Frederiksen
Passion for science

Skriv egne objekter

Arduino-eksperiment	130320	*Stikord*	Tilstandsmaskine, klasser og objekter	
Version	2018-07-10 / HS	*Niveau*	Videregående	*p.* 1/6

Det lærer du:

- Opbyg kontakt-debouncing ved hjælp af en *tilstandsmaskine*
- Lav dine egne *objekter* i programmet som en model af fysiske objekter

Denne vejledning fokuserer på ny software-detaljer. Selve opstillingen er rimeligt "kedelig": Et par kontakter skal styre et par lysdioder – samtidigt med, at programmet løbende udskriver tiden, f.eks. i PC'ens monitorvindue.

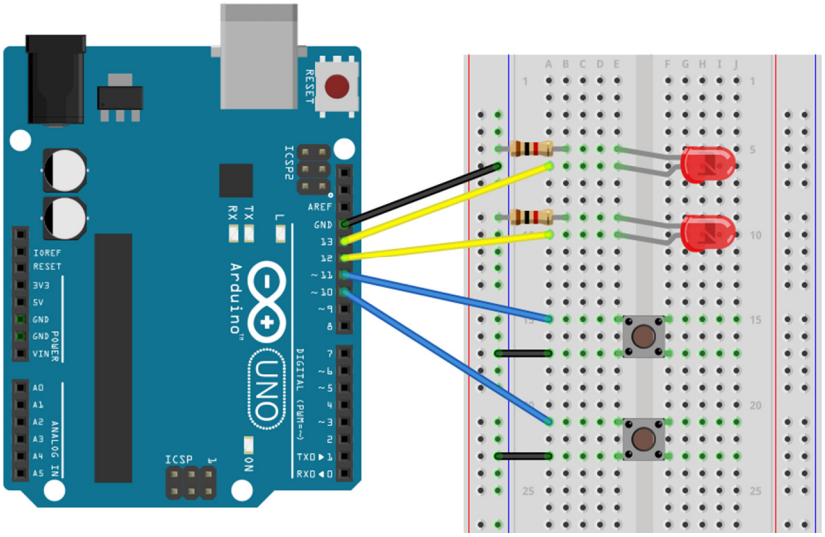
Pointen er, hvordan trykafbryderens opførsel indkapsles i en såkaldt *klasse*, som stiller nogle veldefinerede metoder til rådighed. Når klassen først er defineret, kan man oprette *objekter*, som har klassen som "type". Sammenlign med, hvordan *LiquidCrystal*-biblioteket håndterer LCD-displayets opførsel.

Når vi er færdige, har du en klasse ved navn `Switch`, som kan genbruges overalt, hvor en trykafbryder skal anvendes – komplet med debouncing.

(Der bliver en del læse- og tastearbejde i dette modul – hold ud, det er nyttigt! ☺)

1 – Lav en opstilling med afbrydere og lysdioder

Vi skal arbejde med en opstilling med to afbrydere og to lysdioder. Se opstillingen herunder. (Vi er flyttet op i den høje ende af bennumrene, så du direkte kan kombinere med vejledningen 130315 LCD Display, hvis du ønsker det.)



2 – Lav en indledende test

Brug følgende konstanter til at definere de anvendte ben på Arduinoen:

```
const byte pinLed1 = 12;
const byte pinLed2 = 13;
const byte pinSwitch1 = 10;
const byte pinSwitch2 = 11;
```

I `setup()`-funktionen skal LED-benene sættes til *udgange*, og switch-benene skal være *indgange*, som bruger *pull-up*. (Hvis i tvivl, se vejledning 130115 Tænd med en kontakt.)

