Home Protection

Semesterprojektgruppe 5

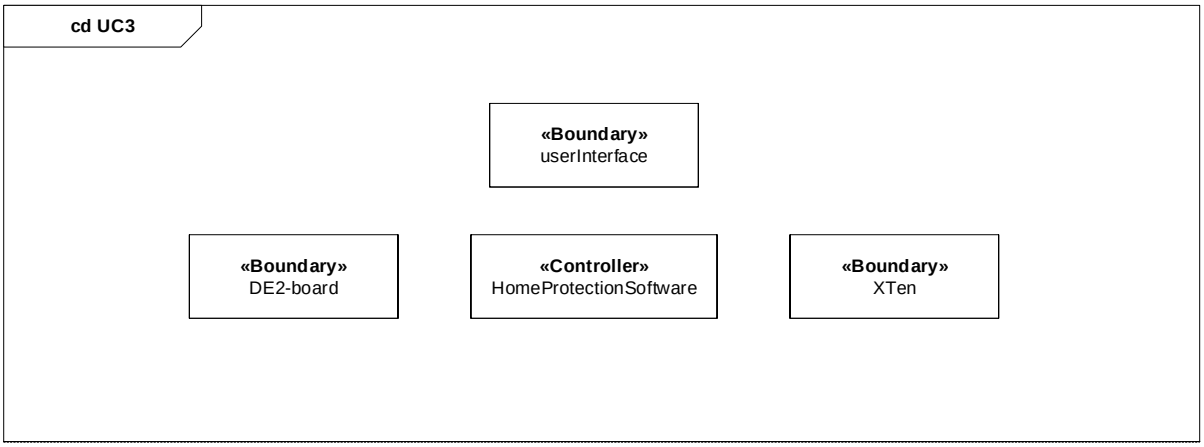
Software Dokumentation

Indholdsfortegnelse

SWA1 - Applikationsmodel:

Use Case 3:

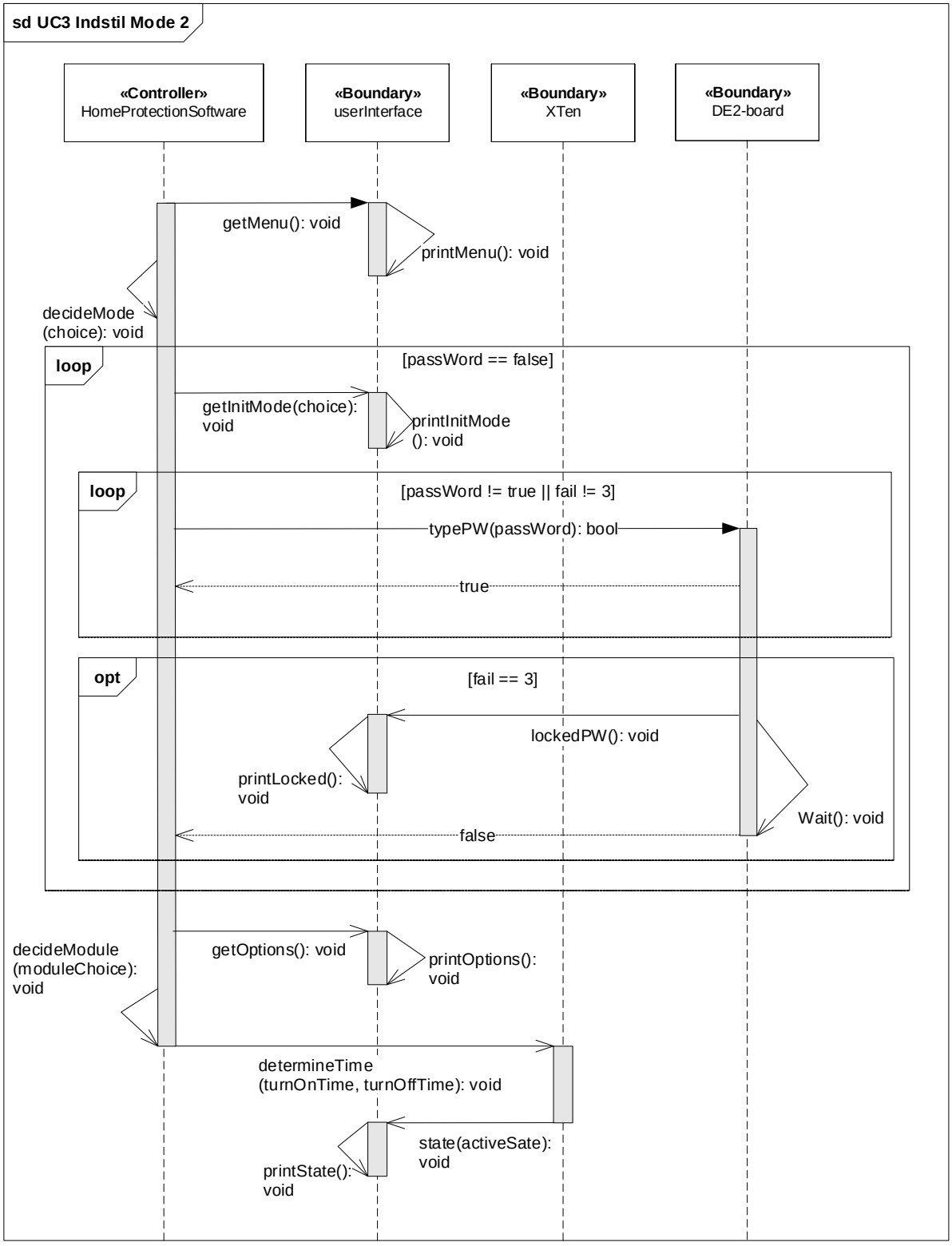
I Use Case 3 vælger brugeren ”Definer mode 2”, hvor brugeren bliver bedt om at indtaste en kode på DE2-boardet. Når den korrekte kode er indtastet, kan man ændre indstillingerne på mode 2. Nedenstående laves klassediagram med udgangspunkt i Use Case 3, bestående af controller klasse ”HomeProtectionSoftware” og boundry klasserne ”userInterface”, ”XTen” og ”DE2-board”:



Figur 1 – Klassediagram for Use Case 3

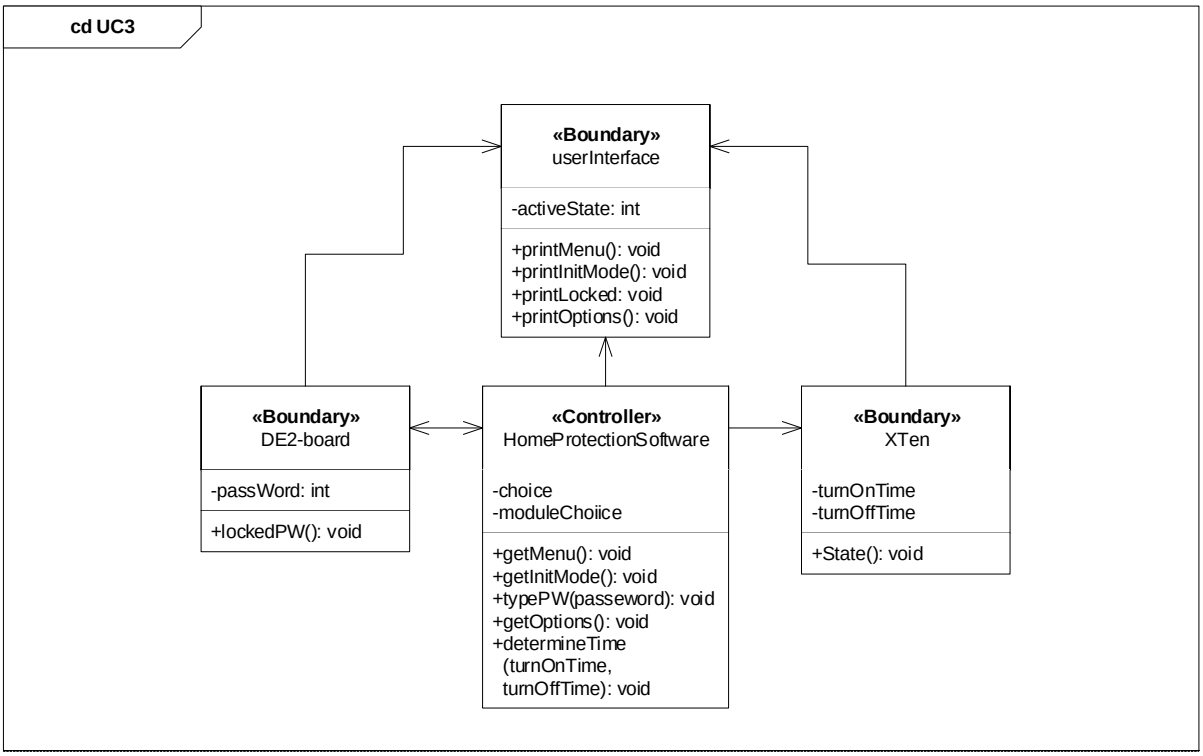
Systemet udskriver menuen for brugeren på userInterface, hvoraf brugeren vælger ”Definer mode 2”. HomeProtectionSoftware skal sørge for at de korrekte funktioner bliver initialiseret for Use Case 3, og vil derfor bede brugeren om at indtaste en kode på DE2-boardet. DE2-boardet består af nogle switches og keys, som skal benyttes for at få adgang til at ændre indstillingerne i mode 2. Superbrugeren indtaster kode for at få adgang. Hvis koden indtastes forkert 3 gange låses systemet. Og der skal herefter trykkes på reset knappen på de2-boardet, før Superbrugeren kan forsøge igen. Hvis koden indtastes korrekt, kan brugeren hermed ændre indstillingerne. Systemet initialiseres og videregiver informationerne til XTen, der indstiller dimmer og switch

