

## LS 3 Sensoren am Roboter

Fahrerlose Transportsysteme sind aus vielen Betrieben nicht mehr wegzu-denken. Ihre wirtschaftlichen und praktischen Vorteile sowie die technische Weiterentwicklung haben Robotern neue Bereiche erschlossen. Oft ersetzen sie bereits herkömmliche Transportgeräte wie Gabelstapler und Hubwagen.

Wenn ein Fahrzeug automatisch, also ohne menschlichen Fahrer, betrieben werden soll, ist die Navigation eine der wesentlichen Aufgaben, die durch folgende Punkte gelöst werden muss:

- durchdachte **Software**
- geeignete **Sensorik**



Das Prinzip, anhand von Sensorwerten Entscheidungen zu treffen, nutzen wir auch für unsere EV3 Roboter. Sensoren bringen sie dazu, sich selbstständig im Raum orientieren zu können.

Wir benutzen bspw. den Ultraschallsensor, um zu erkennen, ob sich ein Hindernis vor dem Roboter befindet oder wir benutzen den Helligkeitssensor um zu entscheiden, ob sich der Roboter auf einer Linie oder daneben befindet.

Alle Sensoren und die benötigten Befehle finden Sie in der Befehlsliste.



Fotos aus Werbevideo für Transprotroboter Weasel

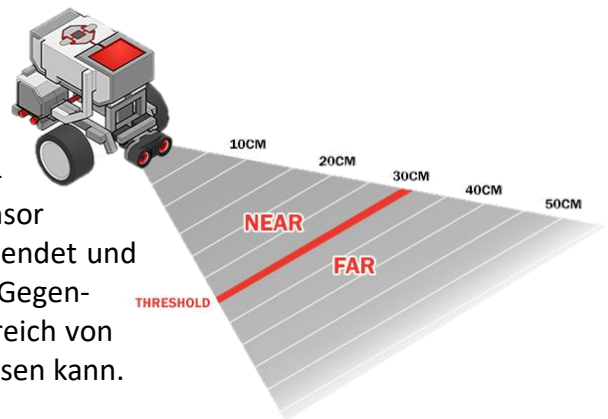
## LS3.1 Losfahrsperre

Der Roboter soll beim Programmstart zunächst prüfen, ob der Weg vor ihm auch frei ist. Hierfür verfügt der Roboter über einen Ultraschallsensor.



### LS3.1.1 Funktionsweise Ultraschallsensor

Mit dem Ultraschallsensor kann der Roboter Entfernungen messen. Theoretisch sind damit Entfernungen von 0cm bis 250cm messbar. Da der Ultraschallsensor jedoch mit einem seiner zwei Augen Ultraschallwellen sendet und diese mit dem anderen empfängt (falls diese von einem Gegenstand reflektiert werden) ergibt sich ein realistischer Bereich von etwa 5cm bis 250cm in dem der Roboter zuverlässig messen kann.



#### Ultraschallsensor (Port 4)

*Distanz (in cm) zwischen Sensor und Objekt erfassen:*

```
getUSDistance(S4);
```



*Mgl. Rückgabewerte*

float 0.0 bis 250.0

Anhand des Funktionsaufrufes **getUSDistance(S4)** kann der aktuelle Messwert des Ultraschallsensors abgefragt werden. Die Funktion benötigt dazu den entsprechenden Port, an dem der Sensor angeschlossen ist (hier S4 – vgl. Standardkonfiguration aus LS1). Anders als die Funktionen welche Sie bisher kennengelernt haben, gibt diese Funktion sogar einen Wert zurück – nämlich den aktuellen Sensorwert des Ultraschallsensors (**Rückgabewert**).

Das bedeutet, dass an der Stelle wo der Funktionsaufruf erfolgt, nach der Ausführung ein Rückgabewert steht. Bei einem Hindernis in 45cm Entfernung wäre das der Wert 45.

Das folgende Programm gibt den Sensorwert des Ultraschallsensors, welcher beim Start des Programmes gemessen wurde für 5 Sekunden auf dem Display aus.

```
6 // #pragma config(Motor1, motorA, leftMotor, tmotorEV3_Motor, PIDControl, driveLeft, encoder)
7 // #pragma config(Motor2, motorB, rightMotor, tmotorEV3_Motor, PIDControl, driveRight, encoder)
8 // #pragma config(Motor3, motorC, rightMotor, tmotorEV3_Motor, PIDControl, driveRight, encoder)
9
10 task main()
11 {
12     //Auslesen und Ausgeben eines Sensorwertes (hier Eingang S4) an der Platzhalterposition %d
13     displayTextLine(1, "Sensor Wert: %dcm", getUSDistance(S4));
14     sleep(5000);
15 }
```

### LS3.1.2 Aufgabe: Ultraschallsensorwerte am Display ausgeben

Schreiben Sie das oben aufgeführte Programm selbst und probieren Sie es mit verschiedenen Entfernungen aus. (empfohlener Dateiname: Losfahrsperre\_Displayausgabe.c)

## LS3.1.3 Entscheidungen (wenn jemand im Weg steht, dann... sonst...)

Damit der Roboter nun weiß, ob er losfahren soll, oder nicht muss er eine Entscheidung treffen. Entscheidungen sind Verzweigungen im Programmablauf (bedingte Anweisungen, conditional statements) und dienen dazu, ein Programm in mehrere Pfade aufzuteilen. Beispielsweise kann so auf die Sensorwerte entsprechend reagiert werden. Je nachdem, welchen Wert der Sensor zurückgibt, ändert sich der Programmablauf.

### Einseitige Entscheidungen: if

Entscheidungen werden mit dem Schlüsselwort **if** begonnen, gefolgt von einer Bedingung in Klammern:

**Bedingungen** sind Ausdrücke, die stets in einer von zwei folgenden Entscheidung münden:

- Ja (= wahr, **true**, alles ungleich 0)
- Nein (= falsch, **false**, 0)

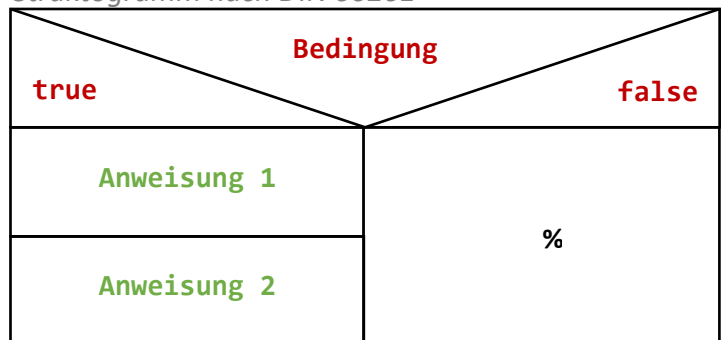
Es lassen sich alle Ausdrücke als Bedingung verwenden, deren Ergebnis logisch wahr oder falsch, also vom Datentyp **bool** ist oder in diesen umgewandelt werden kann. Ganzzahlige und Gleitkommatypen können nach **bool** umgewandelt werden. Ist ein Ausdruck gleich 0, so wird er als false interpretiert, andernfalls ist der Ausdruck true.