Følgende er en quick-and-dirty test på, at ledningerne er forbundet korrekt og lysdioderne vender rigtigt:

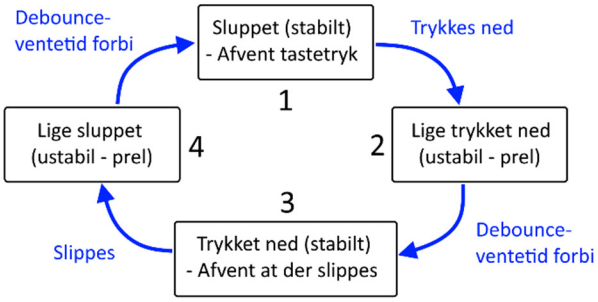
```
void loop() {
  digitalWrite(pinLed1, digitalRead(pinSwitch1)==LOW );
  digitalWrite(pinLed2, digitalRead(pinSwitch2)==LOW );
}
```

Når programmet kører, skal hver afbryder tænde for lysdioden med samme nummer – så længe knappen er nedtrykket. (Hvis ikke det virker, må du løse problemet, inden du går videre.)

3 – Tilstandsmaskine

Man kan opstille et såkaldt *tilstandsdiagram* for en afbryder med debouncing – se nedenfor. Når knappen er sluppet og har været det et stykke tid, er knappen i en stabil, afbrudt tilstand. Den har fået nummeret 1 nedenfor.

I samme øjeblik, afbryderen sluttes, ryger systemet i tilstand 2: Her kan der komme gentagne skift af spændingsniveau på Arduino-benet, men så længe *debounce*-tiden ikke er gået, er vi ligeglade. Først når der har været ro i passende lang tid, betragtes spændingen som stabil, og systemet hopper til tilstand 3.



Så længe kontakten forbliver trykket ned, holder systemet sig i tilstand 3.

Når knappen slippes, hopper systemet til tilstand 4, som ligner tilstand 2, idet vi bare afventer at tiden går, hvorefter der hoppes helt tilbage til tilstand 1 igen.

Denne opførsel vil vi lave en model for i software. En sådan programkonstruktion vil vi kalde en *tilstandsmaskine*.

4 - `switch`

Det er lidt et sprogligt sammentræf, at den programstruktur, vi vil bruge i simuleringen af afbryderen, anvender nøgleordet `switch` (se nedenfor) – i denne sammenhæng skal du dog tænke på det som en flervejs *omskifter* i software og ikke som en afbryder. Princippet er, at vi bruger en heltalsvariabel `switchState` til at holde styr på, hvilken tilstand vi er i. Variablens værdi bestemmer, hvilke dele af programkoden, der skal udføres. Det ser sådan ud:

```
switch (switchState) {
  case 1:
    if (digitalRead(pinSwitch1)==LOW) {
      lastSwitch=millis();
      switchState=2;
    }
    break;
  case 2:
    if (millis()-lastSwitch>debounceTime) switchState=3;
    break;
  case 3:
    if (digitalRead(pinSwitch1)==HIGH) {
      lastSwitch=millis();
      switchState=4;
    }
    break;
  case 4:
    if (millis()-lastSwitch>debounceTime) switchState=1;
    break;
  default: // Hvis noget HELT sært skulle ske: Start forfra
    switchState=1;
}
```

(Se værktøjsboks på næste side for en mere skematisk gengivelse af strukturen.)
I parentesen efter `switch` står variablen `switchState` , som er tilstandens nummer. Programmet hopper derefter ned i det tilfælde – den `case` – som svarer til tilstanden: Hvis `switchState` er 3, hoppes ned til `case 3:` , og programkoden herfra udføres frem til instruktionen `break` – den lysegrønne blok i listningen.

I tilstand 1 og 3 checkes kontakt-indgangens niveau. Når niveauet er det rette til at springe videre til næste tilstand, noteres tiden i variablen `lastSwitch`, som bruges i tilstand 2 og 4 til at afgøre, om vi kan regne med, at kontakt-prel er ophørt.

Hvis denne programstump placeres i `loop()`, vil den blive udført igen og igen uden pause – så tilstanden af tilstandsmaskinen (dvs. variablen `switchState`) hele tiden er et korrekt udtryk for afbryderens rent fysiske tilstand.

Der mangler dog lidt finesser, før vi er helt færdige med *Den Ultimative Kontakt-Debouncing*™.

F.eks. udfører programstumpen endnu ikke andet end at holde styr på sig selv. Men hvis `switchState` er en global variabel, kan dens værdi testes i det øvrige program. (*Udfordring 1* herunder.)