Følgende sekvensdiagram på figur 14 viser funktionskaldene mellem de forskellige moduler i applikationen:



Figur 2 - Sekvensdiagram for Use Case 3

På nedenstående figur vises et opdateret klassediagram med funktioner og attributter. Her ses hvordan funktionerne i de forskellige klasser hører sammen:



Figur 3 - Klassediagram med members for Use Case 3

SWA2 - Klassebeskrivelser:

main.cpp (arduino_sender)

- Main.cpp for SA, har til opgave at opstille den serielle forbindelse mellem computer og arduino, samt initialisere interrupt og CTC signaler og opstille objektet startMode, der er af klassen Transmitter. Main kalder da chooseMode() fra Transmitter.cpp. Herefter ventes der på fuldendelse af chooseMode(). Efter fuldendelse vil main.cpp hoppe tilbage, så chooseMode() igen bliver kaldt.

Transmitter.cpp:

- Trasmitter.cpp har til formål at bestemme hvilket mode der skal udføres, via et kald fra UI: Mode 1, Mode 2, Change mode 2, deactivate Home Protection. Efter valg af mode, skal systemet være i stand til at videresende data fra valgte mode, med mindre ”Change Mode 2” er valgt. Her vil UI bede om oplysninger om ændring fra brugeren.

UI.cpp

- UI.cpp har til formål at kommunikere med brugeren, via PuTTy. Den vil give oplysninger, så som menu, aktiv mode og information om transmittering af data. Den vil yderligere bede om inputs fra brugeren, den returner til Transmitter.cpp.

ModeTwoSettings.cpp

- ModeTwoSettings.cpp har til formål at ændre værdierne for Mode 2. Dette gøres ved input fra Transmitter.cpp, der bestemmer hvordan Home Protection skal opføre sig. Den opbygges i tre dele, to arrays der indeholder 2 bits og et der indeholder 4 bits. De to bits fra de to arrays, bestemmer henholdsvis tænd og sluk. Arrayet med 4 bits bestemmer intensiteten (PWM) af lysstyrken på lampen. Bits for tænd og sluk, samles i arrays fra det respektive array fra Mode 2, hvor plads 2 og 3 bliver udskiftet og svare til tænd og plads 4 og 5 bliver udskiftet og svare til sluk. Arrayet for lysstyrken (PWM) ændres, hvor værdierne på plads 2-5 bestemmer lysstyrken for lampen.

main.cpp (arduino_receiver)

- main.cpp for Arduino receiveren starter med at initialisere interrupt og PWM-signaler. Herefter indgår den i en løkke, der vil bruges til at indsætte værdier i et array, loadBits, fra signaler der modtages fra SA. Efter arrayet er blevet fyldt, vil værdierne blive indsat i sendBits. Dette gøres af 2 grunde. 1. arrayet loadBits vil konstant opdateres, her vil send bits kun blive opdateret når loadBits er klar med nye værdier. Yderligere, er loadBits af typen volatile, hvilket skaber problemer af videresendelse af arrayet. Arduinoen videresender arrayet sendBits til inputReader().

  Forskellen på main.cpp for dimmer og switch, består af at dimmer skal modtage 14 værdier, mens switch kun skal modtage 7. Dimmer skal modtage værdier, både for tænd og sluk, men også intensitet af lys.

Receiver.cpp

- Receiver.cpp består af fire funktioner. En til at initialisere interrupts (initInterrupt()), en der initialisere PWM (initPWM), en der kan konvertere binære tal til decimal tal (binToDec()), og en der, ud fra decimaltal værdierne, kan bestemme start og sluk tidspunkt for dimmer og switch, samt lysstyrken for lampen (inputReader()). inputReader() vil yderligere analysere startbit, for at tjekke om det er det relevante array der skal læses. Efter bestemmelse, vil arduinoen udføre sit tildelte job, hvor der ventes på start, bestemmes intensitet og ventes på stop.

  Forskellen på Receiver i dimmer samt i switch, består af at dimmer indeholder værdier for både tænd, sluk og intensitet, hvilket giver den 8 værdier der skal processeres, i henholdsvis 2 arrays, med størrelsen 7. Switch indeholder kun 4 værdier, der alle er samlet i et array af størrelsen 7. Dimmeren vil skulle omsætte 8 binære værdier (tænd (2 bits), sluk (2 bits), intensitet (4 bits)), mens switch kun skal omsætte 4 (tænd (2 bits), sluk (2 bits)).

SWA3 – Software Modulbeskrivelse:

I tabellen nedenfor findes modulbeskrivelser af hver klasse med member functions og parametre:

Klasse: Transmitter	
Funktion:	**Indeholder/Benytter:**
Transmitter(): Transmitter	modeType: char settings: char inputCheck: bool
Constructur for klassen Transmitter. Sætter variablerne modeType, settings og inputCheck.	
Funktion:	**Indeholder/Benytter:**

chooseMode(): void	-Settings: char -modeType: char -inputCheck: bool -modeOneDimmer: char[] -dimmerPWM: char[] -modeOneSwitch: char[] -modeTwoDimmer: char[] -modeTwoSwitch: char[] -printMenu(): char -printInit(): bool -runMode(char[]) : void -printSettigs(): void -setDimmerOn(): void -setDimmerOff():void -setDimmerPWM(): void -setSwitchOn(): void -setSwitchOff(): void -collectMode(): void -printActive(): void
chooseMode() har som funktion, at vælge hvilken mode der skal køres. Dette gør den ved hjælp af printMenu() funktionen, der fremviser en menu, hvor brugeren kan vælge et tal mellem 1-4. Herefter vil printInit() blive kørt, der tjekker om værdien er korrekt, og herefter printer, hvilket mode man har ønsket. Ved forkert input, vil printMenu() kaldes igen, og der vælges ny værdi. Ved tast af 1, vil Mode 1 blive kørt. Her vil runMode(char[]) blive kaldt 3 gange, en med vært array tilknyttet Mode 1. Samme vil ske ved tast 2, hvor Mode 2 vil kører på samme måde som mode et, dog med de arrays der er tilknyttet Mode 2. Ved tast af 3, vil der blive kaldt printSettings(). PrintSettings() vil fremvise fire menuer for indstilling af mode 2. Dette gøres ved kald af funktionerne setDimmerOn(), setDimmerOff(), setDimmerPWM(), setSwitchOn() og setSwitchOff(). Herefter samles arraysne for mode 2, ved hjælp af funktionen collectMode(). Sidst vil der blive kaldt funktionen printActive(), der fremviser hvilket mode er blevet aktiveret. Herefter slutter chooseMode().	
Funktion:	**Indeholder/Benytter:**
runMode(char[]): void	sendBits: char[] printWaiting(): void printBitSent(): void
runMode() står for at sende et binært tal indeholdt i et array som machester-kode. Funktionen benytter en for-løkke til først at checke om en plads i arrayet indeholde et 1 eller et 0. Derudfra indstilles interrupt til enten rising eller falling edge. En while-løkke, hvori printWaiting() køres, benyttes til at vente på at et interrupt er indtruffet, hvorefter for-løkken foretager samme procedure med det næste element.	

Klasse: initArduino	
Funktion:	**Indeholder/Benytter:**
initInterrupt(): void	
Her initialiseres interrupt registrene. Vi benytter her INT4, indstillet til rising edge som default.	