Alles außer 0 ist true.

*Syntax in C basierten Sprachen (auch in RobotC)*

```
if (Bedingung)
{
    Anweisung 1;
    Anweisung 2;
}
```

*Struktogramm nach DIN 66261*



Hat also der Ausdruck einen Wert, der sich von 0 unterscheidet bzw. wahr ist, ist die Bedingung erfüllt. Somit wird die in geschweiften Klammern eingeschlossene **Anweisungsfolge** (Anweisungsblock) ausgeführt - ansonsten einfach übersprungen.

### Beispiele für Einseitige Entscheidungen

```
//Wenn Helligkeitssensor einen Wert > 50 misst → Textausgabe
if ( getColorReflected(S3) > 50 )
{
    displayTextLine(1, "Es ist heller als 50");
}
```

```
//Wenn der Touch Sensor gedrückt ist (Wert 1) → Textausgabe
if ( getTouchValue(S1) == 1 )
    displayTextLine(1, "TouchSensor gedrückt");
```

Insgesamt gibt es sechs **Vergleichsoperatoren**:

==	identisch
<=	ist kleiner (oder) gleich
>=	ist größer (oder) gleich
<	ist kleiner
>	ist größer
!=	ist ungleich

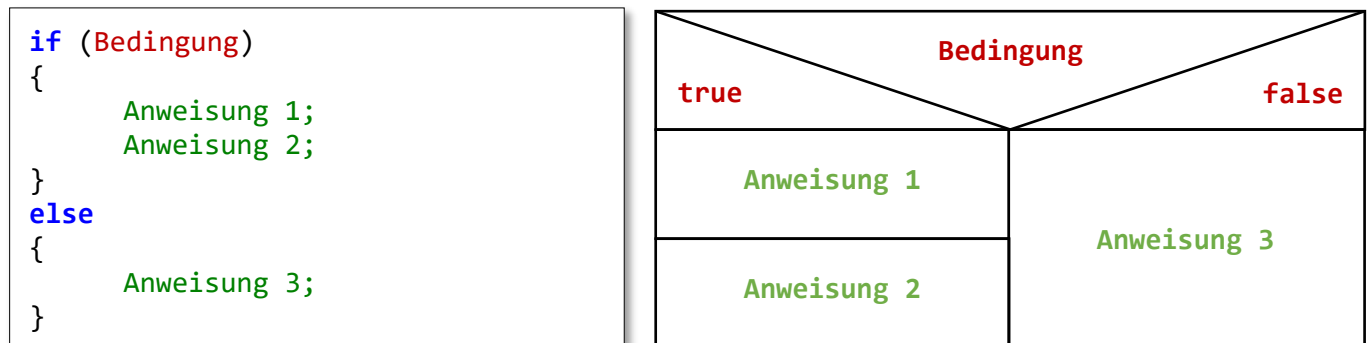
Die geschweiften Klammern { } (welche im vorangegangenen 2. Beispiel mit dem TouchSensor fehlen) können entfallen, solange nur eine Anweisung im true-Fall ausgeführt werden soll. Bei mehreren Anweisungen im true-Fall sind die Klammern dann jedoch zwingend nötig. Da dies oft vergessen wird, ist diese Schreibweise eine häufige **Fehlerquelle** und sollte vermieden werden. Zudem dienen die Klammern der Übersichtlichkeit des Programmcodes. Die Variante mit Klammern (wie im 1. Beispiel mit dem Helligkeitssensor) sollte also bevorzugt werden.

## Zweiseitige Entscheidungen: if ... else

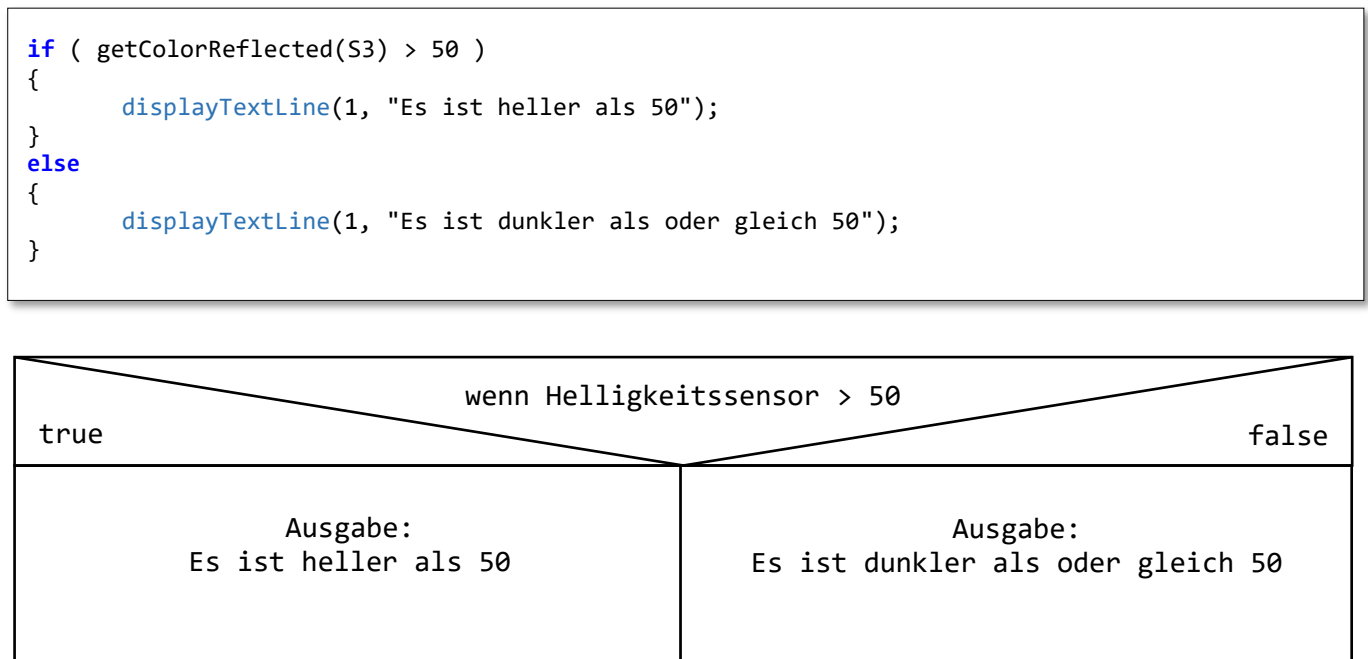
Die zweiseitige Entscheidung enthält neben **if** noch eine Erweiterung der Verzweigung durch das Schlüsselwort **else**:

- Ergibt der Ausdruck der Bedingung einen Wert ungleich 0 (wahrer Wert), wird die Anweisungsfolge des if-Teils ausgeführt
- Nimmt die Auswertung des Ausdrucks den Wert 0 (falscher Wert) an, wird die Anweisungsfolge des else-Zweiges ausgeführt.