Det ville også være rart at kunne udvide funktionaliteten lidt, så vi kan teste, om kontakten er blevet trykket ned *siden sidst vi spurgte* – eller om det stadigvæk er det samme lange tryk som før. Det løser du i *Udfordring 2* nedenfor

– Men bemærk, hvordan debouncing i denne version klares uden kald af `delay()`. Programmet blokeres ikke, mens vi venter på kontakten.

Værktøj: `switch`-konstruktionen

Eksempel:

```
switch (swVar) {
  case 1:
    Serial.println(”One”);
    break;
  case 2:
    Serial.println(”Two”);
    break;
  case 6:
    Serial.println(”six”);
    break;
  default:
    Serial.println(”What?”);
}
```

Afhængigt af værdien af `swVar` udskrives teksten ”*One*”, ”*Two*” eller ”*Six*”. Hvis `swVar` ikke har en af værdierne 1, 2 eller 6, udskrives teksten ”*What?*”

Forklaring:

I hver linje med `case` står en konstant værdi. Disse sammenlignes ovenfra med værdien af `swVar` og den første, som stemmer, får programafviklingen til at springe til den følgende instruktion (efter kolonet).

Når programmet når instruktionen `break`, springes helt ud af den **{** krøllede parentes **}**. Glemmer man at skrive `break`, fortsætter programmet igennem alle de resterende tilfælde.

Tilfælde, som ikke eksplicit er fanget forinden, behandles i ”joker”-tilfældet `default`.
Det er valgfrit, om man vil skrive denne del af konstruktionen.

Udfordring 1

Opret globale variabler *med passende typer* og navnene `switchState` og `lastSwitch`. Opret også en logisk variabel `ledState1`, som skal bruges til at holde rede på tilstanden af lysdiode 1.

Slet det gamle indhold i `loop()` og skriv koden til tilstandsmaskinen dér.
Nederst i `loop()` skal følgende tilføjes:

```
ledState1 = (switchState==1) || (switchState==2); //    || betyder ”eller”
digitalWrite(pinLed1,ledState1);
```

Prøv af, om kontakt 1 og lysdiode 1 virker fuldstændigt som før. (Meget besvær for så lidt, kan man sige ☺)

Udfordring 2

Tilføj en ny global logisk variabel:

```
bool justPressed;
```

Sæt denne `true` lige før `switchState` sættes til at være 2.

Nede i den afdeling, hvor du styrer lysdiodens tilstand, kan du nu spørge til `justPressed` – og hvis den er sand, skal du gøre to ting: Sætte variablen falsk igen og derefter skifte tilstand på lysdioden.

Dvs.: I stedet for at lade lysdioden følge knappen, skal den skifte mellem tændt og slukket for hvert tastetryk.
(Og først *nu* høster vi fordelene af, at tilstandsmaskinen klarer debouncing af kontakten på fornem vis.)

5 – Klasser og objekter

Der er en anden type problem, som har luret, siden vi satte kontakt nr. to i breadboardet:

Vores tilstandsmaskine er designet til at simulere én kontakt. Den næste kontakt kan måske tilføjes ved at dublere koden for hele tilstandsmaskinen – og ændre variabelnavnene, så de bliver specifikke for hver sin kontakt.
OK. Men hvad nu, hvis man får lyst at tilføje endnu én (eller fem)? Det bliver ret hurtigt uoverskueligt, det her.

Som løsning vil vi flytte hele tilstandsmaskinen og dens tilhørende variabler ind i en såkaldt *klasse*, og derefter oprette *objekter* (et for hver knap), som har klassen som type.

(Vi når kun at kradse i overfladen på det kolossale emne *objektorienteret programmering*. Derfor bliver dette afsnit også en del mere "kogebogsagtig" end resten af kurset – men nye udfordringer til dig følger til sidst.)