Funktion:	Indeholder/Benytter:
initCTC(): void	
Her initialiseres CTC. Vi benytter OC3A, indstillet til mode 4, CTC. Clocken er sat til Match Down Counting, med en prescaler på 1.	
Funktion:	**Indeholder/Benytter:**
initPortB(): void	
Initialisere Port B LED'er på Arduino shield til output.	
Funktion:	**Indeholder:**
initUART(): void	
Initialiserer UART. Der indstilles til normal ,asynchronious mode med 1 stop bit.	

Klasse: modeTwoSettings	
Funktion:	**Indeholder/Benytter:**
setDimmerOn(char): void	Settings: char dimmerOn: char[2]
SetDimmerOn(char) modtager fra chooseMode() variablen settings, med en værdi mellem 1 – 4. SetDimmerOn består af en switch, der styres af settings, der vær har en værdi på given plads i dimmerOn[]. Arrayet dimmerOn[] sættes til værdier tilsvarende given setting værdi.	
Funktion:	**Indeholder/Benytter:**
setDimmerOff(char): void	Settings: char dimmerOff: char[2]
serDimmerOff(char) har samme funktion som setDimmerOn(). Det er dog i arrayet dimmerOff der ændres på.	
Funktion:	**Indeholder/Benytter:**
setDimmerPWM(char[], char): void	Settings: char dimmerPWM: char[4]
setDimmerPWM(char[], char) modtager en pointer til arrayet dimmerPWM og variablen settings der ligger mellem 1-10. Her er der, ligesom for setdimmerOn en switch, der bestemmes af settings. Forskellen ligger på at der nu er 10 muligheder, og dimmerPWM ændres direkte via en pointer og at der er 4 værdier der ændres på: plads 2 til 5.	
Funktion:	**Indeholder/Benytter:**
setSwitchOn(char): void	Settings: char switchOn: char[2]
setSwitchOn(char) har samme funktion som setDimmerOn(). Det er dog i arrayet switchOn der ændres på.	
Funktion:	**Indeholder/Benytter:**
setSwitchOff(char): void	Settings: char switchOff: char[2]
setSwitchOff(char) har samme funktion som setDimmerOn(). Det er dog i arrayet switchOff der ændres på.	
Funktion:	**Indeholder/Benytter:**
collectMode(char[] ,char[]): void	modeTwoDimmer: char[] modeTwoSwitch: char[] dimmerOn: char[2] dimmerOff: char[2]

	switchOff: char[2] switchOn: char[2]
collectMode(char[] ,char[]) sørger for at modtage 2 pointers til modeTwoDimmer og modeTwoSwitch. Her vil collectMode(char[] ,char[]) indsætte værdierne der befinder sig i arraysne dimmerOn, dimmerOff, switchOn og switchOff. Værdierne for ”on” arraysne, vil indsættes på plads 2-3 og værdierne for ”off” arraysne vil indsættes på plads 4-5, på deres respektive pladser i modeTwoDimmer og modeTwoSwitch.	

Klasse: UI	
Funktion:	**Indeholder/Benytter:**
printMenu(): char	inputOne: char serialFlush(): void
Funktionen skal printe en menu ud i vores konsol, og tage en input som den retunere. Funktionen serialFlush() køres for at forberede et input. En menu prints vha. Serial.println() funktionen, hvorefter Serial.read() bruges til at indlæse et input ind i variablen inputOne. Hvis inputOne ikke er lig '0', bliver konsolen med Serial.write().	
Funktion:	**Indeholder/Benytter:**
printSettings(): char	Module: char inputTwo: char
printSettings(): char skal printe en undermenu i konsolen. Variablen module bliver indsat i en switch()	
Funktion:	**Indeholder/Benytter:**
printInit(): bool	inputOne: char printWrongInput(): void
Benyttes til at printe en besked i UI’en ud fra hvilken mode der er blevet valgt i menuen. Sætter desuden PortB, så LED’erne på Arduino shield passer til den valgte mode.	
Funktion:	**Indeholder/Benytter:**
enterCode(): void	timer: int serialFlush(): void
Her bedes brugeren indtaste koden på DE2-boardet, så systemet kan låse op for ”mode 3”.	
Funktion:	**Indeholder/Benytter:**
printWaiting(): void	waitValue: char
Sender en besked ud til konsol vindue, der viser at systemet venter på at kunne sende bits videre til arduino modtageren.	
Funktion:	**Indeholder/Benytter:**
printActive(): void	inputOne: char inputTwo: char waitValue: char module: char
Resetter konsollen I Putty. Printer desuden i UI konsol afhængig af hvilken mode der er blevet aktiveret. Slukker for LED’er på Arduino shield.	
Funktion:	**Indeholder/Benytter:**
printBitSend(): void	waitValue: char
Printer en besked I UI konsolen hver gang en bit bliver sendt.	
Funktion:	**Indeholder/Benytter:**

printWrongInput(): void	
Printer en besked I UI konsolen der indformere om ugyldigt input.	
Funktion:	**Indeholder/Benytter:**
serialFlush(): void	
Renser systemet for inputs via PuTTy. Ved få tilfælde, vil systemet læse ekstra indtastninger, som f.eks. ”enter”. serialFlush() sørger for at disse inputs ikke går videre, men bliver ” skyllet” ud. Herved er der gjort klar til næste ønskede input.	

Idet en stor del af funktionaliteten af vores system afhænger af brugen af avr interrupts, finder vi det relevant også at indkludere en modulbeskrivelse af interrupt. Det er her klart for os at INT4 ikke er en klasse og ISR(INT4_vect) ikke er en funktion:

Klasse: INT4	
Funktion:	**Indeholder/Benytter:**
ISR(INT4_vect)	
Interruptet står for at indstile CTC til det 120kHz burst vi ønsker. Registret OCR3A gives værdien 65 for at starte outputtet på 120kHz. EICRB sættes til low-input, for at forberede regsitret til enten at sende ved rising edge eller falling edge. Et delay på 1ms køres for at sikre at burst-timeren passer overens med zeroCrossing, hvorefter global interrupt bliver disabled.	

SWA4 - Testkode for modultest

Test af count/interrupt:

Ved brug ISR(INT2_vec) interrupt, vil vi kunne teste om interruptet er aktivt og virker. Ved vært interrupt, vil count stige med 1, hvilket kan ses på arduino sheildets LED'er.

```
1.  while (count < 7) {
2.        switch (count) {
3.                    case 1: PORTB = 0b00000001;
4.                    break;
5.                    case 2: PORTB = 0b00000010;
6.                    break;
7.                    case 3: PORTB = 0b00000100;
8.                    break;
9.                    case 4: PORTB = 0b00001000;
10.                   break;
11.                   case 5: PORTB = 0b00010000;
12.                   break;
13.                   case 6: PORTB = 0b00100000;
14.                   break;
15.                   case 7: PORTB = 0b01000000;
16.                   break;
17.                   case 8: PORTB = 0b10000000;
18.                   break;
19.       }
20. }
```

Test af loadBits/sendBits

For at teste om der er kommet nogle værdier ind i loadBits (og derfor også sendBits) sættes der et for loop til at køre igennem alle værdierne. Siden værdierne for loadBits og sendBits skal være ens, kan de begge testes på samme tid. De værdier der sættes ind, kommer fra count – plads 0 = count = 0, plads 1 = count = 1 osv.

```
1.  for (int n = 0; n < 7; n++) {
2.  switch (sendBits[n]) {
3.        case 0: PORTB = 0b00000000;
4.        break;
5.        case 1: PORTB = 0b00000001;
6.        break;
7.        case 2: PORTB = 0b00000010;
8.        break;
9.        case 3: PORTB = 0b00000100;
10.       break;
11.
12. }
13.
14. switch (loadBits[n]) {
```

```
15.         case 4: PORTB = 0b00001000;
16.         break;
17.         case 5: PORTB = 0b00010000;
18.         break;
19.         case 6: PORTB = 0b00100000;
20.         break;
21.         case 7: PORTB = 0b01000000;
22.         break;
23.
24.     }
25.
26. }
```

Test af inputReader()

For at teste om inputReaderen er aktiv, vil alle LED'erne på arduinoen lyse. Herved ved vi at funktionen bliver kaldt.

```
1. PORTB = 255;
2. delay(50000);
```

Test af binToDec()

Ved brug af et array, kan vi teste om binToDec omskriver de binære værdier om til dec. Decimaltallet vil da vises på arduinoens LED'er

```
1. char binToDecTest[7] = {0, 1, 1, 0, 1, 1, 0};
2. PORTB = binToDec(loadBits, 2, 3);
3. delay(100000);
```