*Syntax in C basierten Sprachen (auch in RobotC)    Struktogramm nach DIN 66261*

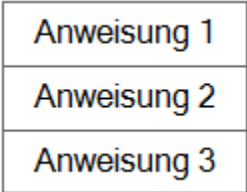
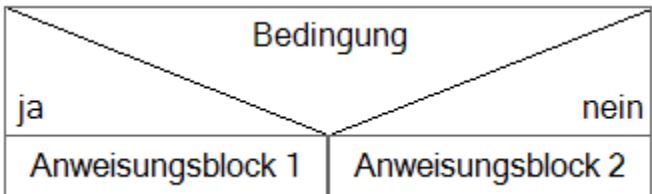
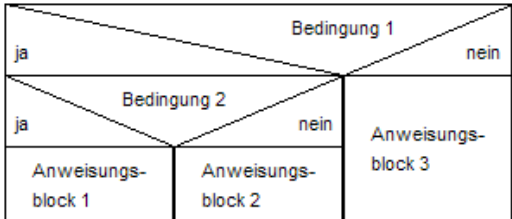


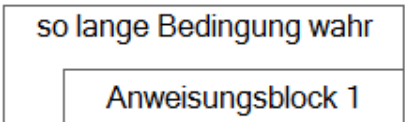
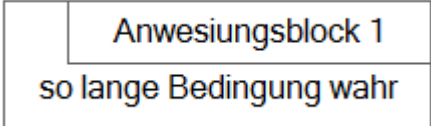
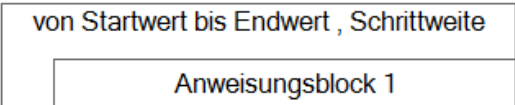
*Beispiel für Zweiseitige Entscheidungen mit entsprechendem Struktogramm*



## LS3.1.4 ShortFacts zu Struktogrammen

Struktogramme dienen der Darstellung von Programmen bzw. Programmentwürfen im Rahmen der strukturierten Programmierung. Die sog. Nassi-Shneiderman-Diagramme (kurz Struktogramme) sind in der DIN 66261 genormt. Struktogramme müssen immer rechteckig sein.

Grundelemente	
<b>Lineare Struktur</b>  Jede Anweisung wird in einem rechteckigen Strukturblock geschrieben	
<b>Verzweigung/Entscheidung</b>  Wenn eine Bedingung zutrifft wird der ja-Block ausgeführt, wenn nicht, wird der nein-Block ausgeführt. Die beiden Blöcke können aus mehreren Anweisungen bestehen oder können im nein-Fall auch leer bleiben (=einseitige Entscheidung).	
<b>Verschachtelte Entscheidungen</b>  Es folgt eine weitere Bedingung. Die Verschachtelung ist ebenso im Nein-Fall noch möglich.	

Exkurs: Weitere mögliche Struktogrammblocke	
<b>Kopfgesteuerte Schleife</b>  Der Anweisungsblock wird so lange durchlaufen, wie die Bedingung zutrifft	
<b>Fußgesteuerte Schleife</b>  Im Gegensatz zur kopfgesteuerten Schleife wird der Anweisungsblock hier mindestens einmal durchlaufen, weil die Bedingungsprüfung erst im Anschluss an den Anweisungsblock stattfindet.	
<b>Zählergesteuerte Schleife</b>  Die Anzahl der Schleifendurchläufe wird durch eine Zählvariable festgelegt. Im Schleifenkopf werden der Startwert der Zählvariablen, der Endwert und die Veränderung der Zählvariablen nach jedem Schleifendurchlauf angegeben.	

## Beispielstruktogramm des start.c Programmes

Motoren B und C einschalten (50 Prozent vorwärts)
weitere Befehle für 1000ms pausieren
Motoren B und C einschalten (20 Prozent vorwärts)
weitere Befehle für 2000ms pausieren
Motoren B und C einschalten (0 Prozent = ausschalten bzw. Motoren stoppen)

### LS3.1.5 Aufgabe: Struktogramm zur Losfahrsperre

#### EINZELARBEIT mit anschließender Präsentation:

Erstellen Sie ein Struktogramm, welches das Programm der Losfahrsperre darstellt. Bedenken Sie – wie in der Ausgangssituation beschrieben –, dass der Roboter direkt am Start prüfen soll, ob ein Hindernis im Weg ist (bspw. innerhalb der ersten 30cm). Nur wenn der Weg frei ist, kann der Roboter etwas nach vorne fahren.

### LS3.1.6 Aufgabe: Programmieren der Losfahrsperre

#### PARTNERARBEIT mit anschließender Code-Vorstellung:

Schreiben Sie nun ein Programm (Dateiname: Losfahrsperre.c) welches Ihr geplantes Struktogramm umsetzt und testen Sie es ausführlich.

**Für Schnelldenker:** Kommentieren Sie Ihren Code bitte entsprechend, damit auch schwächere Schüler gut verstehen können, was in welcher Zeile genau passiert.

### LS3.1.7 ShortFacts zu Kommentaren

#### Kommentare in C basierten Sprachen

In allen Programmiersprachen gibt es die Möglichkeit im Quelltext Notizen, bzw. Kommentare, zu verfassen. Diese sind Passagen innerhalb des Programms die vom Compiler ignoriert werden. Hauptsächlich kommentiert man seinen Code für andere – aber eben auch für sich selbst. Denn nach ein paar Wochen werden Sie möglicherweise Ihren eigenen Quelltext nicht mehr ohne weiteres verstehen. Kommentare sind also Hinweise die Ihnen und anderen helfen besser und vor allem **schneller** zu verstehen, was das Programm tut.