Oppe i starten af programmet – inden vi når til `setup()` – defineres klassen `Switch` (bemærk det store "S"!) således:

```
class Switch {
private:
  unsigned long lastSwitch;    // Tids-stempel
  word switchState;            // Tilstand iflg. tilstandsdiagram
  byte pinSwitch;              // Det anvendte Arduino-ben
  bool justPressed;            // Er kontakten trykket ned siden sidste check?
public:
  Switch(byte pin);            // Constructor (forklares i teksten)
  bool switchDown();           // Sand, hvis knappen er trykket ned i øjeblikket
  bool switchPressed();        // Sand, hvis trykket ned siden sidste kald
  void switchLoop();           // Kaldes ved hvert gennemløb af loop()
};
```

Denne definition fortæller, hvilke variabler, de tilhørende objekter råder over, samt hvilke funktioner, der anvendes. De variabler og funktioner, som skal stilles til rådighed for det øvrige program, skrives i afsnittet `public`, mens resten erklæres som `private` – de kan ikke røres udenfor objektet. Læg mærke til, at det stort set er de globale variabler, vi har brugt indtil nu i projektet, som flyttes ind i objektets private område.

Det er normalt en særdeles god hovedregel at holde variabler `private`. Funktioner kan skrives begge steder. De offentlige funktioner kaldes også for klassens (eller objektets) *metoder*. Vi har i dette tilfælde ingen private funktioner.

En af metoderne – `Switch()` – skiller sig ud ved at have samme navn som klassen, samt ikke at have specificeret nogen type. Den kaldes klassens *constructor*, og bruges når objektet oprettes. Der kommer eksempler nedenfor. Fælles for metoderne er, at ovenstående kun er en erklæring omkring deres navne, typer og parametre. Vi er også nødt til at skrive, hvordan de skal virke i praksis. For at angive procedurens tilhørsforhold til klassen, sætter man klassenavnet foran med et dobbelt-kolon. Lad os se dette i praksis med to af metoderne:

```
bool Switch::switchDown() {
  return ((switchState==2)||(switchState==3));
}

bool Switch::switchPressed() {
  bool p=justPressed;
  justPressed=false;
  return p;
}
```

Den første returnerer en logisk værdi, som afspejler, om kontakten er i tilstand 2 eller 3 (dvs. trykket ned).

Den anden funktion undersøger værdien af den private variabel `justPressed` og sætter den falsk, inden den oprindelige værdi returneres. Hvis tilstandsmaskinen sørger for at sætte `justPressed=true` ved overgangen til tilstand 2, opnår vi den ovenfor beskrevne debouncing.

Herunder følger constructoren:

```
Switch::Switch(byte pin) {
  pinSwitch=pin;
  pinMode(pinSwitch,INPUT_PULLUP);
  lastSwitch=0;
  switchState=1;
  justPressed=false;
}
```

Constructorens kode ligner noget, man normalt ville placere i `setup()` . Der sættes nogle startværdier af variablerne. Men her er det vigtigt, at det anvendte ben-nummer kommer som en parameter! Det betyder nemlig, at vi kan oprette flere objekter af typen `Switch` , som anvender *hvert sit ben* på Arduinoen.

Tilbage ligger den opgave, som tilstandsmaskinen tidligere havde placeret i `loop()` . Det skal flyttes til metoden `switchLoop()` , som så skal kaldes i `loop()`-funktionen. Metoden kodes således:

```
void Switch::switchLoop() {
  switch (switchState) {
    case 1:
      if (digitalRead(pinSwitch)==LOW) {
        justPressed=true;
        lastSwitch=millis();
        switchState=2;
      }
      break;
    case 2:
      if (millis()-lastSwitch>debounceTime) switchState=3;
      break;
    case 3:
      if (digitalRead(pinSwitch)==HIGH) {
        lastSwitch=millis();
        switchState=4;
      }
      break;
    case 4:
      if (millis()-lastSwitch>debounceTime) switchState=1;
      break;
    default:
      switchState=1;
  }
}
```

Sammenlign med koden i afsnit 4!

Alt dette betyder, at vi nu kan oprette et objekt for hver afbryder. Et objekt, som stiller de krævede metoder til rådighed for hovedprogrammet, men som ”skjuler de beskidte detaljer” i private variabler. Pointen er, at de private variabler hører til *hvert sit* objekt, selv om objekterne har samme klasse. Vi skal ikke til at holde styr på en masse globale variabler hørende til den ene, den anden eller den tredje tilsluttede afbryder.