Test af runMode()/interrupt

For at test om runMode() kan køre, kræves det at simulere et interrupt. Dette kan gøres via arduino sheildet, da vi kan gøre brug af ISR(INT2_vec) der kan aktiveres af SW2 knappen. Herved kan vi se, om koden faktisk for sendt bits via interrupts.

```
1. void Transmitter::runMode(char sendBits[]) {
2.
3.         for (bits = 0; bits < 7; bits++) {
4.                     sei(); //global interrupt enable
5.                     if (sendBits[bits] == 1) {
6.                                 EICRB = 00000010;
7.                     }
8.                     else if (sendBits[bits] == 0) {
9.                                 EICRB = 00000011;
10.                    }
11.                    while (active == 0) {ui.printWaiting();}
12.                    active = 0;
```

```
13.                 ui.printBitSent();
14.     }
15. }
```

Test af enterCode()

Test af enterCode() simuleres ved at lave et kode system på arduinoen, da det ikke er muligt at bruge et DE2-board. Koden skal skrives ind af brugeren. Hvis der tastes rigtigt, vil programmet udskrive ”Correct code!” og fortsætte. Tastes der forkert, vil programmet udskrive ”Wrong code! Try again!”, og brugeren skal prøve igen.

```
1.  void UI::enterCode() {
2.      this->serialFlush();
3.
4.      int lockValue = 0;
5.
6.      Serial.print("Please enter your test code: ");
7.
8.      while (lockValue == 0) {
9.              if (Serial.available() > 0) {
10.
11.                     lockValue = Serial.read();
12.
13.             }
14.     }
15.
16.     if (lockValue == '5') {
17.             Serial.println("Correct code!");
18.             lockValue = 0;
19.     }
20.
21.     else {
22.             Serial.println("Wrong code! Try again!");
23.             this->enterCode();
24.     }
25.
26. }
```

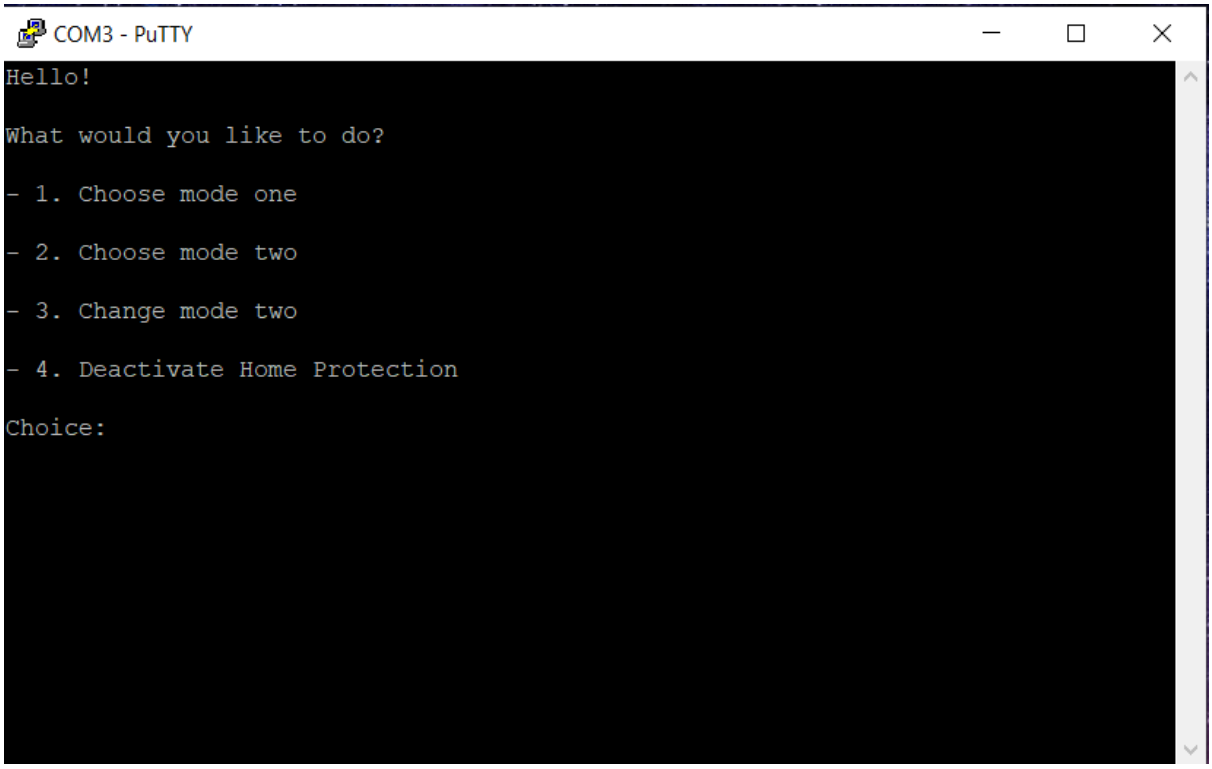
Ændringer af Mode two

Ved at se på ændringerne af Mode two, vil vi printe de værdier der vil ligge i arrayet for Mode two. Arrayet har som standart de samme værdier som Mode one. Det der sker er, at der bruges Mode three, til at omskrive Mode two. Efter omskrivning, vil vi printe værdierne for Mode one (De værdier Mode two startede med) og værdierne for Mode two. Hvis værdierne ikke er ens, er der altså blevet ændret på Mode two.

```
1.  /*
2.  modeOne dimmer: 0, 0, 0, 1, 1, 1, 1
```

```
3.  modeTwo dimmer: 0, 0, 0, 0, 0, 0, 1
4.  */
5.      for (int test = 0; test < 7; test++) {
6.                  if (modeOneDimmer[test] == 1)
7.                  Serial.print("1, ");
8.
9.                  else if (modeOneDimmer[test] == 0)
10.                 Serial.print("0, ");
11.     }
12.
13.     Serial.println("\n");
14.
15.     for (int test = 0; test < 7; test++) {
16.                 if (modeTwoDimmer[test] == 1)
17.                 Serial.print("1, ");
18.
19.                 else if (modeTwoDimmer[test] == 0)
20.                 Serial.print("0, ");
21.     }
22.
23. }
```

SWA5 - PuTTy UI



SWA6 – DE2-board

Systemet er forbundet til et Altera DE2-board, som anvendes i forbindelse med Use Case 3, hvor brugeren indstiller mode 2. Når brugeren vælger ”Definer Mode 2”, beder systemet om at få en kode fra superbrugeren. Som tidligere nævnt er koden anvendt for at, det ikke er alle der har adgang til denne funktion. Det er dermed superbrugeren der skal indtaste en kodesekvens på DE2-Boardet for at låse ”Definer Mode 2” op. DE2-boardet skal heraf bestå af en code_lock, UART og baudrate generator. Da der tidligere i DSD-kurset er blevet lavet journalopgaver med UART og code_lock, kan disse anvendes som skabelon i dette tilfælde. Koden skrives i VHDL, da det er det sprog, der er blevet anvendt til at arbejde med i forbindelse med FPGA-boardet.

Baud_Rate_Generator

Baud.vhd

- Baud.vhd består af entiteten og arkitekturen for Baud_Rate_Generator. Formålet med denne implementering er at man selv kan indstille en passende hastighed, for hvor hurtigt data skal overføres fra vores transmitter.

Baud_Rate_Generator implementeres i de to følgende kodeafsnit, hvor entiten og arkitekturen er opbygget.

```
entity Baud_Rate_Generator is
generic(
   scale : natural := 41666;
   MIN_CLK_VAL : natural := 0;
   MAX_CLK_VAL : natural := 50000000;
   baud        : natural := 41666 -- BAUD-Rate på  1200 Hz
   );
   port(
   reset, clk  : in std_logic;
   clk_baud    : out std_logic
   );
end;
```

Figur 4 - Entity for Baud_Rate_Generator

$$\frac{50000000\ Hz}{41666} \approx 1200{,}019\ Hz$$

Ved brug af generic, kan baudraten indstilles til en værdi som ønsket. Denne værdi skal hænge sammen med reciever enhedens læserate, så der ikke opstår malplacerede bits. Under test af implementeringen af code_lock er der benyttet en baudrate på 1200 Hz, der gør det nemmere og mere overskueligt at læse de bit der udsendes.

```
architecture arch of Baud_Rate_Generator is
begin
   counting : process(clk, reset)
   variable clk_counter : integer range MIN_CLK_VAL to MAX_CLK_VAL;
   variable clear       : std_logic;
   begin
      if reset = '0' or clear = '1' then
         clk_counter := 0;
      elsif rising_edge(clk) then
         clk_counter := clk_counter + 1;
      end if;

      if clk_counter = baud then
         clk_baud <= '1';
         clear := '1';
      else
         clk_baud <= '0';
         clear := '0';
      end if;
   end process counting;
end;
```

Figur 5 - Architecture for Baud_Rate_Generator

I arkitekturen anvendes der to yderligere variable, clk_counter og clear. Disse benyttes i forb. med if-else og elsif-sætninger der sammen har til formål at sætte værdien af clk_baud, som er output og bliver benyttet af UART_transmitteren. Det kan ses, at hvis der opstår rising_edge(clk) gentagne gange, så bliver clk_counter = baud hvorefter clk_baud sættes '1', og kort efter cleares den og clk_counter samt clk_baud sættes tilbage til '0'. clk_baud benyttes i efterfølgende afsnit om UART_transmitter.