*Ein Programm ist meistens nur so gut, wie seine Kommentare.*



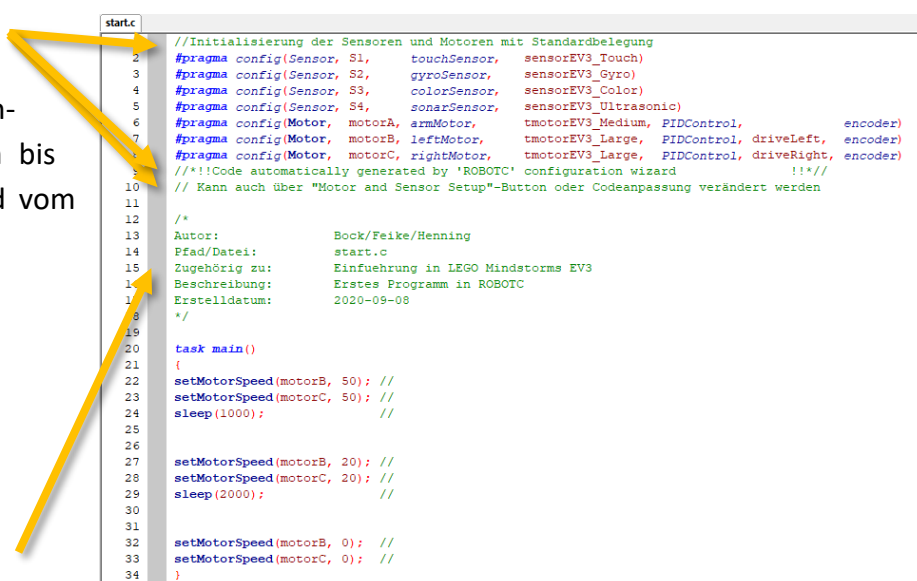
Wie zu kommentieren ist, sehen Sie anhand der folgenden Beispiele. Dieses Vorgehen gehört zu den **Programmierrichtlinien** im AWP-Unterricht hier an der Berufsschule - dazu zählen auch Benennung von Projekt- und Code-Dateien, Quelltext-Einrückung usw.

In Ihrer Firma gibt es u. U. ebenso solche Programmierrichtlinien, die von denen hier gezeigten abweichen können - erkundigen Sie sich doch einfach einmal danach.

In C basierten Sprachen stehen i.d.R. zwei verschiedene Arten an Kommentarsymbolen zur Verfügung:

## • Einzeilige Kommentare

Die zweite Möglichkeit wird mit den Zeichen `//` eingeleitet. Alles was danach bis zum Zeilenende folgt, wird vom Compiler als Kommentar betrachtet. Auch hier darf zwischen den Zeichen `//` kein Leerzeichen stehen



```

1 //Initialisierung der Sensoren und Motoren mit Standardbelegung
2 #pragma config(Sensor, S1, touchSensor, sensorEV3_Touch)
3 #pragma config(Sensor, S2, gyroSensor, sensorEV3_Gyro)
4 #pragma config(Sensor, S3, colorSensor, sensorEV3_Color)
5 #pragma config(Sensor, S4, sonarSensor, sensorEV3_Ultrasonic)
6 #pragma config(Motor, motorA, armMotor, tmotorEV3_Medium, PIDControl, encoder)
7 #pragma config(Motor, motorB, leftMotor, tmotorEV3_Large, PIDControl, driveLeft, encoder)
8 #pragma config(Motor, motorC, rightMotor, tmotorEV3_Large, PIDControl, driveRight, encoder)
9 //!!!Code automatically generated by 'ROBOTC' configuration wizard !!!
10 // Kann auch über "Motor and Sensor Setup"-Button oder Codeanpassung verändert werden
11
12 /*
13 Autor: Bock/Feike/Henning
14 Pfad/Datei: start.c
15 Zugehörig zu: Einführung in LEGO Mindstorms EV3
16 Beschreibung: Erstes Programm in ROBOTC
17 Erstelldatum: 2020-09-08
18 */
19
20 task main()
21 {
22     setMotorSpeed(motorB, 50); //
23     setMotorSpeed(motorC, 50); //
24     sleep(1000); //
25
26     setMotorSpeed(motorB, 20); //
27     setMotorSpeed(motorC, 20); //
28     sleep(2000); //
29
30     setMotorSpeed(motorB, 0); //
31     setMotorSpeed(motorC, 0); //
32 }
33
34

```

## • Mehrzeilige Kommentare

Der Mehrzeilen-Kommentar wird eingeleitet durch die beiden Zeichen `/*` und durch die Zeichen `*/` beendet. Zwischen diesen beiden Zeichen darf kein Leerzeichen stehen. Und alles was zwischen diesen Zeichenfolgen steht wird vom Compiler ignoriert.

Mithilfe von Kommentaren ist es vorallem möglich, einzelne oder mehrere Anweisungen z. B. für Testzwecke außer Kraft zu setzen (vgl. Kommentierübung aus LS1). Dazu werden die Anweisungen einfach „auskommentiert“. Bei einzelnen Befehlen eben mit den zwei Schragstrichen vor dem Befehl wodurch der Rest der Zeile zu einem Kommentar wird.

24 `//sleep(1000);`

## LS3.2 Displayausgabe des korrekten Messwertes der Losfahrsperre

Basierend auf Ihrer umgesetzten Losfahrsperre, soll der Roboter nun über das Display mit seiner Umgebung kommunizieren.

Ist der Weg frei ( $>30\text{cm}$ ) soll er bspw. anzeigen:  
„Ich fahre los! Abstand ist 32cm“

Ist der Weg nicht frei ( $\leq 30\text{cm}$ ) soll er bspw. anzeigen:  
„Ich fahre nicht los! Abstand ist 28cm“

WICHTIG: Der genau Wert, der für die Entscheidung ausschlaggebend war muss also ausgegeben werden. Es darf nicht zweimal gemessen werden.



### LS3.2.1 Variablen

Um Daten kurzzeitig (oder auch längerfristig) speichern zu können, muss zuerst Speicherplatz geschaffen (genauer: reserviert) werden. Zusätzlich muss auf diesen reservierten Platz noch zugegriffen werden können. Hierfür gibt es **Variablen**. Sie stehen für den Inhalt eines Speicherplatzes. Zuerst muss eine Variable **definiert** werden. Nach der Definition ist Speicherplatz für die Variable entsprechend ihrer Größe im Arbeitsspeicher reserviert. Danach kann ihr ein Wert zugewiesen werden. Welcher Wert zugewiesen werden darf, ist vom Variablentyp abhängig.

