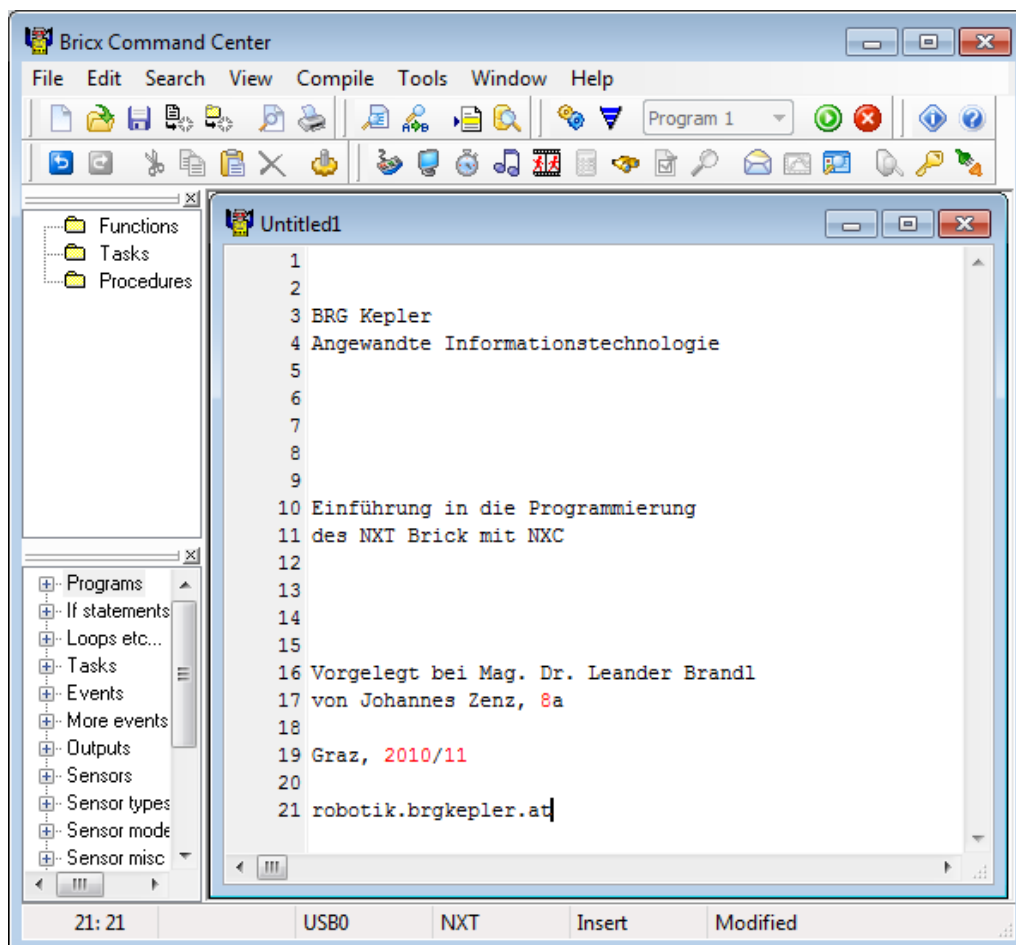


Einführung in die Programmierung des NXT Brick mit NXC



Johannes Zenz
2010/11

robotik.brgkepler.at

Inhaltsverzeichnis

1. Grundlagen & Vorbereitung..... 4

1.1 LEGO Mindstorms..... 4

1.2 Die Hardware 4

1.2.1 Der NXT Brick4

1.2.2 Die Motoren.....5

1.2.3 Die Sensoren5

1.3 Die Software 6

1.3.1 NXT-G mit der grafischen Oberfläche LEGO MINDSTORMS NXT6

1.3.2 Die Programmiersprache NXC6

1.3.3 Aktualisieren der Firmware7

1.3.4 Die Entwicklungsumgebung BricxCC7

1.3.5 Einführung in BricxCC8

2. Die Programmiersprache NXC..... 12

2.1 Das erste Programm programmieren und abspielen..12

2.1.1 Das Programm schreiben.....12

2.1.2 Das Programm kompilieren und downloaden.....13

2.1.3 Das Programm abspielen.....13

2.2 Die Motoren.....14

2.2.1 Kommentare14

2.2.2 Die Funktionen OnFwd() und OnRev()14

2.2.3 Die Funktion RotateMotor()15

2.2.4 Die Funktionen OnFwdReg() und OnRevReg()16

2.2.5 Die Funktionen OnFwdSync() und OnRevSync().....17

2.2.6 Bremsen18

2.2.6.1 Bremsen mit Off()18

2.2.6.2 Bremsen mit Float()18

2.3 Konstanten einführen.....19

2.4 Ausgabe von Text und Zahlen am LCD.....20

2.4.1 Ausgabe von Text.....20

2.4.2 Ausgabe von Zahlen.....20

2.5 Schleifen und Verzweigungen	21
2.5.1 Die repeat-Schleife	21
2.5.2 Die while-Schleife	22
2.5.3 Die if-Verzweigung.....	23
2.5.3.1 Vergleichsoperatoren für Bedingungen	23
2.5.3.2 Verknüpfungsoperatoren für Bedingungen.....	23
