

## Programmeringen afgør, hvor god din autonome robot bliver

Autonome robotter finder anvendelse mange steder. Langt de fleste rumsonder må klare sig selv, da signalet fra Jorden tager lang tid at få frem, og fjernstyring er ikke muligt. Mere jordnære robotter til støvsugning og græsslåning er også autonome.

I første afsnit blev hardware præsenteret. Vores robot består af en lyssensor og to motorer samt en Arduino. For at gøre robotten autonom, skal vi her i anden afsnit se nærmere på programmeringen. Robottens program afgør, hvor godt vores robot er i stand til at finde rundt på banen på egen hånd.

### Lidt om Arduino-programmering

I første afsnit var vi kort inde på det, og vi skal heller ikke igennem et større kursus i Arduino-programmering. Programmeringssproget er C++ med en række begrænsninger. For eksempel er exceptions slået fra. Grunden er, at exceptions ofte bruger mere plads, end hvad godt er – en Arduino har ikke meget plads at gøre godt med. Som C eller C++-programmør tænker du nok, at du altid skal implementere funktionen main. I et program til en Arduino skal du ikke implementere main.

Der er to andre funktioner, som du altid skal implementere: setup og loop. Som navnet antyder, afvikles setup som det første. Her kan du sætte alt op, f.eks. kan du sætte hastigheden på den serielle port og initialisere globale variabler.

### Se også: Sikker programudvikling med C++

Når setup er færdig, kommer loop i spil. Når loop er afviklet, begynder funktionen forfra. Med andre ord, koden i loop afvikles i en uendelig løkke (loop er engelsk og betyder løkke). Mange programmører er vant til at tænke over byggesystem. C++-programmører har traditionelt brugt Make, mens Java-folk bruger Ant, Maven eller måske det hippe Gradle. Arduinos udviklingsmiljø tager sig af den opgave, og inkluderer du et eller flere biblioteker, skal du ikke selv have styr på byggeprocessen. De inkluderede biblioteker vil blive taget med helt automatisk.

### Byg en robot med et lille styresystem

Godt nok er Arduino en lille microcontroller, og det er ikke det store styresystem, som følger med. Faktisk er der ikke tale om et styresystem, men mere et mindre runtime-miljø med begrænset funktionalitet.

Langt de fleste styresystemer i dag kan afvikle flere programmer samtidig. Endvidere er det ofte muligt, at et kørende program består af to eller flere tråde. En tråd er en kopi af programmet, og hver tråd afvikles uafhængig af de andre tråde. Men trådene er i stand til at tilgå fælles data og koordinere, hvad du gør.

### Se også: Byg en robot med et Arduino kit

Der er flere fordele ved at strukturere et program til at bruge flere tråde. Ofte tager det langt tid at læse værdien fra en sensor. Tid skal her ses i forhold til, hvor hurtig en cpu er. Hver klok-cyklus tabt på at vente på en sensor, er tid tabt, hvor cpu'en aktivt kunne styre robotten. På 1 millisekund kan en cpu med en klokfrekvens på 1 MHz nå op til 1000 instruktioner. Ved at dele programmet op i tråde, kan en tråd vente på en sensor, men en anden tråd aktivt tager beslutninger om, hvordan robotten skal opføre sig.

En af ulemperne ved at bruge flere tråde er, at det er meget sværere at fejlrette. Komplexiteten vokser mindst en størrelsesorden, når du introducerer tråde i dit program! Heldigvis findes der flere glimrende Arduino-biblioteker, som håndterer multitrådet programmering. Nogle biblioteker er avancerede med mange funktioner, andre små med lidt funktionalitet. I infoboksen "Få mere at vide" finder du links til flere af dem.

## Arduino-Scheduler

Til vores robot er valget faldet på Arduino-Scheduler. Personligt synes jeg ikke, at det er det mest elegante bibliotek. Men det har den fordel, at det har meget funktionalitet samtidig med, at det understøtter mange forskellige Arduino-boards (herunder Funduino som er boardet i vores robot).

Installationen af Arduino-Scheduler er ganske let.

Har du en Github-konto, kan du klonе mikaelpatel/Arduino-Scheduler direkte i folderen med dine biblioteker. Den folder finder du under Arduino med navnet libraries. Hvis du ikke lige ved, hvor din Arduino-folder er, kan du se det i Preferences i udviklingsmiljøet.