#### Definition von Variablen

Wenn man beispielsweise folgende Variable **definiert**...

```
int i_zahl;
```

..., dann „sagt“ man dem Compiler: „Compiler, ich will, dass du Speicherplatz für eine Integer-Variable mit dem Namen `i_zahl` reservierst.“

Wenn eine Variable **definiert und** gleich danach **initialisiert** (d. h. ein Wert zugewiesen) wird, ...

```
int i_zahl = 7;
```

..., dann teilt man dem Compiler mit: „Ich will, dass diese Variable mit Namen `i_zahl` vom Typ `int` (bis hierher ist es eine Definition) mit dem Wert 7 vorbelegt werden soll.“

Eine **Zuweisung** alleine (falls `int i_zahl` schon vorher definiert wurde) würde folgendermaßen aussehen:

```
i_zahl = 7;
```

Jetzt „weiß“ der Compiler, dass der bereits definierten (aber möglicherweise noch nicht initialisierten) Variablen `i_zahl` der Wert 7 zugewiesen werden soll.



Die Programmierrichtlinie für den AWP-Unterricht sieht vor, dass der eigentliche Variablenname noch den Datentyp als Buchstabe vorangestellt bekommt. Ferner sollen immer sprechende Namen, die den Variableninhalt gut beschreiben, verwendet werden. Im letzten Beispiel also statt:

```
int a = 6;           →   int i_Sensorwert = 6;
```

oder statt

```
bool x = 0;          →   bool b_gedrueckt = 0;
```

Für die weiteren Aufgaben in RobotC sind die beiden Datentypen `int` und `bool` ausreichend.

## Exkurs: Weitere Beispiele von Variablendefinitionen mit passender Initialisierung

```
bool b_Erkrankt = 1;
bool b_HatFuererschein = false;
float f_PreisMilchtuete = 0.99;
double d_Pi = 3.14159265359;
```

Um einer `char`-Variable einen Buchstaben zuzuweisen darf man nicht

```
char c_zeichen = "a";
```

sondern

```
char c_zeichen = 'a';
```

schreiben, denn "a" wäre aufgrund der doppelten Anführungszeichen ein String (also eine Kette von mehreren Zeichen) und 'a' (aufgrund der beiden einzelnen Hochkommas) ein einzelnes Zeichen.

Unter einer Variable versteht man:  
Festlegung des Datentyps, Namen und reserviere Speicher.

## Übung zu Variablen:

- Überlegen Sie sich, wie man die Variablendefinition und die -initialisierung mit einer Lagerkiste in einem Hochregallager (z. B. bei Ikea) vergleichen kann (siehe Vorlage unter \\10.0.27.12\awp\_10\LS3.2.1\Pinnwand\_Vorlage.pptx).
- Bitte kreuzen sie an, ob die folgende Aussage wahr oder falsch ist:

Aussage (immer bezogen auf C++)	wahr	falsch
Mit <code>i_zahl = 7;</code> wird eine Integer-Variable mit dem Namen <code>i_zahl</code> angelegt und mit dem Wert 7 initialisiert.		
<code>int i_ganzzahl;</code> reserviert 4 Byte im Arbeitsspeicher.		
In <code>bool b_richtig = 1;</code> steht <code>bool b_richtig</code> für die Definition und der Ausdruck <code>= 1</code> für die Initialisierung der Variablen.		
Jede Variable muss bei der Definition einen Typ und einen Namen besitzen.		

3. Erstellen Sie mit ihrem Banknachbarn einen Merksatz, was eine Variablendefinition ist.

*Unter einer Variablendefinition versteht man, ...*

.....

.....

.....

.....

4. Welche Variable benötigt man, um den Messwert des Ultraschallsensors zu speichern? Bitte überlegen Sie sich auch eine sinnvolle Benennung für diese Variablen.

.....

.....

.....

.....

5. Beschreiben Sie kurz den Unterschied zwischen einer Variablendefinition und einer Initialisierung.

.....

.....

.....

.....

6. Berechnen sie die Werte der Integer-Variablen i\_zahl1 und i\_zahl2.

**Rechnen sie dabei jeweils mit den Werten der vorangegangenen Zeile weiter.**

Rechenoperatoren			Wert von i_zahl1 nach Ausführung	Wert von i_zahl2 nach Ausführung
Operator	Bezeichnung	Befehl		
=	Zuweisung	i_zahl1 = 7; i_zahl2 = 8;		
+	Addition	i_zahl1 = i_zahl2 + 3		
-	Subtraktion	i_zahl2 = i_zahl1 - 3		
*	Multiplikation	i_zahl2 = i_zahl2 * 3		
/	Division	i_zahl2 = i_zahl1 / 3		
%	Modulo (Rest einer Division)	i_zahl1 = i_zahl2 % 3		

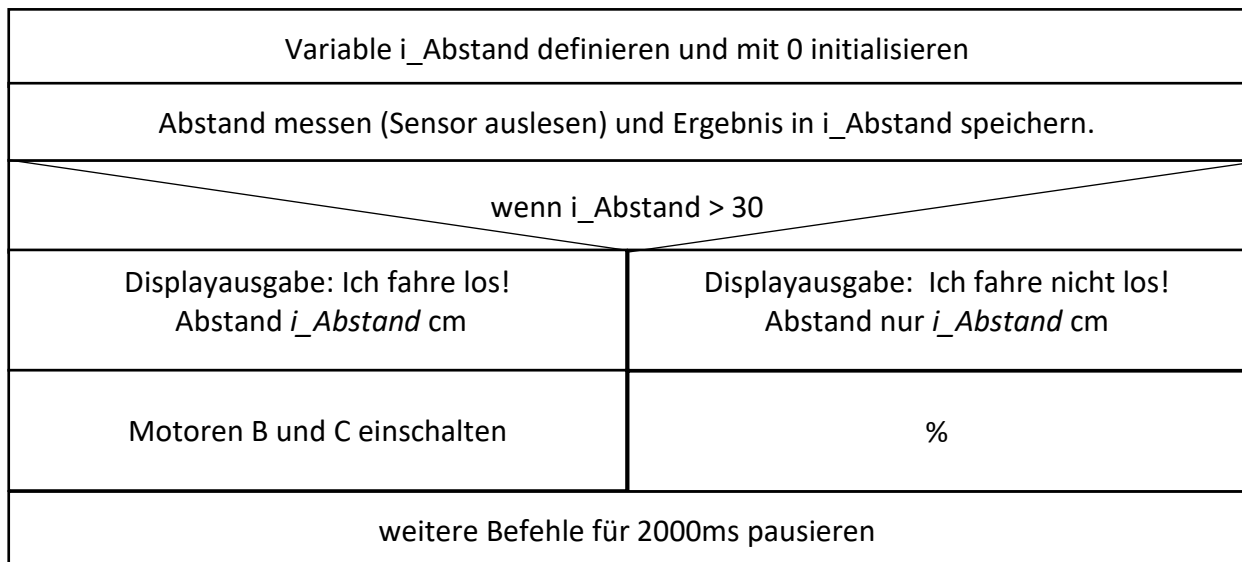
**Merke:** Bei einer **Zuweisung** steht die Variable, der ein Wert zugewiesen werden soll, immer auf der linken Seite vom „=" (Zuweisungsoperator). Die Zuweisung erfolgt also von rechts nach links.