UART_transmitter

UART_transmitter.vhd

- Formålet med UART_transmitteren er at der kan oprettes UART forbindelse mellem DE2-Board og Arduino, hvorved der kan sendes et antal databit, der skal initiere en ønsket mode på arduino'en.

UART_transmitteren implementeres med en entitet og arkitektur for denne, det kan ses i de følgende kodeafsnit

```
entity UART_transmitter is
port(
   --input
   reset, txvalid, clk_baud   : in std_logic;
   txdata                     : in std_logic_vector(7 downto 0);
   --output
   txd                        : out std_logic
);
end;
```

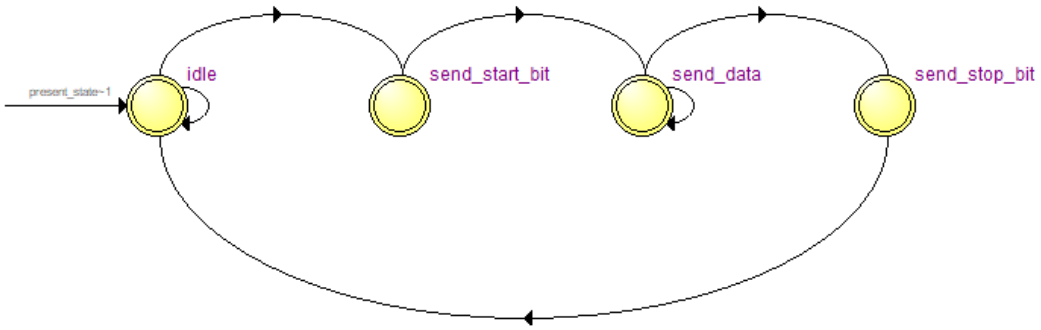
Figur 6 - Entity for UART_transmitter

Ovenfor ses entiteten UART_transmitter, denne indeholder in-/output-porte, som vi specificere i arkitekturafsnittene nedenfor samt bruges disse porte i vores samlede .vhd fil "code_lock_uart"

```
architecture arch of UART_transmitter is

type state is (idle, send_start_bit, send_data, send_stop_bit);
signal next_state, present_state : state;
signal latch_present     : std_logic_vector(7 downto 0);
signal latch_next        : std_logic_vector(7 downto 0);
signal bit_cnt_present   : integer range 0 to 8 ;
signal bit_cnt_next      : integer range 0 to 8;
```

Figur 7 - Architecture for UART_transmitter (del 1)

Første arkitekturafsnit ses ovenfor, hvor der implementeres 4 states, der udgør en samlet state machine, kan ses på Figur 8.



Figur 8 – State Machine View for UART_transmitter

I ovenstående state machine illustreres funktionaliteten af UART_transmitteren. Der består af de fire states (idle, send_start_bit, send_data og send_stop_bit) disse initieres vha. koblingen mellem input og signalerne (latch_present, latch_next, bit_cnt_present, bit_cnt_next samt state signalerne next_state og present_state).

```
begin
   state_reg: process(clk_baud, reset)
   begin
      if reset = '0' then
         present_state <= idle;
      elsif rising_edge(clk_baud) then
         present_state <= next_state;
         bit_cnt_present <= bit_cnt_next;
      end if;
   end process;

   nxt_state: process(present_state, txvalid, bit_cnt_present)
   begin
   next_state <= present_state; --default
   bit_cnt_next <= bit_cnt_present; --default
   case present_state is
      when idle =>
         if txvalid = '1' then
            bit_cnt_next <= 0;
            next_state <= send_start_bit;
         end if;
      when send_start_bit =>
         next_state <= send_data;
      when send_data =>
         if bit_cnt_present = 7 then
            next_state <= send_stop_bit;
         else
            bit_cnt_next <= bit_cnt_present + 1;
         end if;
      when send_stop_bit =>
         next_state <= idle;
      when others =>
         null;
   end case;
   end process;

   mealy_out: process(present_state, bit_cnt_present, txdata)
   begin
   case present_state is
      when send_start_bit =>
         txd <= '0';
      when send_data =>
         txd <= txdata(bit_cnt_present);
      when send_stop_bit =>
         txd <= '0';
      when others =>
         txd <= '1';
   end case;
   end process;
end;
```

Figur 9 - Architecture for UART_transmitter (del 2)

I ovenstående kodeafsnit ses den anden del af arkitekturen for UART_transmitteren, her ses 3 processer (state_reg, nxt_state og mealy_out).
Den første proces tjekker hvornår der opstår rising_edge fra baud_rate clocken(clk_baud) som blev implementeret i afsnittet om Baud_Rate_Generatoren, ved rising edge initieres process ”nxt_state”. Derudover tjekker state_reg også om der trykkes på reset-knappen, hvorefter present_state så sættes til idle
Den anden proces ”nxt_state” indeholder funktionaliteten af de 4 states vist på figur 8. Når present_state er ”idle”, så skal txvalid gå høj = ’1’ før den næste state, send_start_bit, initieres og hele processen for transmitteren aktiveres. Send_start_bit aktiveres og efter startbit er sendt bliver værdien for txdata transmitteret under send_data som kan ses i 3. process ”mealy_out”. Som default vil transmitteren sende et ’1’ signal ud, initieres sekvensen af state machinen korrekt vil den sende ’0’ ud.

Code_lock_err:

Code_lock_err.vhd

- Code_lock_err.vhd er essensen bag vores code_lock_uart. Formålet med dette afsnit er, at man kan bruge DE2-Boardet som kodelås. Code_lock_err er implementeret således, at der skal indtastes en kodesekvens på DE2-Boardet, som skal ved korrekt kode låse op for kodelåsen for at give superbrugeren adgang til at rette i indstillingerne på dimmeren og switchen, dette vil blive uddybet i afsnittet om code_lock_uart.

```
entity code_lock_err is
port (clk, reset, enter : in std_logic;
      pin : in std_logic_vector(3 downto 0);
      lock : out std_logic);
end code_lock_err;
```

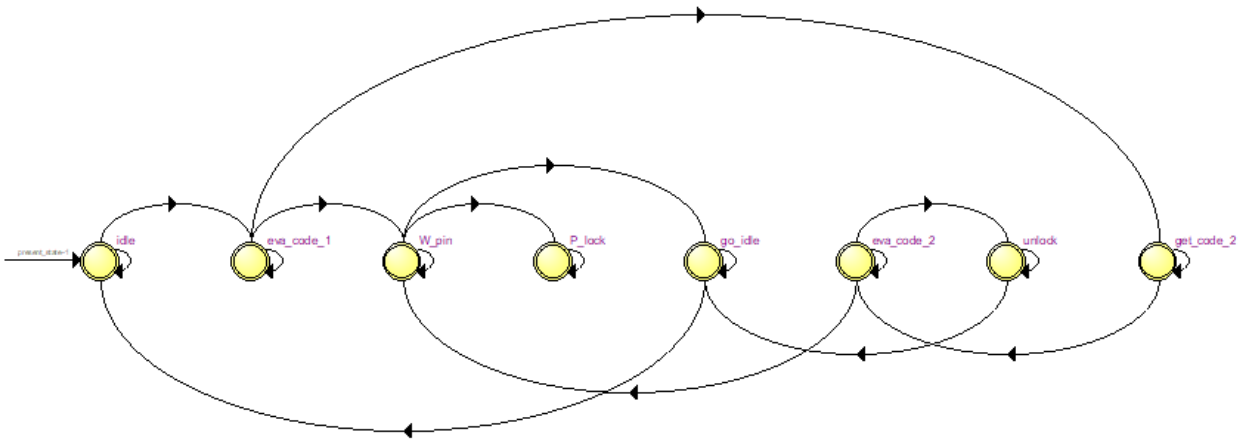
Figur 10 - Entity for code_lock_err

Entiteten for code_lock_err ses på figur 12, her instantieres in-/output-portene som der bruges i code_lock_uart og i arkitekturen nedenfor.

```
architecture code_lock_err_arch of code_lock_err is
type state is (idle, eva_code_1, eva_code_2, get_code_2, go_idle, unlock, W_pin, P_lock);
signal present_state, next_state : state;

type wc_state is (Err_0, Err_1, Err_2, Err_3 );
signal wc_present, wc_next : wc_state;

signal locked, err_event, failed : std_logic;

constant code1 : std_logic_vector(3 downto 0) := "1111";
constant code2 : std_logic_vector(3 downto 0) := "0000";
```

Figur 11 - Architecture for code_lock_err (del 1)

På figur 11 ses første afsnit af arkitekturen. Her implementeres der endnu en state machine. Vises på figur 12. Derudover kan der også ses, hvordan de to koder er indstillet, den første kode er indstillet til ’1111’ og den anden kode ’0000’.