Alternativt kan du downloade en zip-fil fra <https://github.com/mikaelpatel/Arduino-Scheduler> og installere den gennem menupunktet Sketch – Include Library – Add .ZIP file.

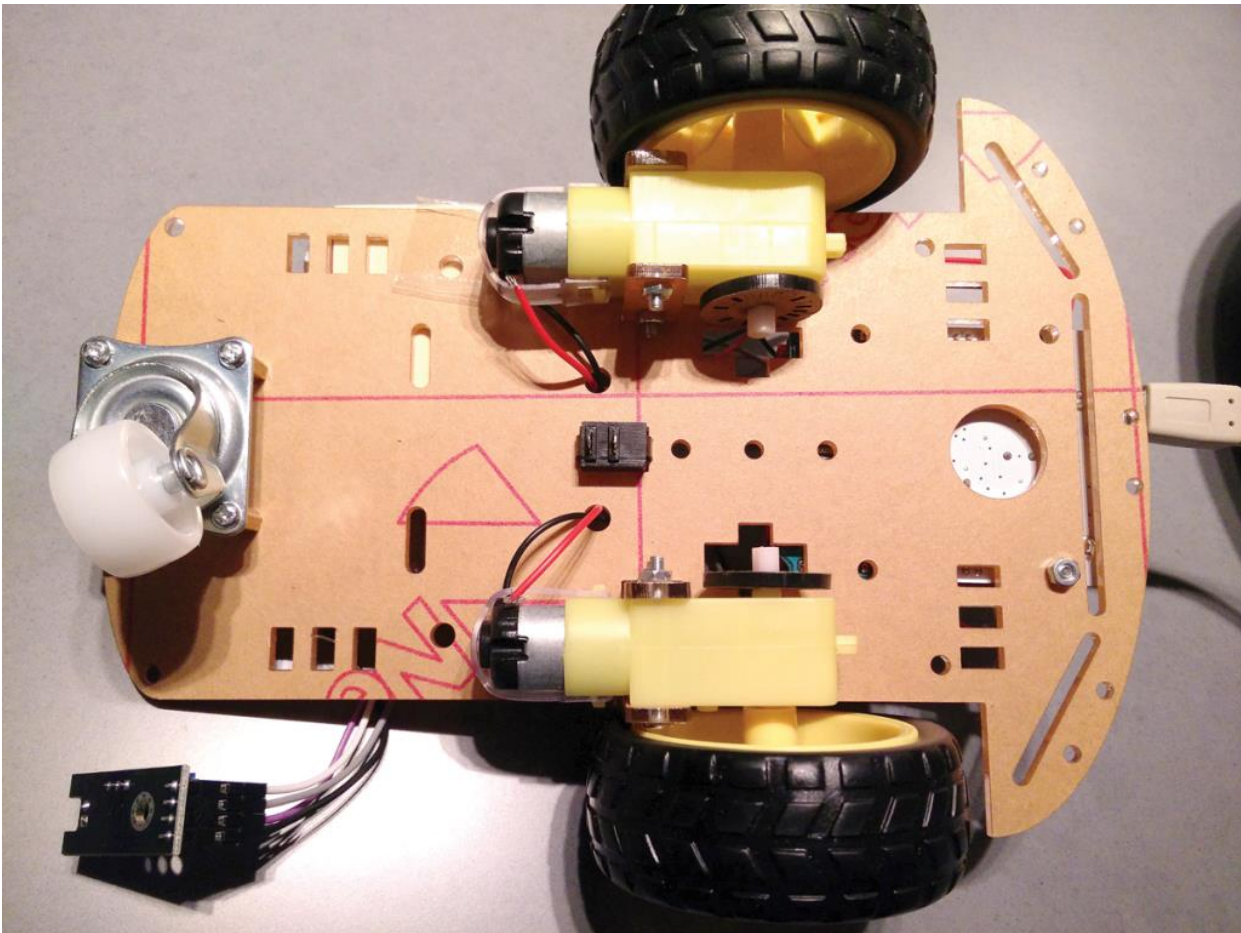
For at bruge Arduino-Scheduler i dit program skal du inkludere Scheduler.h. Antallet af tråde, som Arduino-Scheduler kan håndtere, afhænger af hvilket board, du benytter. Funduino er baseret på AT Mega 2560, og det er muligt at have op til 48 tråde. Et skift mellem to tråde (også kaldet et context switch) tager omkring 200 instruktioner!