2.6 Die Sensoren	24
2.6.1 Der Tastsensor	25
2.6.2 Der Lichtsensor	26
2.6.3 Der Tonsensor.....	27
2.6.4 Der Ultraschallsensor.....	28
2.7 Variablen einführen.....	29
2.7.1 Deklaration von Variablen	29
2.7.2 Variablen einen Wert zuweisen.....	30
2.7.3 Rechnen mit Variablen	31
2.7.4 Vergleichen von Variablen.....	32
2.7.4.1 Zuweisendes und vergleichendes „Ist gleich“	32
2.8 Subroutinen und Makros erstellen und einsetzen	33
2.8.1 Subroutinen	33
2.8.2 Makros	34
<u>3. Anhang A – Befehlsübersicht</u>	<u>35</u>
3.1 Motoren	35
3.2 Timer	37
3.3 LC-Display	38
3.4 Schleifen & Verzweigungen	39
3.5 Sensoren	41
3.6 Variablen	42
3.7 Präprozessoren.....	43
3.8 Subroutinen	44
<u>4. Anhang B – Aufgabensammlung.....</u>	<u>45</u>

1. Grundlagen & Vorbereitung

1.1 LEGO Mindstorms

Seit 2006 besteht mit LEGO Mindstorms eine neue Roboter-Produktreihe von LEGO. Das Besondere an dieser Reihe ist der intelligente, programmierbare Multi-Sensor-Baustein, der



NXT Brick (engl. „Stein“).

Mit dem NXT Brick wird es ermöglicht LEGO Roboter zu bauen, die mithilfe von eigenen Elektromotoren und Sensoren sowohl agieren, als auch reagieren können. Diese Eigenschaften machen den NXT Brick perfekt für Schulen. So werden mittlerweile Tausende Roboter programmiert, um an nationalen sowie internationalen Wettbewerben, wie der FIRST Lego League oder dem RoboCup Junior, teilzunehmen und dort die Fähigkeiten des Roboters und der (Nachwuchs-)Programmierer unter Beweis zu stellen.