Figur 12 - State Machine View af code_lock

Code_lock_err virker på den måde at den efter korrekt udførsel af kodesekvens fra superbrugeren, vil deaktivere kodelåsen. Derudover er der blevet gjort krav på, at kodelåsen kan blive sat i permanent låst tilstand, hvis en uvedkommende person prøvede på at indtaste koden, eller hvis superbrugeren selv indtaster koden forkert 3 gange. Det kan ses på figur 12. Hvor der efter begge evaluate_code(eva_code_1 og eva_code_2) er en pil hen til W_pin, som betyder, at det er en forkert kode, der er indtastet. Hvis koden indtastes forkert, vil den gå i P_lock hvorved kodelåsen vil låses indtil der trykkes på reset-knappen, hvor processen startes forfra. Evalueres første kode til at være korrekt, afventes den anden kode og hvis begge er indtastet rigtig, vil den gå over i ”unlock”-state, hvorefter kodelåsen deaktiveres.

```
begin
   code_lock_state: process(clk, reset)
   begin
      if reset = '0' then
         present_state <= idle;

      elsif rising_edge(clk) then
         present_state <= next_state;
      end if;
   end process;
```

Figur 13 - Architecture for code_lock_err (del 2)

På figur 13 vises første process, der tjekker hvornår der opstår rising edge for clk, ved rising edge sættes present_state til next_state. Derudover tjekker code_lock_state også om der trykkes på reset hvor processen sættes idle/ starter forfra.

```
code_lock_next_state: process (pin)
variable tries: integer := 0;
begin
next_state <= present_state;
   case present_state is
      when idle =>
         err_event <= '0';
         next_state <= idle;
         locked <= '1';
         if enter = '0' then
            next_state <= eva_code_1;
         end if;

      when eva_code_1 =>
         locked <= '1';
         err_event <= '0';
         if enter = '1' and pin = code1 then
            next_state <= get_code_2;
         elsif enter = '1' and pin /= code1 then
            err_event <= '1';
            next_state <= W_pin;
         end if;

      when get_code_2 =>
         locked <= '1';
         err_event <= '0';
         if enter = '0' then
            next_state <= eva_code_2;
         end if;

      when eva_code_2 =>
         locked <= '1';
         err_event <= '0';
         if enter = '1' and pin = code2 then
            next_state <= unlock;
         elsif enter = '1' and pin /= code2 then
            err_event <= '1';
            next_state <= W_pin;
         end if;

      when go_idle =>
         locked <= '1';
         err_event <= '0';
         if enter = '1' then
            next_state <= idle;
         end if;

      when W_pin =>
         locked <= '1';
         err_event <= '0';
         if (failed = '1' and enter = '0') then
            next_state <= P_lock;
         elsif (enter = '0') then
            next_state <= go_idle;
         end if;

      when P_lock =>
         err_event <= '0';
         locked <= '1';

      when unlock =>
         err_event <= '0';
         locked <= '0';
         if enter = '0' then
            next_state <= go_idle;
         end if;
   end case;
end process;
```

Figur 14 - Architecture for code_lock_err (del 3)

I arkitekturafsnittet på figur 14, ses hvordan present_state er stillet op med case-sætninger. Disse case-sætninger omhandler blot, hvad der er skrevet tidligere om code_lock_err's virkemåde. En tilføjelse er, at der ses hvilke signaler der bliver indstillet, for hver state i state machine view, figur 12. F.eks. sættes locked <= '0' når programmet går i "unlock"-state hvor det tidligere er nævnt, at kodelåsen bliver deaktiveret.

```
   code_lock_out : process(present_state)
   begin
      lock <= locked;
   end process;

   wc_sr: process (reset, clk)

      begin
      if reset = '0' then
         wc_present <= Err_0;

      elsif rising_edge(clk) then
         wc_present <= wc_next;
      end if;
   end process;

   wc_ns: process (wc_present, err_event)
   begin
      wc_next <= wc_present;
      case wc_present is
         when Err_0 =>
            if err_event = '1' then
               wc_next <= Err_1;
            end if;

         when Err_1 =>
            if err_event = '1' then
               wc_next <= Err_2;
            end if;

         when Err_2 =>
            if err_event = '1' then
               wc_next <= Err_3;
            end if;

         when Err_3 =>
      end case;
   end process;

   wc_os: process (wc_present)
   begin
      case wc_present is
         when Err_3 =>
            failed <= '1';
         when others =>
            failed <= '0';
      end case;
   end process;

end code_lock_err_arch;
```

Figur 15 - Architecture for code_lock_err (del 4)

Den 4. del af arkitekturen indeholder code_lock_err's output som sættes lig værdien af locked. Desuden er der også implementeret err_states heri. Disse 2 processer der omhandler wrongcode, (wc_ns og wc_os)

skal blot forstås som at koden indtastes forkert 3 gange, og hvis dette sker sættes failed <= '1' hvorefter programmet går i staten "P_lock", som låser kodelåsen permanent indtil der trykkes på reset knappen.

```
if (failed = '1' and enter = '0') then
   next_state <= P_lock;
```

Code_lock_uart:

Code_lock_uart.vhd

- Code_lock_uart.vhd består af de 3 ovenstående .vhd filer, her er de 3 filers funktionalitet blot samlet i én fil, for at vi kan lave en samlet indstilling af in-/output-portene for at de kan blive tildelt en pin på DE2-Boardet, som kan benyttes i praksis.

```
entity code_lock_uart is
port(
   --input
   KEY : in std_logic_vector(3 downto 2);
   CLOCK_50 : in std_logic;
   SW : in std_logic_vector(3 downto 0);
   --output
   GPIO_0    : out std_logic_vector(0 downto 0));
end;
```

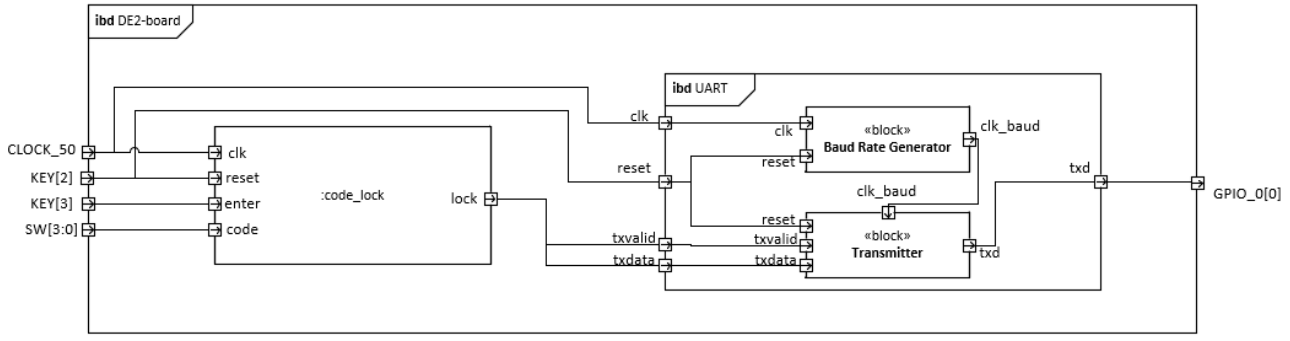
Figur 16 - Entity for code_lock_uart

Entiteten for code_lock_uart vises på figur 14. Her bemærker vi at KEY[3] og KEY[2] samt SW[0 til og med 3] kan benyttes af superbrugeren som inputs. Derudover sættes output til GPIO_0 så det er denne pin der skal forbindes med en ledning til Arduino for at etablere UART-kommunikationen.

```
architecture arch of code_lock_uart is
signal lock_signal: std_logic_vector(7 downto 0);
signal clk_baud_signal: std_logic;

begin
   u1: Baud_Rate_Generator(arch) port map(
   clk_baud => clk_baud_signal,
   reset    => KEY(2),
   clk      => CLOCK_50
   );

   u2: code_lock_err(code_lock_err_arch) port map(
   code => SW,
   enter => KEY(3),
   clk => CLOCK_50,
   reset => KEY(2),
   lock => lock_signal(0)
   );

   lock_signal(7 downto 1) <= "0000000";

   u3: UART_transmitter(arch) port map(
   reset    => KEY(2),
   txvalid  => lock_signal(0),
   txdata   => lock_signal,
   txd      => GPIO_0(0),
   clk_baud => clk_baud_signal
   );

   lock_signal(7 downto 1) <= "0000000";

end;
```

