Frederiksen®
Passion for science

Schrijf je eigen objecten

Arduino-experiment 130320-EN	*Trefwoorden* State machine, klassen en objecten	
Versie 2018-07-09 / HS	*Niveau* Gevorderd	*p.* 1/8

Wat je leert:

- Maak een switch debouncing met behulp van een *state machine*
- Maak je eigen *objecten* in het programma als een model van fysieke objecten

Deze handleiding richt zich op nieuwe software details. De opstelling als zodanig is nogal “saai”: een paar schakelaars voor het aansturen van een paar LED's – terwijl het programma tegelijkertijd de looptijd uitvoert, bijvoorbeeld in het monitorvenster van de IDE.

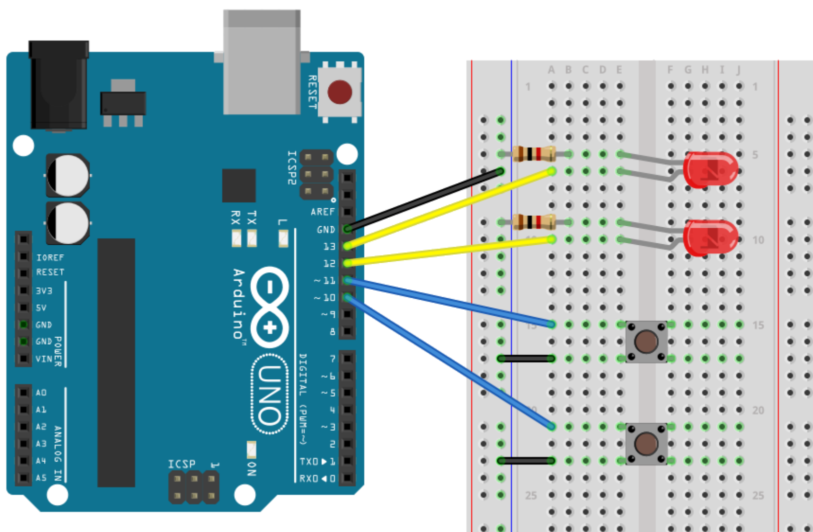
Het punt is hoe het gedrag van de schakelaar is ingekapseld in een zogenaamde *klasse* die een aantal goed gedefinieerde methoden biedt. Wanneer de klasse is gedefinieerd, kun je *objecten* aanmaken die deze klasse als “type” hebben. Vergelijkbaar met de manier waarop de *LiquidCrystal library* omging met het gedrag van het LCD-display.

Eenmaal klaar heb je een klasse genaamd `Switch`, die je overal kunt hergebruiken waar een drukknopschakelaar wordt gebruikt – inclusief de nodige debouncing.

(Deze module bevat een hoop lezen en typen – even doorzetten, want het is nuttig! ☺)

1 – Opstelling met schakelaars en LED's

We werken met een opstelling met twee schakelaars en twee LED's. Zie de opstelling hieronder. (We zijn naar de hogere pinnummers gegaan, zodat je dit desgewenst direct kunt combineren met handleiding 130315 LCD-Display.)



2 – De eerste test

Gebruik de volgende constanten om de pins te definiëren die op de Arduino worden gebruikt:

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

In de `setup()`-functie moeten de LED-pins *outputs* worden gemaakt, en de schakelpins moeten *inputs* worden gemaakt, *met behulp van pullup*. (Raadpleeg in geval van twijfel de handleiding van module 130115 Een schakelaar gebruiken.)

Hieronder volgt een sneltest om de aansluitingen en de positionering van de LED's te controleren:

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

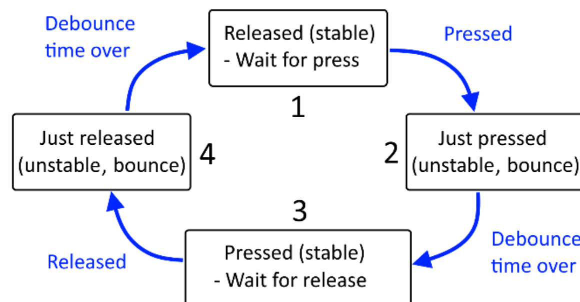
```
    digitalWrite(pinLed2, digitalRead(pinSwitch2)==LOW );
  }
```

Bij het draaien moet elke schakelaar de LED met hetzelfde nummer aanzetten – zolang de knop maar wordt ingedrukt. (Als dit niet werkt moet je het probleem vinden en oplossen voordat je verder gaat.)

3 – State Machine

Je kunt een zogenaamd *state diagram* maken voor een schakelaar met debouncing – zie hieronder. Wanneer de knop wordt losgelaten en enige tijd is gepasseerd, is de schakelaar in een stabiele "uit"-toestand. Deze toestand wordt hieronder aangegeven met het nummer 1.

