14                buffCount = 8;
15            }
16            else if (pattCount == 3){
17                buffer = command;
18                buffCount = 8;
19            }
20            else if (pattCount == 4){
21                buffer = 0b00000011;
22                buffCount = 6;
23            }
}
```

```

24     else if (pattCount == 5){
25         buffer = house;
26         buffCount = 8;
27     }
28     else if (pattCount == 6){
29         buffer = unit;
30         buffCount = 8;
31     }
32     else if (pattCount == 7){
33         buffer = command;
34         buffCount = 8;
35     }
36     else if (pattCount == 8){
37         buffer = 0b11110000;
38         buffCount = 4;
39     }
40     else if (pattCount == 9){
41         start = false;
42         pattCount = 0;
43     }
44     pattCount++;
45 }
46
47 if (buffCount > 0){
48     sendCrossing((buffer & 0b10000000)); //Send out the MSB
49     buffCount--;
50     buffer = (buffer << 1); //shifts the buffer one time to the left
51 }
52 }
53 PORTC = ~getMyTx10()->buffer; //For debugging
54 }
```

turnAllOff()

Metode der fylder **house** variablen med 0 (koden for at slukke alt) og sætter **start** til TRUE. Dette medfører at ved de efterfølgende nulgennemgang sender mønstret for at slukke alt på nettet.

2.4.6 TxUART klassen

Denne klasse er overordnet implementeret til brug i projektet ved overførsel af scenarier, men bruges også til dels til debugging af de øvrige klasser.

Constructor

Constructoren sørger for at initiere **mode** til 'i' (idle) og initierer ellers UART på almindelig vis i henhold til protokollen.

rxInt()

Hjælpemetode, som håndterer interrupts når der modtages en **char** på UARTEn. Metoden er implementeret ved at den kontrollerer hvilken tilstand, klassen er i. Hvis den er i idle udføres den kommando, som er sendt via UART. Hvis den er i receiving mode og der ikke er send 80 chars endnu, sendes der data til **Transmitter** klassen.

```

1 void TxUART::rxInt() {
2     char input = UDR;
```

```

3  if (mode == 'i'){
4      if (input == 'L'){
5          bool temp = myTrans->getLockStatus();
6          if (temp)
7              sendChar('U');
8          else
9              sendChar('L');
10     }
11    else if (input == 'N'){
12        mode = 'r'; //put in receiving mode
13        myTrans->newScen();
14    }
15    else if (input == 'S'){
16        myTrans->stopAll();
17    }
18 }
19 else if (mode == 'r'){
20     if (charNo <= 80){
21         myTrans->scenData(input);
22         charNo++;
23     }
24     if (charNo == 80){ //end of data
25         charNo = 0; //reset counter
26         mode = 'i'; //put in idle mode
27     }
28 }
29 }
```

sendChar(char)

Hjælpemetode, sender en char via UART.

sendString(const char * sendMe)

Hjælpemetode til debugging, sender en række chars via UART, modtager "string literal", som de forekommer i C++. Dvs man fx kan skrive `sendString("Hello world.")`.

sendNumber(int sendMe)

Sender en integer via UART, som oversættes til ASCII-værdier først. Bruges primært til debugging af systemet.

```

1 void TxUART::sendNumber( int sendMe){
2     sendChar(((sendMe / 1000)%10)+48);
3     sendChar(((sendMe / 100)%10)+48);
4     sendChar(((sendMe / 10)%10)+48);
5     sendChar((sendMe)%10)+48);
6 }
```

2.5 Receiver

Receiveren modtager et signal fra ZeroCross detectoren, når der er et 120khz signal på nettet, og derefter læser den fra bitstreamen som indeholder dataene fra *Ref Treansmitteren?*

Klassen har en indbygget debugging-funktion, som udnytter RxUART klassen til at "sladre" om hvad der foregår. Debugging kan aktiveres ved at sætte konstanten `debugging` til `true`. Dette hjælper os rigtig meget under implementering da vi kan se helt precise, hvor langt den er nået i modtagelsen af data.

2.5.1 Rx10 klassen

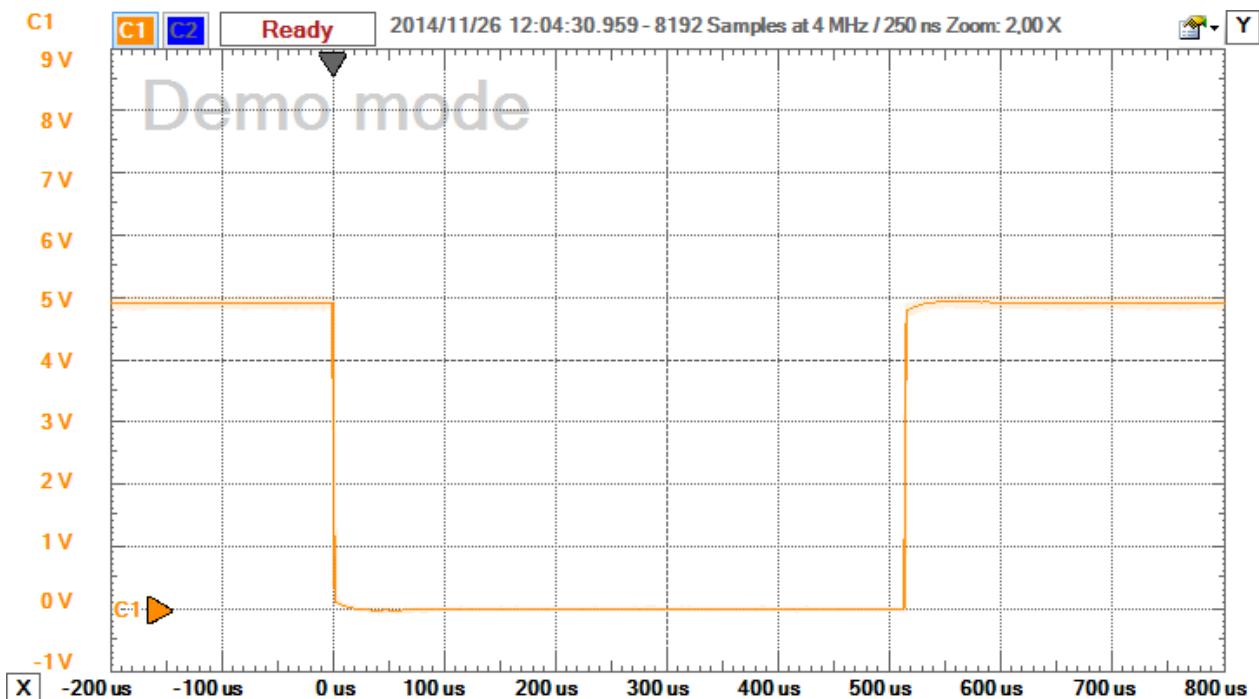
waitMs()

Denne hjælpemetode er til for at være sikker på at ramme midt i dataen for en given nulgennemgang. Da disse er 1 ms lange, vil vi læse 0.5 ms efter nulgennemgangen. På denne måde er vi mest muligt sikre på at ramme dataen, hvis nu timingen skulle skride en smule.

Metoden fungerer ved at starte timer2 på 0.5 ms, dette er regnet ud ud fra følgende udregninger:

$$\begin{aligned} t &= \frac{1}{f_{osc}} N(C - 1) \\ &= \frac{1}{3.6864\text{MHz}} 64(29 - 1) \\ &= 0.5\text{ms} \end{aligned}$$

Metoden er testet ved at sætte et ben lavt før og højt igen efter metoden og dette er målt med Analog Discovery i Figur 19. Det ses at timingen passer meget godt med 0.5 ms.



Figur 19: Måling af waitMs() metoden.

interruptHandler()

Hjælpemetode til at håndtere interrupts fra `zeroCrossDetected`.

Metoden tjekker i første omgang på hvert interrupt om den har modtaget start-koden fra X10 protokollen (se side X).