## Tråde

Hele ideen med at bruge Arduino-Scheduler er, at vi gerne vil have flere tråde i vores program. I infoboks "Tråde og kritiske regioner" finder du et meget kort program. Det består af to tråde. Hovedtråden er altid implementeret som loop, men du kan tilføje flere tråde ved at kalde Scheduler.start(NULL, funktion), hvor funktion er navnet på en funktion i dit program. Som ved loop, skal trådene i Arduino-Scheduler ses som kroppen i en uendelig løkke. Dette valg samt at lade loop være en tråd er i min optik mindre elegant.

### Se også: Håbløse robotter i duel

For at gøre det tydeligere er begge tråde implementeret i to funktioner, og loop kalder blot den ene funktion. Som du kan se, tager funktionerne ingen argumenter, og de har heller ingen returnværdi. Det betyder, at du er nødt til at bruge globale variabler til at overføre og returnere værdier.



Robotten har motorer og hjul.

### Kritiske regioner

Forestil dig, at du har to værdier, som en tråd (skriveren) kan overføre til en anden tråd (læseren). Værdierne hænger sammen, da programmets opførsel ellers vil kunne ende i en fejltilstand. Et eksempel er, at du vil angive, hvor hurtigt de to motorer skal køre. Har du kun den ene værdi, kan du risikere, at robotten drejer og ender med at køre ad banen.

De to værdier er globale variabler og i langt de fleste tilfælde, vil det ikke give problemer. Men det er ikke svært at forestille sig situationen, hvor læseren har læst den ene værdi, og skriveren når at opdatere begge værdier, inden læseren har læst den anden værdi. Med andre ord, det kræver en vis portion koordinering mellem de to tråde.

### Se også: **Byg en robot med et Arduino kit**

De to værdier eller globale variabler er et eksempel på en delt ressource.

Det er vigtigt, at adgangen til delte ressourcer koordineres. Redningen fra datalogien hedder semaforer. Det er variabler/objekter, som du kan bruge til at koordinere tråde. De områder i programmet, som kan tilgå en delt ressource, omtales som en kritisk region.

## Et lille trick

At kopiere værdien af den delte ressource value over i en lokal variabel. På den måde bliver den kritiske region så lille som muligt. Jo mindre kritiske regioner er, jo mindre sandsynligt er det, at andre tråde må vente på adgang til den delte ressource.

Kaldet `mutex.wait()` vil få tråden til at vente, hvis en anden tråd er inde i en kritisk region. Ved afslutning af regionen, gives adgangen til value igen frit ved at kalde `mutex.signal()`. Når programmet kører, kan du åbne den serielle monitor og se, hvordan de to tråde hhv. opdaterer og henter værdien af value.

## Hvad programmet består af

Der er mange måder at konstruere programmet til vores robot på. For at gøre det mere overskueligt, vil programmet bestå af en række tråde og delte ressourcer. For det første er der kommandocentralen. Den tager data ind fra de forskellige sensorer. På baggrund af data og hvor robotten har været tidligere, har kommandocentralen en beslutning om hvilken vej, robotten skal køre.

Motorerne er det hardware, som driver robotten fremad. En særlig tråd omsætter kommandocentralens beslutninger til input til motorerne.

Hver sensor har en tråd. En sensortråd vil løbende læse værdier fra sensoren og overlade værdierne til kommandocentralen.

### Se også: **Byg en robot med et Arduino kit**

Ved at have flere løstforbundne tråde, kan trådene køre i hvert deres tempo. Ikke alle sensorer er lige hurtige til at foretage en måling. Hvis alle sensorer skal aflæses med samme frekvens, vil den langsomste tråd være den begrænsende faktor.

## Styring af robotten

Vores robot har to motorer. Den eneste måde, robotten kan dreje på, er at lade den ene motor køre hurtigere end den anden. Med andre ord, styringen af robotten kræver to værdier.