Her følger det meste af det resterende program – forklaring nedenfor:

```
Switch sw1(pinSwitch1);
Switch sw2(pinSwitch2);

void setup() {
  pinMode(pinLed1,OUTPUT);
  pinMode(pinLed2,OUTPUT);
  Serial.begin(500000);
}

void loop() {
  sw1.switchLoop();
  sw2.switchLoop();
  checkTime();
  if (sw1.switchjustPressed()) {
    ledState1=!ledState1;
    digitalWrite(pinLed1,ledState1);
  }
  if (sw2.switchjustPressed()) {
    ledState2=!ledState2;
    digitalWrite(pinLed2,ledState2);
  }
}
```

Lad os lige kommentere den første linje: `Switch sw1(pinSwitch1);` .
Her oprettes et objekt ved navn `sw1` , af typen `Switch` , og som skal håndtere signaler på benet `pinSwitch1` .

Sammenlign evt. med, hvordan et LCD-display blev håndteret i modul 130315 LCD Display!

`setup()` håndterer klargøring af benene til lysdioderne samt den serielle forbindelse til PC'en.

Nede i `loop()` ser vi, hvordan først `switchLoop()` kaldes for hvert af de to objekter.

Derefter følger et kald af en procedure `checkTime()` , som vi endnu ikke har set noget til – den skal du skrive i *Udfordring 4*.

Og nederst holdes der styr på, og der skal tændes eller slukkes for lysdioderne.

Udfordring 3

Gem programmet under et nyt navn.

Indtast / rediger de forskellige dele af programmet, som gennemgået ovenfor. Udelad i første omgang linjen, som kalder `checkTime()` . Programmets struktur skal være:

1. Globale konstanter
2. Klassen `Switch` – erklæring af strukturen, efterfulgt af implementering af metoderne
3. Globale variabler: `ledState1, ledState2, sw1` og `sw2`
4. `setup()`
5. `loop()`

Kontrollér, at programmet opfører sig, som det skal.

Som kort nævnt tidligere, foregår debouncing i denne version uden at processoren lægges død med et kald af funktionen `delay()` . Det muliggør f.eks. løsningen af den overordnede opgave, som nævnes helt oppe i indledningen:

"Et par kontakter skal styre et par lysdioder – ***samtidigt med***, at programmet løbende udskriver tiden, f.eks. i PC'ens monitorvindue."

Udfordring 4

Gem programmet under et nyt navn.

Skriv en funktion `void checkTime()` som skal kaldes for hvert gennemløb af `loop()` , og som har til opgave at udskrive tiden siden Arduino startede.

Tiden skal udskrives i IDE'ets monitorvindue på PC'en (eller alternativt på LCD-displayet) mindst hvert 0,1 sekunder. Tiden skal angives i sekunder med to decimaler.

Der er stadigvæk et par globale variabler tilbage af dem, der blev oprettet i udfordring 1, hørende til lysdioderne. Det er en udmærket øvelse at rydde helt op, så kun objekter oprettes globalt.

Udfordring 5

Gem programmet under et nyt navn.

Skriv definition på en ny klasse `LED` , som skal indkapsle lysdiodens opførsel. Den globale variabel, som gemte lysdiodens tilstand, skal nu være `private` . Du skal også have en variabel til ben-nummeret.

Skriv en constructor `LED::LED(byte pin)` som gemmer ben-nummeret i den private variabel. Flyt klargøringen af benet som udgang væk fra `setup()` og hertil.

Indfør tre metoder `turnOn()` , `turnOff()` og `toggle()` , som hhv. tænder, slukker og skifter nuværende tilstand til det modsatte. (Metoderne skal både sætte benets tilstand og opdatere variablen, som husker tilstanden.)

Opret objekter `led1` og `led2` af typen `LED` , og test de tre metoder af. Når du er tilfreds med opførslen af din `LED`-klasse, vender du tilbage til den oprindelige opgave.
(Du får her ret beset kun brug for `toggle()`-metoden.)