```
1  waitMs(); // wait 0.5 ms
2  buffer = buffer << 1; // Make rooms the next bit
3  buffer |= ( PINA & 0b00000001 ); //Adds the bit in the buffer
4  PORTC = ~buffer;
5
6  if (!start){
7      if ( ( buffer & 0b00011111 ) == 0b00001110 ){
8          start = true;
9          pattCount = 2;
10         buffCount = 7;
11         if (debugging){
12             myRxUART->sendString("New command!\n\r");
13         }
14     }
15 }
```

Listing 2.1: Der ventes 0.5 ms og der tjekkes efter startbit.

Når start-koden er fundet, sættes `start` til TRUE og `pattCount` samt `buffCount` initieres. De efterfølgende otte nulgennemgange fyldes bufferen med data fra `bitstream`. Herefter tjekkes det om dataen er valid i forhold til protokollen (så at en høj nulgennemgang efterfølges af en lav og vice versa). Er protokollen overholdt oversættes dataen og gemmes i en variabel.

```
1 if (checkData(buffer)){
2     if (pattCount == 2){ // checks if it is in pattern house
3         house = translate(buffer);
4         buffCount = 7;
5
6         if (debugging){
7             myRxUART->sendString("HouseCode: ");
8             myRxUART->sendNumber(translate(buffer));
9             myRxUART->sendString("\n\r");
10        }
11    ...
12 }
13 ...
14 else{
15     start = false;
16     if (debugging){
17         myRxUART->sendString("Invalid data!\n\r");
18     }
19 }
```

Dette gentages for tre fyldte buffere (24 interrupts/nulgennemgange), hvorefter der forventes 6 tomme nulgennemgange. Er dette også iorden kontrolleres de næste 24 nulgennemgange om de matcher de foregående, som indeholdte data omkring Actionen. Hvis disse også er okay, ventes der på en slutkode, som beskrevet i protokollen. Er der fejl i blot én af ovenstående check kasseres alt og der ventes igen på start-koden.

Under processen overvejede vi også at lave en form for fejlhåndtering, da dette ikke giver mening i forhold til at vi kun har énvejskommunikation og kun dobbelttjekker data. Vi ville ikke risikere at kommandoer gik igennem og blev fortolket forkert.

translate(unsigned char zeroCrossingCode)

Er implementeret ved at tage de bits vi er interesserede i og gemme dem over i en lokal variabel, som herefter returneres.

```
1 unsigned char Rx10::translate( unsigned char zeroCrossingCode ){
2     char temp = 0;
3     temp = temp | (0b00000001 & (zeroCrossingCode >> 1) ); // Shifts bit 1 into to temp
4     temp = temp | (0b00000010 & (zeroCrossingCode >> 2) ); // Shifts bit 2 into to temp
5         and so on.
6     temp = temp | (0b00000100 & (zeroCrossingCode >> 3) );
7     temp = temp | (0b00001000 & (zeroCrossingCode >> 4) );
8     return temp;
}
```

2.5.2 Receiver control klassen

Receiver klassen fungerer ved, at Rx10 klassen kalder en funktionen *Input()*, med 3 chars, (*houseCode*, *unitCode* og *command*) som parametre. Først tjekker funktionen om *houseCode* er lig nul, da dette er en global commando som betyder sluk alt. Dernæst tjekker den om *houseCode* og *unitcode* passer til den respektive receiver. Hvis disse passer til receiveren, vil funktionen, afhængig af kommandoen kalde en funktion i lampe klassen via en pointer til denne.