Det er let at indse, at de to værdier til motorerne samlet set er en delt ressource. Læser motortråden hastigheden for den venstre motor, mens kommandocentralen tager en ny beslutning om retning, inden motortråden har læst værdien for den højre motor, har vi balladen. Kommandocentralen tror, at robotten kører i én retning, men motortråden har sendt robotten i en anden retning.

I infoboksen "Samlet program" på næste side kan du se en semafor ved navn `motor_mutex`. Det bruges netop til at koordinere kommandocentralen (tråden `command_center`) og motortråden (tråden `control_motor`). De to værdier som bruges til at sætte hastigheden på de to motorer er slået sammen i en struct, men de kunne godt have været to adskilte variabler.

### Se også: **Håbløse robotter i duel**

## At følge en linje

Banen til DTU Robocup (og mange andre robot-konkurrencer) har en linje, som robotten skal følge. Ofte er banen hvid med en sort stribe (eller omvendt) for at give stort mulig forskel. At følge en linje/stribe kaldes i robot-verdenen en "line follower".

Robotten kan se striben ved at bruge en fotosensor. En måling uden for striben vil give en høj værdi, men en måling af striben vil give en lav værdi. Det er også muligt, at robotten har placeret sig så fotosensoren måler både lidt af striben og lidt af baggrunden. Værdien vil så være et sted mellem lav og høj.



Robotten bruger en sonar til at "se".

I programmet i infoboks Samlet program finder du i tråden `light_sensor` kode som foretager en måling med en fotosensor. Den valgte sensor kan tjekke, om der er nok baggrundslys (`digitalRead(DIGITAL_SENSOR_PIN)`). Den returnerede værdi er enten HIGH eller LOW, dvs. et binært output. Hvis der er for lidt baggrundslys, tænder robotten for en lysdiode. Det minder lidt om, hvordan den automatiske blitz i dit kamera virker.

Når semaforen oprettes, gøres det med en værdi for, hvor mange gange en delt ressource kan tilgås samtidig. Når en tråd skal påbegynde en kritisk region, tjekker den, om der allerede er tråde, som tilgår den delte ressource.

Hvis ressourcen allerede tilgås af mange tråde, må tråden vente. Når en af de andre tråde forlader den kritiske region, vil den ventende tråd få lov til at komme ind i regionen.

Arduino-Scheduler understøtter kun en type af semaforer, nemlig heltalssemaforer. Vælger du ikke, hvor mange gange en ressource kan deles, er der kun en tråd, som kan tilgå ressourcen. En sådan binær semafor kaldes ofte for en mutex (mutual exclusion, gensidig udelukkelse).

I infoboks "Tråde og kritiske regioner" finder du et lille eksempel på et Arduino-program. Programmet har to tråde, og de deler den globale variable `value`. Ved at bruge semaforen mutex kan de to tråde koordinere adgangen til `value` således, at tråden `first_task` kan opdatere ressourcen.

Selve målingen af lys sker derefter. Den valgte lyssensor har en opløsning på 10 bits, dvs. værdien vil ligge mellem 0 og 1023. Den målte værdi gemmes i den globale variabel `light_value`. Kun selve skrivning af værdien til den globale variabel er en kritisk region.

Der findes mange måder at følge en linje på. Det mest enkle er at se på, om værdien fra lyssensoren er over en tærskelværdi. Er værdien det, er robotten ikke på linjen, og den bliver nødt til at dreje. Ulempen ved denne binære beslutningsstrategi er, at robotten ofte ender med at køre i zigzag. Det giver en ujævn kørsel med en lav fast og et højt batteriforbrug.

En zigzagbevægelse for en robot er acceptabel, når den kører fremad, men i sving bliver det svært for robotten at få drejet korrekt. Jo skarpere svinget er, jo mere sandsynligt er det, at robotten mister orienteringen og kommer væk fra linjen.

## PID Line Follower

En mere avanceret metode til at følge en stribe er ved at bruge en PID. PID er en forkortelse for Proportional, Integral og Derivative. Ideen er at bruge den målte lysværdi til at se, hvor godt robotten er på linjen. Er værdien i den lave ende, er robotten ved at være væk fra linjen, mens en høj værdi indikerer, at den er midt på linjen.