**Beispiel:**  i\_Nummer = 5; //bedeutet, der Variable i\_Nummer wird der Wert 5 zugewiesen

## LS3.2.2 Programmierung Losfahrsperre mit Displayausgabe

### PARTNERARBEIT mit anschließender Code-Vorstellung:

Um den Wert der Messung korrekt ausgeben zu können, muss dieser vorher in einer Variablen gespeichert werden. Das folgende Struktogramm verdeutlicht das Programm.

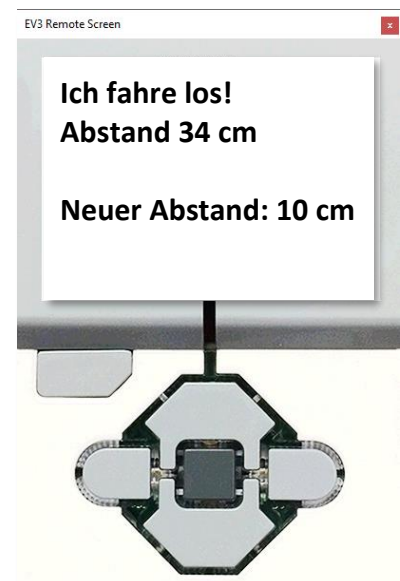


Schreiben Sie nun ein Programm (Dateiname: `LosfahrsperreDisplay.c`) welches das obige Struktogramm exakt umsetzt und testen Sie es ausführlich.

### Zusatzaufgabe für Schnelldenker:

Kommentieren Sie Programm sauber!

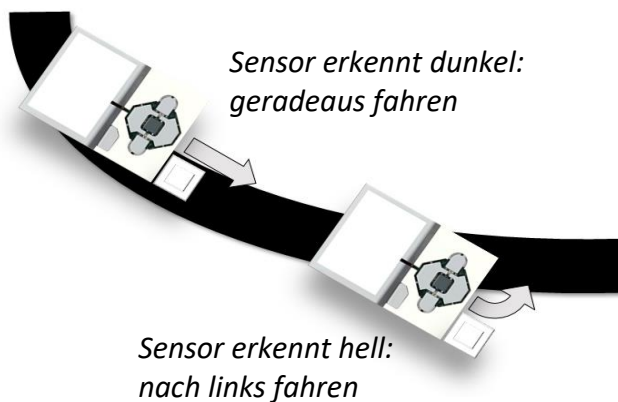
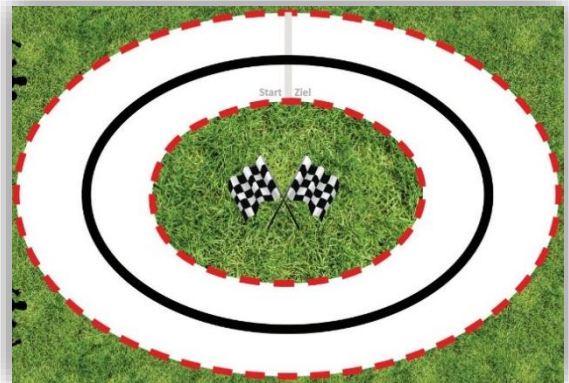
Implementieren Sie (nach den 2 Sekunden fahren bzw. nicht fahren) eine erneute Messung in eine zweite Variable (bspw. `i_AbstandEnde`), welche ebenfalls für zwei Sekunden in der nächsten Zeile des Displays ausgegeben werden soll.



### LS3.3 Kreisfahrer

Zunächst beauftragt Herr Müller Sie damit, einen Roboter zu programmieren, welcher eine runde Teststrecke in möglichst kurzer Zeit absolviert.

Der Roboter muss ständig überprüfen (Endlosschleife), welchen Wert der Helligkeitssensor hat. Führt er auf der Linie (Helligkeitswert gering), soll er geradeaus fahren. Ist er von der Linie abgekommen (Helligkeitswert hoch), soll er nach links wieder auf die schwarze Linie schwenken.



Hier muss sich der Roboter entscheiden: Wenn er dunkel erkennt, dann muss er geradeaus fahren. Registriert er allerdings hell, dann muss er links fahren. Die schwarze Linie muss entsprechend dick sein, damit sie der Roboter auch wahrnehmen kann.

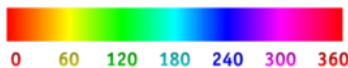
#### LS3.3.1 Farbsensor auslesen

Finden Sie heraus, wie Sie mit dem Farbsensor den Helligkeitswert erfassen können.

##### Farbsensor (Port 3)

**Farbwert erfassen:**

`getColorHue(S3);`



long 0 bis 360

**Farbnummer erfassen:**

`getColorName(S3);`

0	nichts	4	gelb
1	schwarz	5	rot
2	blau	6	weiß
3	grün	7	braun

int 0 bis 7

**Helligkeitswert erfassen:**

`getColorReflected(S3);`



short 0 bis 100

## LS3.3.2 Die Endlosschleife – Wiederhole immer

Bisher wurden Ihre Programme lediglich der Reihe nach abgearbeitet und u. U. verschiedenen Zweigen gefolgt. Da eine einmalige Prüfung (wie bei der Losfahrsperre) jedoch für den Kreisfahrer nicht ausreicht, sondern ständig geprüft und korrigiert werden muss ob sich der Roboter auf der Linie befindet, muss das Programm also endlos wiederholt werden. Mit Wiederholungen, sogenannten **Schleifen**, können Sie erreichen, dass Programmabschnitte mehrfach abgearbeitet werden können. Da der Roboter solange im Kreis fahren soll, bis er ausgeschaltet wird, verwenden wir in diesem Fall eine besondere Schleife - die Endlosschleife.

Sogenannte **kopfgesteuerte Wiederholungen** haben Ihren Kontrollpunkt, also ihre **Ausführungsbedingung**, noch vor dem ersten Schleifendurchlauf, demnach vor dem Eintritt in den **Schleifenrumpf**.

Vor Eintritt in die **Schleife** wird geprüft, ob die **Ausführungsbedingung** erfüllt ist - also der entsprechende Ausdruck einen Wert **ungleich 0** (also **true**) hat. Ist dies der Fall, so werden die darauf folgenden **Anweisungen** ausgeführt. Anschließend wird die **Bedingung** erneut geprüft. Sobald der Bedingungsausdruck dem **Wert 0** (also **false**) entspricht, wird das Programm mit der nächsten Anweisung **nach** der Schleife fortgesetzt.