Figur 17 - Architecture for code_lock_uart

I ovenstående kodeafsnit er arkitekturen for den samlede code_lock_uart. Her er den samlede arkitektur for Baud_Rate_Generator, code_lock_err og UART_transmitteren implementeret. Her vises hvordan de forskellige in-/output-porte er port mappet, altså hvilke pins på DE2-Boardet, som de er blevet tildelt. Derudover tilføjes der to signaler, clk_baud_signal og lock_signal. Clk_baud_signalet er oprettet for at få baud_rate over i transmitterdelen, så der er bestemt en datasignaleringshastighed. Lock_signalet oprettes for at lave en kobling mellem code_lock_err's output "lock" og UART_transmitterens "txdata og txvalid" inputs, se figur 18.



Figur 18 - IBD for DE2-Board

Desuden skal der gøres opmærksom på at txdata er et 8bit datasignal, men at der samtidig kun skal benyttes 1 bit til at determinere om kodelåsen er aktiveret eller deaktiveret. Dermed er lock_signalets 7 første bit sat lig '0'. Så der kun kan aflæses fra den sidste bit, (0).

```
lock_signal(7 downto 1) <= "0000000";
```

Dermed kan man indstille det sidste bit, til at være lig med outputtet fra code_lock_err, nemlig om DE2-Boardet er låst '1' eller låst op '0'

```
lock => lock_signal(0)
```

SWA7 - Test af DE2-Board

Test for code_lock

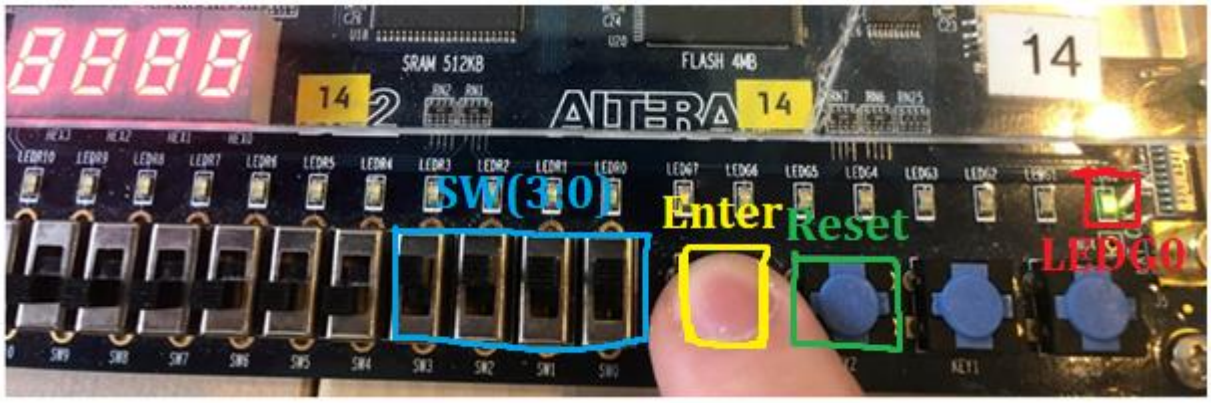
Herunder ses testbenchen af code_lock_err.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.all;

entity code_lock_test is
port (CLOCK_50 : in std_logic;
      KEY : in std_logic_vector (3 downto 2);
      SW : in std_logic_vector(3 downto 0);
      LEDG : out std_logic_vector (0 downto 0));
end code_lock_test;

architecture code_lock_test_arch of code_lock_test is

begin

   il: entity code_lock_err port map
   (
   clk => CLOCK_50,
   reset => KEY(2),
   enter => KEY(3),
   pin => SW(3 downto 0),
   lock => LEDG(0)
   );

end code_lock_test_arch;
```

Figur 19 - Kode for tester code_lock_test

I testbenchen implementers portmapping af entiteten og arkitekturen for code_lock_err.

På følgende figurer vises der test for code_lockens funktionalitet på DE2-boardet:

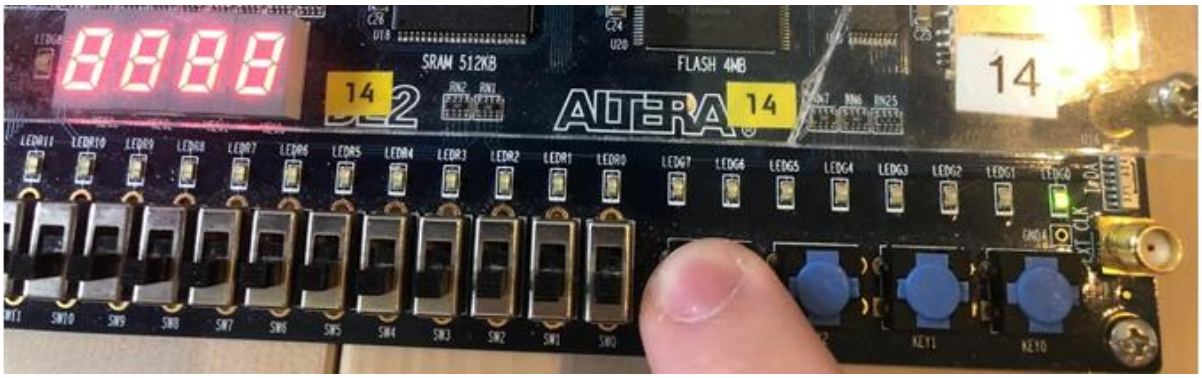


Figur 20 Test på De2-Board her indtastes code1 ”1111”, det ses at LEDG[0] lyser så lock = ’1’

På figur 22 bliver den første korrekte kode (1111) indtastet, efterfulgt af enter-knappen. LED´et lyser, da programmet er i en locked tilstand.

Figur 21 Test på DE2-Board her indtastes code2 ”0000”, hvor LEDG[0] stopper med at lyse så unlock er initieret

Den anden korrekte kode (0000) bliver nu indtastet efterfulgt af enter-knappen. LED´et stopper med at lyse, hvilket vil sige at programmet ændrer tilstand fra locked til unlocked.



Figur 22 Test på DE2-Board her tester vi om permanently locked virker ved at indtaste rigtig kodesekvens 1&2 hvor inden der er trykket forkert kode 3 gange.

På figur 24 er der blevet testet for om permenantly locked virker. Hvis koden bliver indtastet forkert 3 gange, bliver programmet låst, og reset-knappen skal anvendes før man kan prøve igen. Hvis der ikke

trykkes på reset-knappen, vil man ikke kunne låse op. I ovenstående tilfælde er der blevet indtastet forkert kode 3 gange, og programmer er hermed blevet låst. Den rigtige kode er herefter blevet indtastet, men dog lyser LED´et stadig. Dette er et resultat af, at reset-knappen ikke er blevet brugt. Reset-knappen skal derfor benyttes, hvis programmet bliver låst, før den korrekte kode kan indtastes.

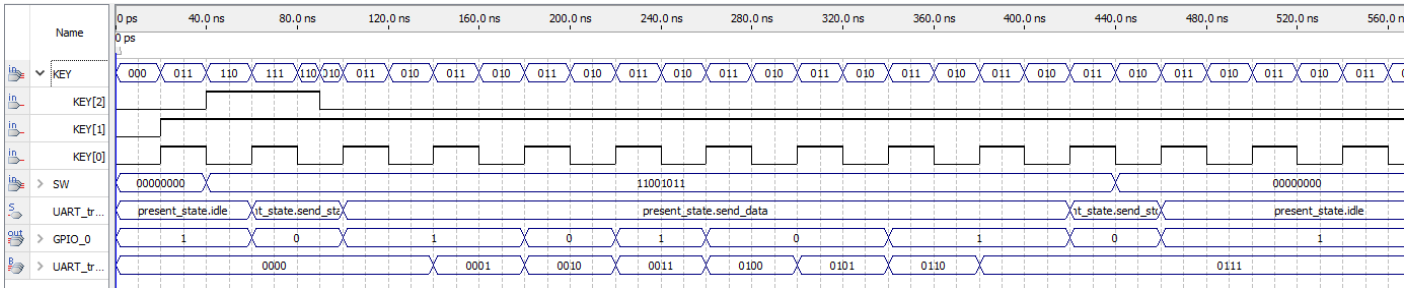
Test for UART_transmitter

Testbenchen for UART_transmitteren ses herunder, her vises hvordan entiteten for UART_transmitteren portmappes på DE2-Boardet