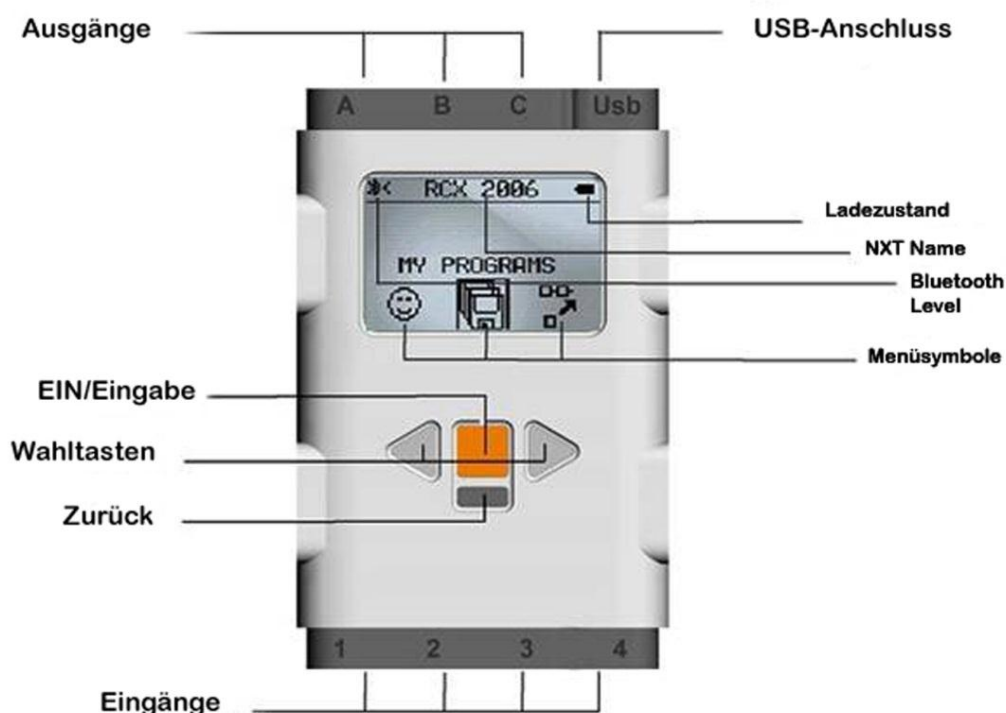
1.2 Die Hardware

1.2.1 Der NXT Brick

Der NXT Brick ist das „Herz“ bzw. das „Hirn“ des Roboters. Hier kommen die Signale von außen herein und werden die Befehle erteilt. Er besitzt 4 Eingänge, um die Werte der Sensoren zu erhalten, und 3 Ausgänge, um die Motoren anzusteuern. Um Programme zu übertragen verfügt er über einen USB-Anschluss und Bluetooth.

Das LCD (engl. *Liquid Crystal Display* = „Flüssigkristallbildschirm“) besitzt eine Auflösung von 100 x 64 Pixel. Mit den vier Buttons navigiert man das angezeigte Menü.

(Oranger Button = OK/Bestätigen; Hellgrau = links/rechts; Dunkelgrau = Abbruch/Zurück)



1.2.2 Die Motoren

Für den NXT wurden neue Servomotoren entwickelt. Sie können je nach Belieben nach Zeit, Geschwindigkeit und jetzt auch in Grad angesteuert werden. Die Motoren können gradweise (360° für eine ganze Umdrehung) nach vorne oder nach hinten gesteuert werden. An den Ausgängen A, B und C können je ein Servomotor angeschlossen werden.



1.2.3 Die Sensoren

Die Sensoren sind die „Sinnesorgane“ des Roboters. Sie ermöglichen ihm Interaktionen mit seiner Umwelt. Im LEGO Mindstorms NXT Set sind folgende Standard-Sensoren enthalten: Ultraschallsensor, Tastsensor, Tonsensor und der Lichtsensor (v. l.).



Mittlerweile gibt es auch von anderen Firmen und Herstellern LEGO kompatible Sensoren für den NXT. So gibt es z.B. auch Kameras, Fernsteuerungen, Infrarotsensoren, Kompass usw. für das Interface von LEGO Mindstorms zu kaufen. An den Eingängen 1, 2, 3 und 4 kann je ein Sensor angeschlossen werden.