Genau dieser Fall tritt bei einer Endlosschleife jedoch nie ein,  
da die Bedingung immer **true** sein wird.

Syntax in C basierten Sprachen (auch in RobotC)      Struktogramm nach DIN 66261

```
while (true)
{
    Anweisung 1;
    Anweisung 2;
}
```

} Schleifenrumpf  
bzw. Schleifenkörper

solange true, wiederhole

Anweisung 1

Anweisung 2

Beispiel für eine Endlosschleife

```
while (true)
{
    displayTextLine(1, "Schleife läuft");
}
```

solange true, wiederhole

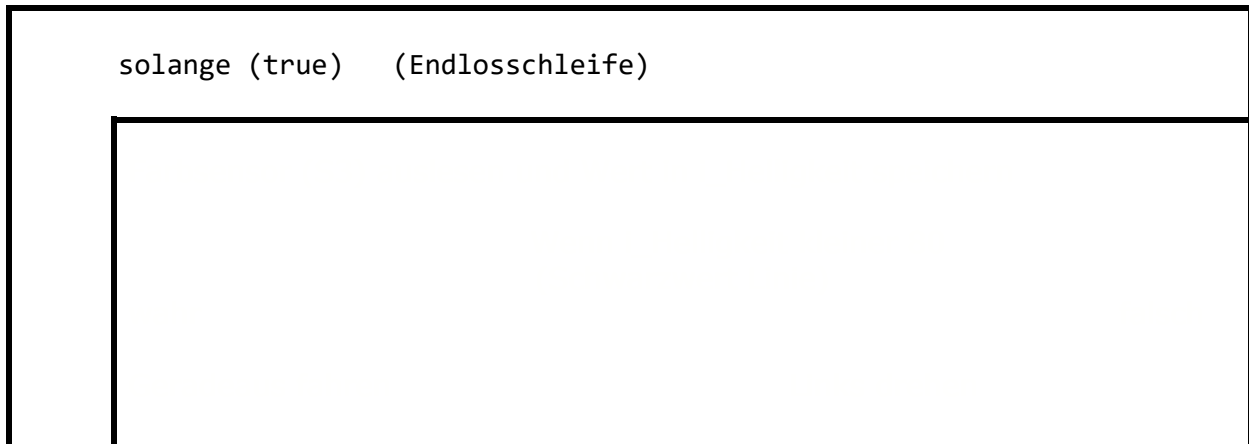
Ausgabe: Schleife läuft

Wenden Sie nun Ihr Wissen zu Entscheidungen und zu Endlosschleifen an, um ein Struktogramm für den Kreisfahrer zu erstellen (3.3.3) und diesen dann zu programmieren (3.3.4).

### LS3.3.3 Kreisfahrer planen (Struktogramm)

#### EINZELARBEIT mit anschließender Präsentation:

Wie sieht das Struktogramm des Kreisfahrers vor und innerhalb der Endlosschleife (while(1){...}) aus? (Verwenden Sie eine Variable i\_Helligkeit um den Sensorwert zu speichern)



### LS3.3.4 Kreisfahrer programmieren

#### PARTNERARBEIT mit anschließender Code-Vorstellung:

Schreiben Sie nun ein Programm (Dateiname: kreisfahren.c) welches Ihr geplantes Struktogramm umsetzt. Eine entsprechende Vorlage mit Sensorinitialisierung finden sie auf dem EV3-Laufwerk.

Wie lang braucht Ihr Roboter für eine Runde auf der Teststrecke? Messen Sie Ihre Bestzeit.



**< 7 Sekunden!**

#### Erweiterung A:

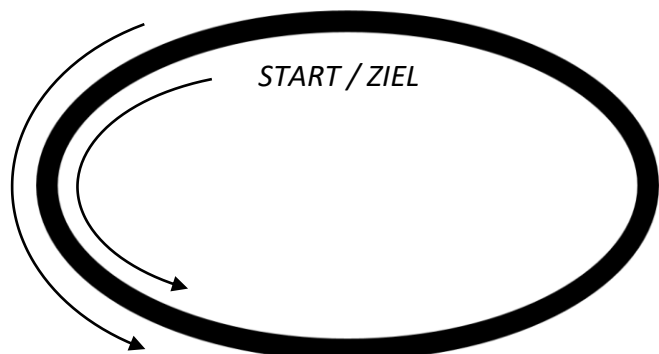
Herr Müller erwartet eine möglichst schnelle Version des Kreisfahrers. Optimieren Sie Ihr Programm so, dass der Roboter eine Runde in weniger als sieben Sekunden absolviert.

#### Erweiterung B:

Der Roboter soll nicht mehr an der Außenkante der Linie fahren, sondern sich an der Innenkante orientieren.

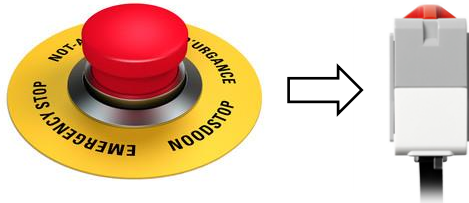
Setzen Sie eine entsprechende Lösung um.

Welchen Einfluss hat diese Änderung auf die Rundenzeit?





**ZUSATZAUFGABEN** für Fortgeschrittene -> zu lösen mit der **Befehlsliste**.



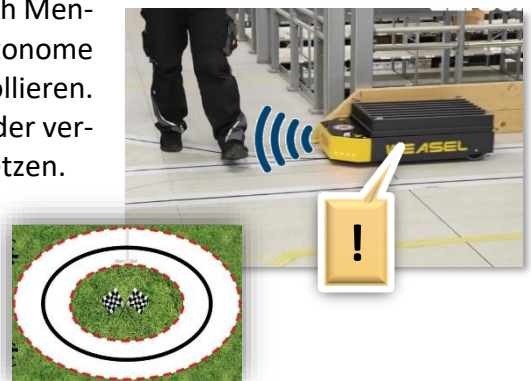
## Erweiterung C:

Für den Gefahrenfall haben Maschinen, Fahrzeuge und Anlagen einen Notausschalter. Ihr Transportroboter soll auch mit einem NOT-AUS versehen werden. Realisieren Sie eine Umsetzung mit Hilfe des Berührungssensors des EV3 Roboters.

## LS3.4 FINALE: Kreisfahrer mit Hinderniskontrolle

Damit es in Räumen, in denen sich sowohl Roboter als auch Menschen aufhalten, zu keiner Kollision kommt, müssen autonome Roboter ständig ihre Fahrbahn auf Hindernisse kontrollieren. Steht ein Objekt im Weg, soll er stehen bleiben bis es wieder verschwunden ist. Danach soll der Roboter seine Fahrt fortsetzen.