Zodra de schakelaar wordt gesloten, komt het systeem in toestand 2 te staan: hier kunnen talrijke veranderingen van het spanningsniveau op de Arduino-pin gebeuren, maar zolang de *debounce time* niet voorbij is, maakt dat ons niets uit. Pas als het systeem lang genoeg in rust is, wordt de spanning als stabiel beschouwd en komt het systeem in toestand 3 terecht.



Zolang de knop wordt ingedrukt, blijft het systeem in toestand 3 staan.

Als de knop wordt losgelaten, springt het systeem naar toestand 4 – die lijkt op toestand 2 terwijl we gewoon even wachten tot we erdoor zijn – waarna we weer helemaal teruggaan naar toestand 1.

We gaan voor dit gedrag een model maken in de software. We noemen zo'n constructie voor een *state machine*.

4 – `switch`

Het is een taalkundig toeval dat de softwarestructuur die we gaan gebruiken voor het simuleren van een switch gebruik maakt van het trefwoord `switch` (zie hieronder). In dit verband lijkt het meer op een draaischakelaar in software dan op een drukknopschakelaar. Het principe is dat er een integer-variabele `switchState` wordt gebruikt om het toestandsnummer van het systeem bij te houden. De waarde van de variabele bepaalt welk deel van de programmacode wordt uitgevoerd. Het ziet er als volgt uit:

```
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: // If anything REALLY weird happens: Start over
    switchState=1;
```

```
}
```

(Zie de toolbox op de volgende pagina voor een meer schematische weergave van deze structuur)
Tussen de haakjes na `switch` staat de variabele `switchState`, dat is het nummer van de status. Het programma springt dan naar beneden naar de `case` die overeenkomt met het nummer: Als `switchState` 3 is, springt de uitvoering naar `case3:`, en wordt de programmacode vanaf hier tot aan de `break` uitgevoerd – het lichtgroene blokje in de lijst.

In toestanden 1 en 3 wordt het niveau van de switch input afgelezen. Wanneer het niveau goed is om door te gaan, wordt de tijd genoteerd in de `lastSwitch`-variabele die wordt gebruikt in toestand 2 en 4 om te bepalen of we de *switch bouncing* als voorbij kunnen beschouwen.

Als dit stukje code in `loop()` wordt geplaatst, zal het steeds opnieuw worden herhaald zonder enige onderbreking – waardoor de waarde van de toestandsvariabele (`switchState`) altijd een geldige afspiegeling is van de fysieke toestand van de schakelaar.

Er ontbreken nog enkele functies voordat we de voltooiing van *The Ultimate Switch Debouncer™* kunnen vieren.

De programmacode *doet* bijvoorbeeld niets anders dan zichzelf op de rails houden. Maar als `switchState` een globale variabele is, kan de waarde ervan elders in het programma worden getest *(Uitdaging 1* hieronder).

Een andere leuke eigenschap zou zijn om te kunnen testen of de schakelaar is ingedrukt *sinds de vorige keer dat we dezelfde vraag stelden* – of dat het nog steeds dezelfde verlengde druk is als voorheen. Dit voeg je hieronder toe in *Uitdaging 2*.

– Maar let op hoe het debouncen in deze versie gebeurt zonder het aanroepen van `delay()`. Het programma is niet geblokkeerd terwijl we op de schakelaar wachten.

Toolbox: De `switch`-constructie

Voorbeeld:

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

Afhankelijk van de waarde van `swVar` wordt de tekst ”*One*”, ”*Two*” of ”*Six*” als output geproduceerd. Als `swVar` geen van de waarden 1, 2 of 6 heeft, wordt de tekst ”*What?*” weergegeven.

Uitleg:

Elke regel met `case` bevat een constante waarde. Deze worden, van boven naar beneden, vergeleken met de waarde van `swVar` en bij de eerste overeenkomst springt de programma-uitvoering naar de volgende instructie (na de dubbele punt).

Wanneer het programma het trefwoord `break` bereikt, springt het uit de **{** accolades **}**. Als je vergeet om een `break` toe te voegen, gaat het programma verder met alle overige cases.

Elke waarde die niet eerder expliciet is gepakt, wordt behandeld door de ”joker” case `default`. Het schrijven van het `default` onderdeel van de constructie is optioneel.

Uitdaging 1

Maak globale variabelen *met geschikte types* en de namen `switchState` en `lastSwitch`. Maak ook een Booleaanse variabele `ledState1`, om de toestand van LED 1 bij te houden.