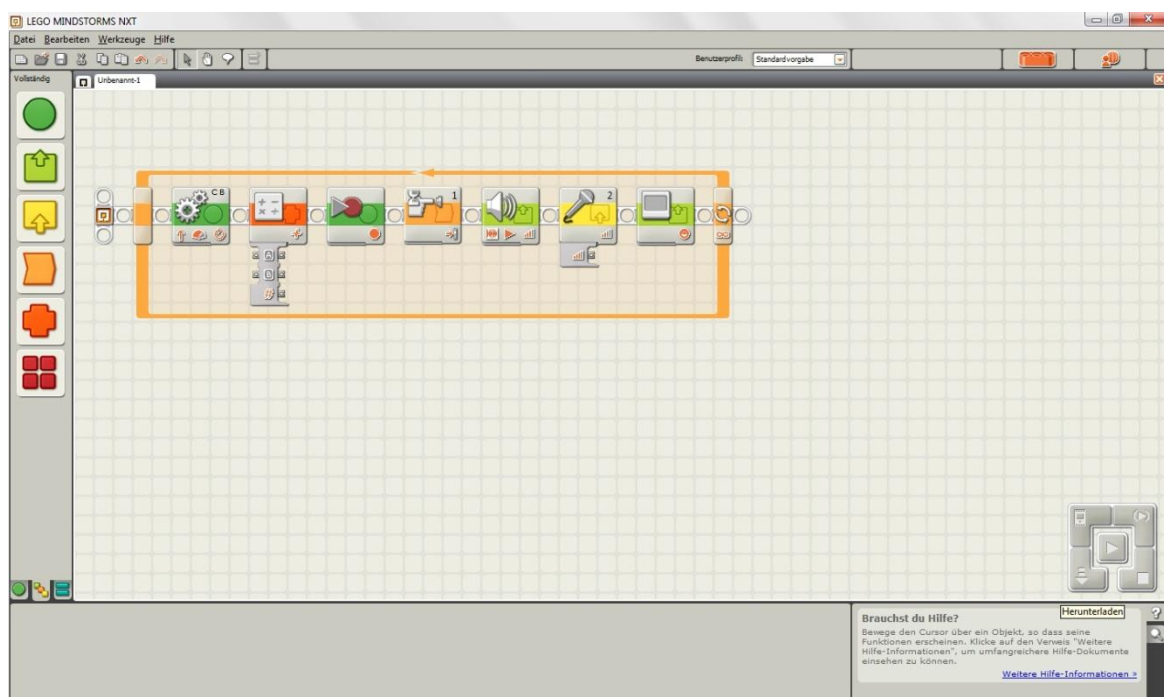
Ein weitaus genauerer Blick auf die Sensoren und vor allem ihre Funktionsweise erfolgt später (siehe Kap. „2.6 Die Sensoren“).

1.3 Die Software

1.3.1 NXT-G mit der grafischen Oberfläche LEGO MINDSTORMS NXT

Die mitgelieferte Software ermöglicht das Erstellen von Programmen für den NXT mit einer grafischen Oberfläche. Diese Sprache wird NXT-G (engl. **G**rafic = „grafisch“) genannt. Die Programme werden in einer Art Baukastensystem Aktion für Aktion aneinandergereiht. Für die sehr übersichtliche und einfach zu bedienende Software werden keinerlei Programmierkenntnisse vorausgesetzt und bietet dabei zwei verschiedene Programmierbereiche, für Einsteiger und Fortgeschrittene.

Leider kann man mit der grafischen Oberfläche jedoch nicht das gesamte Potential des NXT ausschöpfen, da verschiedene Funktionen nicht vorhanden sind. Ein weiterer Nachteil ist, dass ein von der Funktion her identes NXT-G-Programm gegenüber einem NXC-Programm ein Vielfaches des zugegebenermaßen begrenzten Speichers des NXT Brick beansprucht.



1.3.2 Die Programmiersprache NXC

Da man mit der grafischen Programmieroberfläche NXT-G nicht den vollen Leistungsumfang des NXT Brick nutzen kann, gibt es mehrere Möglichkeiten, bzw. Programmiersprachen, mit denen eben dies möglich ist. Die Programmiersprache NXC „**Not eXactly C**“ ist in Anlehnung an die sehr bekannte Programmiersprache C unter John C. Hansen entstanden. Ihm ist es zu verdanken, dass auch schon frühere LEGO-Robotergenerationen wie der RCX besser programmierbar geworden sind. Wie bereits oben erwähnt, belegen zwei von der Funktion her äquivalente Programme (je eines in NXT-G und NXC programmiert) verschieden viel des Speichers des NXT Brick. Hinzu kommt noch, dass gilt: je kleiner das Programm, desto schneller. Ein NXC-Programm wird vom NXT Brick bis zu **zehn Mal schneller** verarbeitet, als ein NXT-G-Programm. Somit hat die NXC-Sprache mehrere entscheidende Vorteile gegenüber NXT-G.