Anhand der Erfahrungen mit der „Losfahrsperre“ sollte das für Sie durchaus machbar sein.



### LS3.4.1 Kopfgesteuerte Schleife – Wiederhole, wenn...

Um die Fahrt des Roboters nicht ständig zu wiederholen, muss es eine Schleife geben, welche anhand des Ultraschallsensors gesteuert wird. Die Schleife darf also nur laufen, wenn eine bestimmte Bedingung erfüllt ist.

Wie bereits von der Endlosschleife bekannt, haben Kopfgesteuerte Wiederholungen dafür Ihren Kontrollpunkt, also ihre **Ausführungsbedingung**, noch vor dem ersten Schleifendurchlauf, also vor dem Eintritt in den **Schleifenrumpf**.

Syntax in C basierten Sprachen (auch in RobotC)

```
while (Bedingung)
{
    Anweisung 1;
    Anweisung 2;
}
```

} Schleifenrumpf  
bzw. Schleifenkörper

Struktogramm nach DIN 66261

**solange Bedingung erfüllt, wiederhole**

**Anweisung 1**

**Anweisung 2**

Die **Ausführungsbedingung** (kurz Bedingung) wird vor Eintritt in die **Schleife** geprüft. Ist sie wahr, werden die darauf folgenden **Anweisungen** ausgeführt. Anschließend wird die **Bedingung** erneut geprüft. Nur wenn der Bedingungsausdruck dem **Wert 0** (also **false**) entspricht, wird die Wiederholung durch die Schleife beendet.

#### Beispiel für kopfgesteuerte Schleifen

```
int i_zahl = 0;

while (i_zahl < 20)
{
    displayTextLine(1, "Wert:%d", i_zahl);
    sleep(100);
    i_zahl++;
}
```

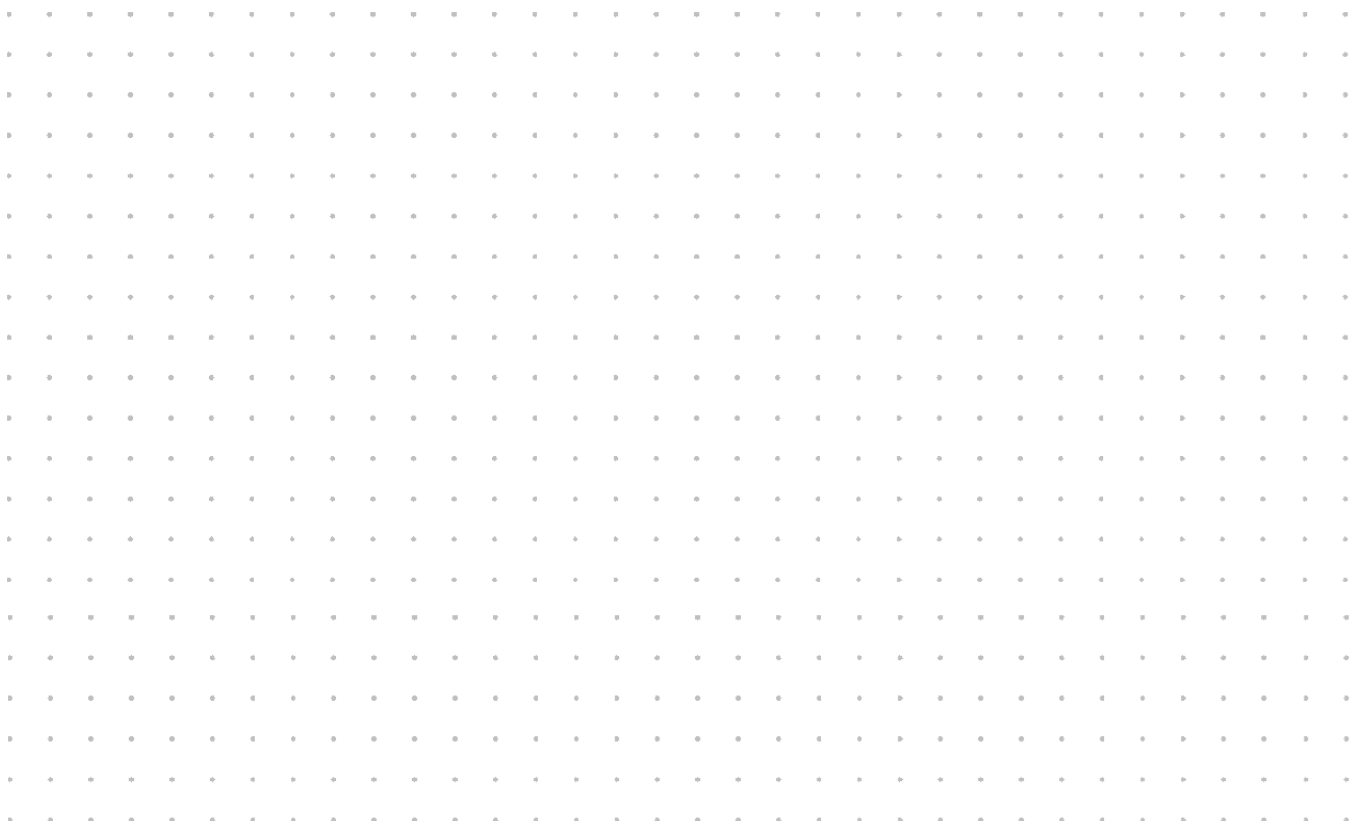
Variable definieren und initialisieren: i_zahl = 0
solange i_zahl kleiner 20 wiederhole
Ausgabe: i_zahl
weitere Befehle für 100ms pausieren
i_zahl um 1 erhöhen

### LS3.4.2 Kreisfahrer mit Hinderniskontrolle planen (Struktogramm)

#### EINZELARBEIT mit anschließender Präsentation:

Wie sieht das Struktogramm des Kreisfahrers mit Hinderniskontrolle aus?  
(Verwenden Sie eine Variable i\_Helligkeit um den Sensorwert zu speichern)

Tipp: Der Roboter soll weiterhin so lange im Kreis fahren, bis das Programm abgebrochen wird. Allerdings soll der Roboter stets stoppen, solange der Abstand kleiner oder gleich 30cm ist. Wird der Abstand nachher wieder größer, soll der Roboter einfach weiterfahren.



**LS3.4.2 Kreisfahrer mit Hinderniskontrolle programmieren****PARTNERARBEIT mit anschließender Code-Vorstellung:**

Schreiben Sie nun ein Programm (Dateiname: kreisfahrenHinderniss.c) welches Ihr geplantes Struktogramm umsetzt.