### **Se også: Byg en robot med et Arduino kit**

Ved at huske hvilken retning, robotten kører i, kan en sensorværdi mellem lav og høj vise, om robotten er på vej væk fra linjen eller på vej ind mod midten. Derved kan du hurtigere korrigere retningen på robotten, så den holder sig til linjen – især i sving. Med andre ord, robotten vil ikke have samme tendens til at køre i zigzag.

## Afslutning

I forrige artikel i sidste udgave af Alt om DATA og denne artikel har du set, hvordan en lille robot kan konstrueres. Autonome robotter er fascinerende og svære at få succes med. Det kræver indsigt i flere discipliner: mekanik, elektronik, el og software.

Heldigvis er Arduino-boards, sensorer og motorer i dag så billige, at det ikke koster spidsen af en jetjager at komme i gang.

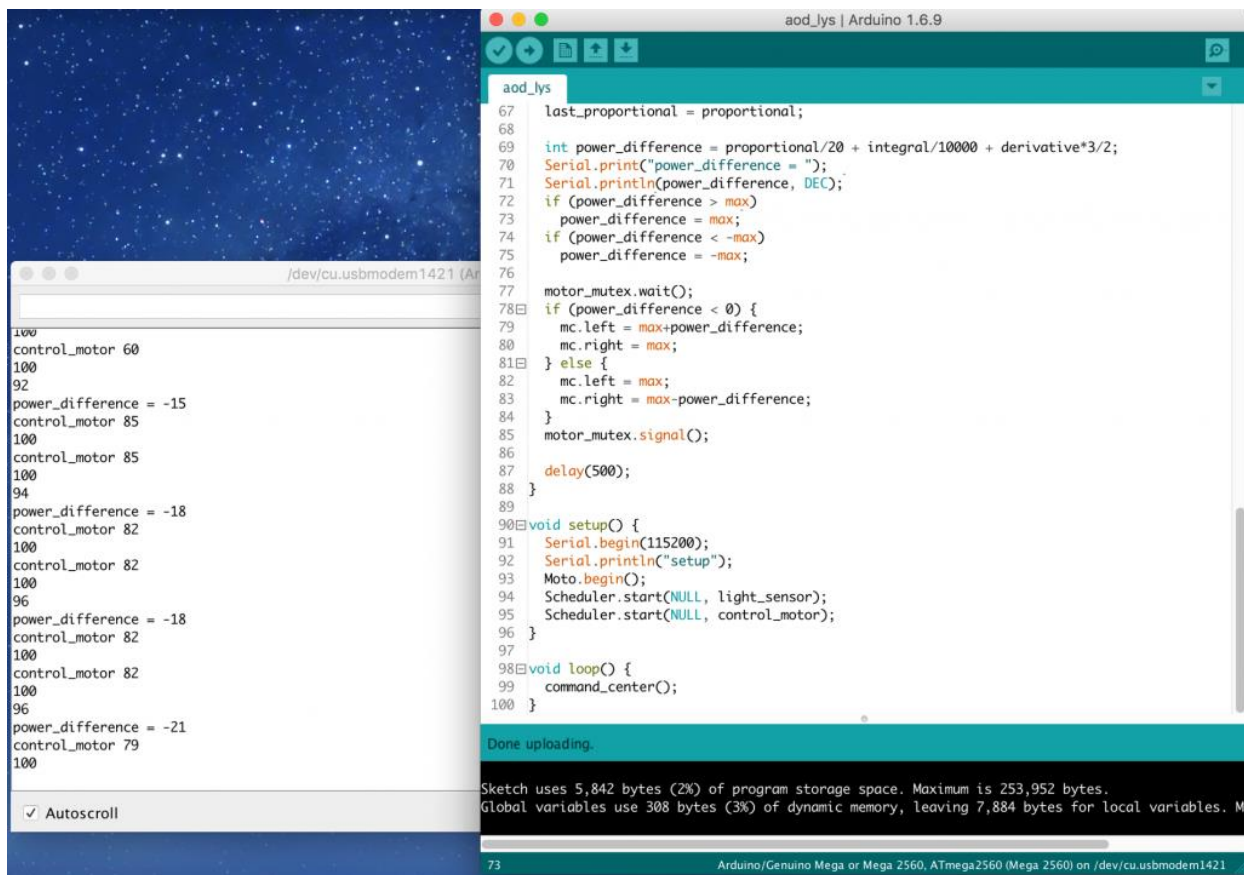
Robotten og programmet, som er præsenteret, vil sandsynligvis udvikle sig i den kommende tid.