Verwijder de oude inhoud in `loop() en voer daar de code voor de state machine in.`
`Onderaan de loop()` moet het volgende worden toegevoegd:

```
ledState1 = (switchState==1) || (switchState==2); //    || means ”or”
digitalWrite(pinLed1,ledState1);
```

Test of schakelaar 1 en LED 1 zich precies zo gedragen als voorheen. (Veel gedoe voor zo weinig... ☺)

Uitdaging 2

Voeg een nieuwe globale, Booleaanse variabele toe:

```
bool justPressed;
```

Stel deze variabele `true` invlak voordat `switchState` wordt gewijzigd naar 2.

Beneden in de afdeling voor de controle van de LED-status, kun je nu de waarde van `justPressed` controleren – en als deze `true` is, moet je twee dingen doen: stel de variabele `false` opnieuw in en verander dan de LED-toestand.

D.w.z. in plaats van de LED de drukknopstand te laten volgen, moet hij voor elke druk op de knop tussen aan en uit schakelen.

5 – Klassen en objecten

Sinds we de tweede schakelaar in de breadboard hebben geplaatst ligt er nog een totaal ander probleem op de loer:

onze state machine is ontworpen om één schakelaar te simuleren. De volgende schakelaar kan wellicht worden toegevoegd door de code voor de volledige state machine te verdubbelen – en de namen van de variabelen te wijzigen om ze specifiek te binden aan elke schakelaar.
OK. Maar wat als je nog een (of vijf) schakelaars wilt toevoegen? Dat wordt al snel ingewikkeld!

We lossen dit op door de state machine en de bijbehorende variabelen te verplaatsen naar een zogenaamde *klasse*, en vervolgens *objecten* te creëren (één voor elke schakelaar) die deze klasse als type hebben.

(We blijven slechts aan de oppervlakte van het enorme onderwerp dat *object georiënteerde programmering* heet. Daarom lijkt dit gedeelte meer op een kookboek dan de meeste andere Arduino-experimenten – maar uiteindelijk volgen er nieuwe uitdagingen.)

Bovenaan het programma – vóór het bereiken van de `setup()` – definiëren we de klasse `Switch` (let op, hoofdletter ”S”!) op deze manier:

```
class Switch {
private:
  unsigned long lastSwitch;   // Timestamp
  word switchState;           // State according to state diagram
  byte pinSwitch;             // The Arduino pin to use
  bool justPressed;           // Has switch been pressed since last time?
public:
  Switch(byte pin);           // Constructor (explained in the text)
  bool switchDown();          // True if swith is pressed at the moment
  bool switchPressed();       // True if pressed since last call
  void switchLoop();          // Must be called for each execution of loop()
};
```

De definitie geeft aan tot welke variabelen de corresponderende objecten toegang hebben en welke functies worden gebruikt. Variabelen en functies die beschikbaar zijn voor de rest van het programma worden gespecificeerd in het `public` gedeelte, terwijl de rest als `private` wordt gespecificeerd – deze kunnen niet buiten het object worden aangeraakt. Merk op hoe een aantal van de globale variabelen in het project nu naar het privé-gebied van het object worden verplaatst.

Het is meestal een erg goed idee om variabelen `private` te houden. Functies kunnen in beide secties worden gespecificeerd. De publieke functies worden ook wel de *methoden* van de klasse (of het object) genoemd. In dit geval zijn er geen privé-functies.

Eén van de functies – `Switch()` – valt op: deze heeft dezelfde naam als de klasse en geen gespecificeerd type. Het wordt de *constructor* van de klasse genoemd en wordt gebruikt wanneer het object wordt aangemaakt. Voorbeelden hiervan volgen hieronder. Wat de bovenstaande methoden met elkaar gemeen hebben is dat we alleen hun namen, types en parameters hebben gespecificeerd. We zullen ook hun implementatiecode moeten schrijven. Om het lidmaatschap van de klasse te specificeren, wordt de naam van de klasse voorafgegaan door twee keer een dubbele punt. Laten we dit in de praktijk bekijken met twee voorbeelden:

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

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

```
    return p;
  }
```

De eerste geeft een logische waarde terug die aangeeft of de schakelaar in state 2 of 3 staat (d.w.z. ingedrukt).

De andere functie controleert de waarde van de privé-variabele `JustPressed`, maakt deze `false` voordat de oorspronkelijke waarde wordt teruggegeven. Als de state machine zorgt voor het instellen va n `justPressed=true` bij het bereiken van state 2, zorgt dit voor de hierboven beschreven debouncing.

Hier volgt de constructor:

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

De code van de constructoren ziet eruit als iets wat je normaal gesproken in `setup()` zou plaatsen. De variabelen zijn voorzien van beginwaarden. Maar het is belangrijk dat het gebruikte pinnummer via een parameter wordt gegeven! Dit betekent dat we meer objecten van het type `Switch` kunnen maken, elk *met een eigen individuele pin* op de Arduino.