```
library ieee;
use ieee.std_logic_1164.all;
use work.all;

entity UART_transmitter_tester is
port(
   --input
   KEY   : in std_logic_vector(2 downto 0);
   SW    : in std_logic_vector(7 downto 0);
   --output
   LEDR  : out std_logic_vector(0 downto 0)
);
end;

architecture arch of UART_transmitter_tester is
begin
   ul:entity UART_transmitter(arch) port map(
   clk_baud => KEY(0),
   reset => KEY(1),
   txvalid => KEY(2),
   txdata => SW,
   txd => LEDR(0)
   );
end;
```

Figur 23 - Kode for tester UART_transmitter_tester

For at teste UART'en er der benyttet Quartus' simuleringsværktøj til at oprette en funktionel simulering, hvorpå man kan se Transmitterens virkemåde.

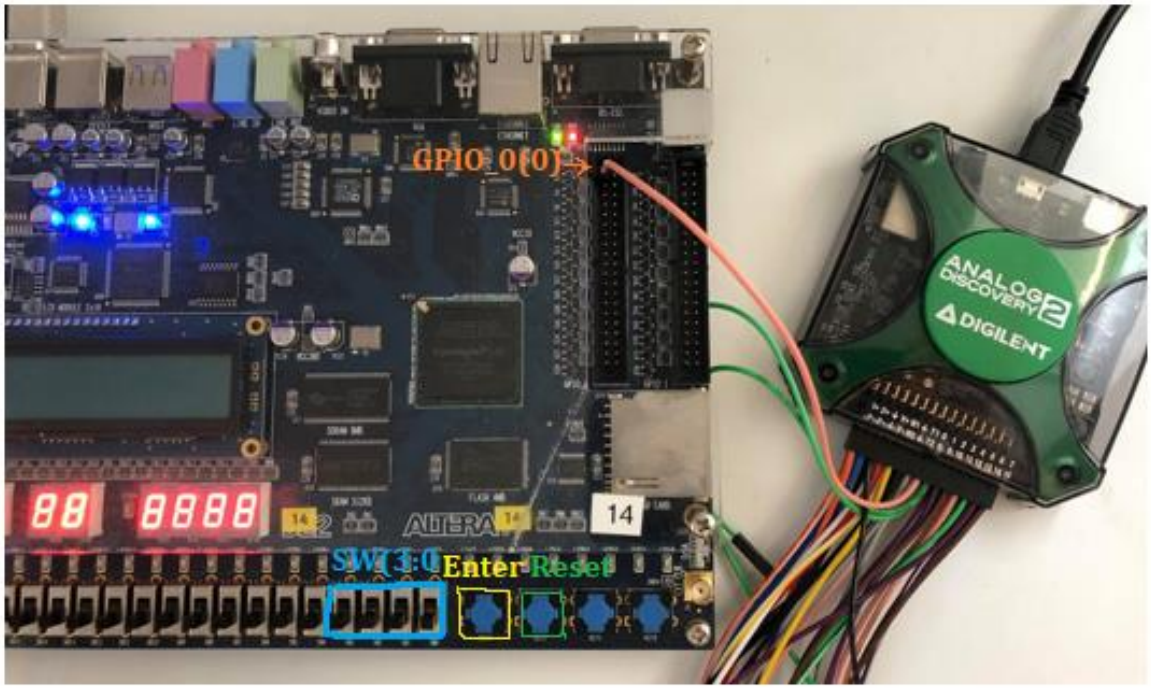


Figur 24 - Funktionel simulering for UART_transmitter_tester

Efter tilfredsstillende simulering af transmitteren er der blevet foretaget sidste test for at sikre os at DE2-Boardet virker som det skal, denne test er for code_lock_uart.

Test for code_lock_uart

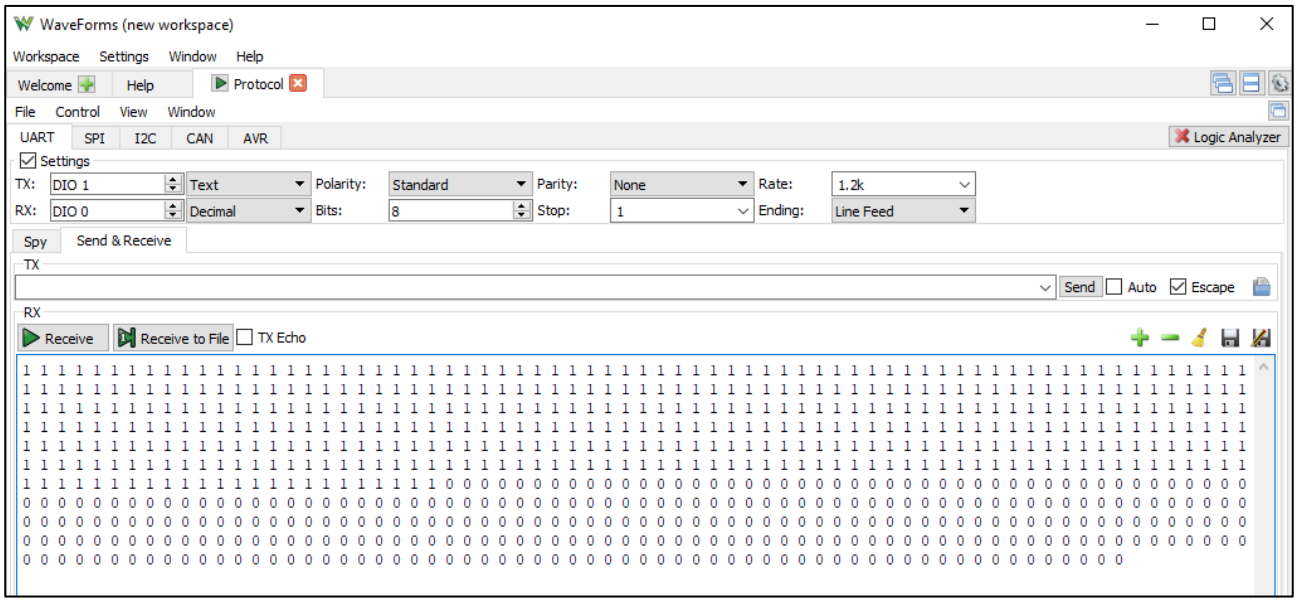
Nedenstående vises et billede af opstillingen af test for DE2-board.



Figur 25 - Opstilling af DE2-board og Analog Discovery

Opstillingen for test af kodelåsen på figur 25 består af DE2-boardet og Analog Discovery 2. Via Analog Discovery kan man benytte protocol-funktionen i Waveforms, her kan man under UART, se hvilke signaler der bliver transmitteret. På opstillingen er de forskellige keys og switches markeret med farver. Koden er skrevet således, at programmet er i en locked-state, og bliver derfor ved med at sende et højt signal ud. Under testen indtastes kodesekvensen på de fire tildelte switches, efterfulgt af at man trykker på enter-knappen. Hvis koden er korrekt, ændres tilstanden fra locked-state til en unlocked-state og sender hermed et lavt signal ud. Hvis koden til gengæld er forkert, vil programmet blive ved med at sende et højt signal ud.

Figur 26 er et screendump af WaveForms, der viser hvordan outputtet ser ud, når den korrekte kodesekvens indtastes. Som man kan se, stater den med at sende et højt signal ud, dvs. at programmet er i locked-state. Der vil altså fortsat blive sendt ’1’-taller ud så længe den rigtige kodesekvens ikke er blevet tastet på DE2-Boardet. Så snart den korrekte kode bliver indtastet på DE2-boardet, vil programmet ændres til unlocked state og dermed sende et lavt signal (0´ere) ud gennem GPIO_0 pinen. Waveforms viser det ønskede output og der kan derfor konkluderes, at kodelåsen virker.



Figur 26 - Resultater i Waveforms af signal

Ideelt set skal DE2-boardet ikke forbindes til Analog Discovery. Outputtet der kommer ud af GPIO_0 skal ud til arduinoen, så arduinoen kan give brugeren adgang til at ændre indstillingerne omtalt i Use Case 3. Men da vi grundet omstændighederne, ikke har haft mulighed for at benytte skolens laboratorie, eller kunne samle os som gruppe og opstille kredsløbet. Har vi forsøgt os at teste funktionaliteten vha. Analog Discovery.