Du er velkommen til at følge med på <https://github.com/kneth/ArduRobo>. Der vil nok blive tilføjet sensorer og andet hardware.



## Kodestykker til at bygge en Robot

### Tråde og kritiske regioner



```
67 last_proportional = proportional;
68
69 int power_difference = proportional/20 + integral/10000 + derivative*3/2;
70 Serial.print("power_difference = ");
71 Serial.println(power_difference, DEC);
72 if (power_difference > max)
73   power_difference = max;
74 if (power_difference < -max)
75   power_difference = -max;
76
77 motor_mutex.wait();
78 if (power_difference < 0) {
79   mc.left = max+power_difference;
80   mc.right = max;
81 } else {
82   mc.left = max;
83   mc.right = max-power_difference;
84 }
85 motor_mutex.signal();
86
87 delay(500);
88 }
89
90 void setup() {
91   Serial.begin(115200);
92   Serial.println("setup");
93   Moto.begin();
94   Scheduler.start(NULL, light_sensor);
95   Scheduler.start(NULL, control_motor);
96 }
97
98 void loop() {
99   command_center();
100 }
```

Done uploading.

Sketch uses 5,842 bytes (2%) of program storage space. Maximum is 253,952 bytes.  
Global variables use 308 bytes (3%) of dynamic memory, leaving 7,884 bytes for local variables. Memory not used.

73 Arduino/Genuino Mega or Mega 2560, ATmega2560 (Mega 2560) on /dev/cu.usbmodem1421

```
#include <Scheduler.h>
```

```
#include <Scheduler/Semaphore.h>
```

```
Semaphore mutex;
```

```
int value;
```

```
/* Tasks */
```

```
void first_task() {
```

```
static int i = 1;
```

```
Serial.print("first_task ");
```

```
mutex.wait();
```

```
value = i;
```

```
Serial.println(i);
```

```
mutex.signal();
```

```
delay(1000);
```

```
i++;
```

```
}
```

### Se også: Sikker programudvikling med C++

```
void second_task() {
```

```
Serial.print("second_task ");
```

```
mutex.wait();
```

```
int i = value;
mutex.signal();
Serial.println(i);
delay(2000);
}

void setup() {
  Serial.begin(115200);
  Serial.println("setup");
  Scheduler.start(NULL, second_task);
}

void loop() {
  first_task();
}
```

**Samlet program**



```

#include<Scheduler.h>
#include<Semaphore.h>
#include<ArduMoto.h>
}
ArduMoto Moto;
void command_center() {
Semaphore light_mutex;
const int max = 100;
Semaphore motor_mutex;
static int last_proportional = 0;
static int integral = 0;
struct motor_command {
int left;
unsigned int position;
int right;
},
light_mutex.wait();
position = light_value;
typedef struct motor_command MotorCommand;
light_mutex.signal();
MotorCommand mc;
int proportional = ((int)position) - 511; // lyssensor mellem 0 og 1023
int derivative = proportional - last_proportional;
last_proportional = proportional;
// proportional = proportional; center overskriver dem
int left = mc.left;
int power_difference = proportional/20 + integral/10000 + derivative*3/2;
int right = -mc.right;
if (power_difference > max)
motor_mutex.signal();
power_difference = max;
if (power_difference < -max)
power_difference = -max;
motor_mutex.wait();
if (power_difference < 0) {
mc.left = value + power_difference;
mc.right = max;
#define ANALOG_SENSOR_PIN A0
} else {
#define DIGITAL_SENSOR_PIN 3
mc.left = max;
#define LEDPIN 13
mc.right = max - power_difference;
}
void light_sensor() {
motor_mutex.signal();
int switch_state = digitalRead(DIGITAL_SENSOR_PIN);
if (switch_state == LOW) {
delay(500);
digitalWrite(LEDPIN, HIGH);
} else {
digitalWrite(LEDPIN, LOW);
}
}
void setup() {
Moto.begin();
Scheduler.start(NULL, light_sensor);
light_mutex.wait();
Scheduler.start(NULL, control_motor);
light_value = analogRead(ANALOG_SENSOR_PIN);
}

void loop() {
command_center();
}

```