Tot slot hebben we de taak van de state machine, die eerder in een `loop()` werd geplaatst. Deze moet worden verplaatst naar de methode `switchLoop()`, die dan wordt aangeroepen vanuit de `loop()`-functie. De methode is als volgt gecodeerd:

```
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;
  }
}
```

Vergelijk met de code in onderdeel 4!

Dit alles betekent dat we nu voor elke schakelaar een object kunnen maken. Een object dat de methoden biedt die nodig zijn voor het hoofdprogramma – maar dat “de vervelende details verbergt” in privé-variabelen. Het punt is dat de privé-variabelen tot *afzonderlijke* objecten behoren, zelfs als de objecten dezelfde klasse hebben. Het is dus niet meer nodig om veel globale variabelen bij te houden die bij de ene, de andere of een derde aangesloten schakelaar horen.

Hieronder volgt het grootste deel van het resterende programma – uitgelegd op de volgende pagina:

```
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);
  }
}
```

Eerst een opmerking over de eerste regel: `Switch sw1(pinSwitch1);`.
Hierdoor ontstaat een object met de naam `sw1`, met het type `Switch`, dat de signalen op pin `pinSwitch1` verwerkt.

Vergelijk dit met hoe een LCD-display wordt behandeld in module 130315 LCD-Display!

`setup()` voert nog steeds de initialisatie van de LED-pins en de seriële communicatie met de PC uit.

Onderaan in de `loop()`-functie zien we hoe `switchLoop()` wordt aangeroepen voor elk van de twee objecten.

Daarna volgt een aanroep van de functie `checkTime()`, die we nog niet eerder gezien hebben – deze schrijf je in *Uitdaging 4*.

Tot slot worden de mogelijke toestandsveranderingen van de LED's behandeld.

Uitdaging 3

Sla het programma op met een nieuwe naam.

Schrijf/bewerk de verschillende onderdelen van het programma zoals hierboven besproken. Laat de regel die de (nog niet gedefinieerde) `checkTime()`-functie aanroept weg. De structuur van het programma zou als volgt moeten zijn:

1. Globale constanten
2. De `Switch`-klasse – specificering van de structuur, gevolgd door de uitvoering van de methode
3. Globale variabelen: `ledState1, ledState2, sw1` en `sw2`
4. `setup()`
5. `loop()`

Controleer of het programma zich gedraagt zoals het hoort.

Zoals eerder vermeld, wordt bij deze implementatie het debouncen gedaan zonder al te veel belasting van de processor door de `delay()`-functie aan te roepen. Dit stelt ons in staat om software-oplossingen te creëren zoals bijvoorbeeld de algemene taak die we in het begin van dit handboek hebben opgezet:

“Een paar schakelaars voor het aansturen van een paar LED's – terwijl het programma ***tegelijkertijd*** de looptijd van de machine weergeeft”.

Uitdaging 4

Sla het programma op met een nieuwe naam.

Schrijf een functie `void checkTime()` die wordt aangeroepen voor elke uitvoering van `loop()`, waardoor de taak van het weergeven van de tijd sinds het begin van de Arduino wordt uitgevoerd.

De tijd moet ten minste elke 0,1 seconde in het monitorvenster van de IDE (of op een LCD-display) worden weergegeven. De tijd moet worden weergegeven in seconden met twee decimalen.

Er zijn nog een paar globale variabelen over van de variabelen die in *Uitdaging 1* zijn gecreëerd, gerelateerd aan de LED's.
Het is een mooie oefening om volledig op te ruimen, zodat er alleen nog maar globale objecten gecreëerd hoeven te worden.

Uitdaging 5

Sla het programma op met een nieuwe naam.

Schrijf de definitie van een nieuwe klasse `LED` om het gedrag van de LED's in te vangen. De globale variabele die de toestand van de LED bijhield, zou nu `private` moeten worden. Je hebt ook een variabele nodig voor het pinnummer.

Schrijf een constructor `LED::LED(byte pin)` die het pinnummer in de privé-variabele opslaat. Verplaats de initialisatie van de pin van `setup()` naar de constructor.

Schrijf drie methoden: `turnOn()`, `turnOff()` en `toggle()` waardoor de LED aangaat, uitgaat, resp. in de omgekeerde toestand komt. (Deze methoden moeten zowel de pin-toestand instellen als de variabele bijwerken die de toestand volgt.)

Maak objecten `led1` en `led2` van het type `LED`, en test de drie methoden. Als je tevreden bent met het gedrag van je `LED`-klasse, ga je terug naar de oorspronkelijke taak.
(Je hebt hiervoor eigenlijk alleen de `toggle()`-methode nodig.)