Das vorliegende Skriptum soll das selbstständige Erlernen der NXT-Programmierung mit NXC ermöglichen. Es wird vor allem darauf geachtet, auch ohne jegliche Vorkenntnisse den NXT Brick erfolgreich programmieren zu können.

1.3.3 Aktualisieren der Firmware

Damit der NXT Brick immer auf dem neuesten Stand bleibt und alle derzeit verfügbaren (Programmier-)Ressourcen ausnutzen kann (wie z.B. das Programmieren mit Gleitkommazahlen), muss die Firmware stets auf dem neuesten Stand sein. Die Firmware ist quasi das Betriebssystem für den NXT Brick.

Die Firmware wird in 4 einfachen Schritten upgedatet:

(Achtung! Für das Firmware-Update muss LEGO MINDSTORMS NXT installiert sein!)

- 1.) Lade die neueste (höchste) Version der Firmware herunter:
<http://mindstorms.lego.com/en-us/support/files/default.aspx#Firmware>
- 2.) Entpacke die heruntergeladene Datei nach:
„...\\Program Files\\LEGO Software\\LEGO MINDSTORMS NXT\\engine\\Firmware“
- 3.) Starte LEGO MINDSTORMS NXT und gehe im Menü auf „Tools“ bzw. „Werkzeuge“ und wähle dort „Update NXT Firmware...“ bzw. „NXT-Firmware aktualisieren...“ aus
- 4.) Wähle die neueste Firmware aus und drücke „Download“ bzw. „Herunterladen“
(Der NXT Brick muss dafür angeschlossen und eingeschalten sein)
- 5.) ...Fertig! Der NXT Brick ist auf dem neuesten Stand!

Nun kann kontrolliert werden, ob alles funktioniert hat, indem man im NXT Menü „Einstellungen“ bzw. „Settings“ und dann „NXT Version“ auswählt: in der obersten wird die aktuelle FW (Firmware Version) angezeigt.



1.3.4 Die Entwicklungsumgebung BricxCC

Die freie Software BricxCC bietet sich besonders wegen seiner einfachen Oberfläche und umfassenden Funktionalität an.

Das **Bricx Command Center** (engl. *„Kommandozentrum der (Lego-)Steine“*) ist eine IDE für alle *„programmierbaren Legosteine“*. Eine IDE (engl. **I**ntegrated **D**evelopment **E**nvironment) ist eine *„integrierte Entwicklungsumgebung“*, also ein Programm, mit dem man Software entwickeln kann, in diesem Fall für den NXT Brick.

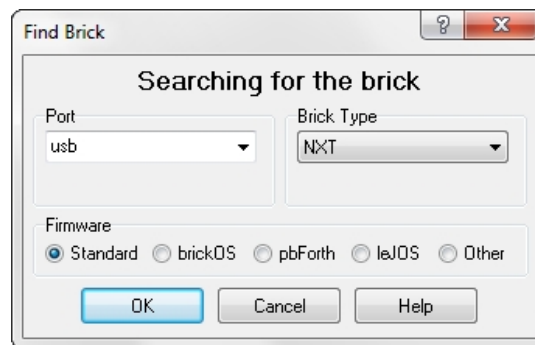
Die aktuellste Version von BricxCC steht unter folgender Adresse zum freien Download bereit: <http://sourceforge.net/projects/bricxcc/files/bricxcc/>

Nach dem Herunterladen, das Programm installieren und starten. Für weitere Schritte siehe Kap. „1.3.5 Einführung in BricxCC“.

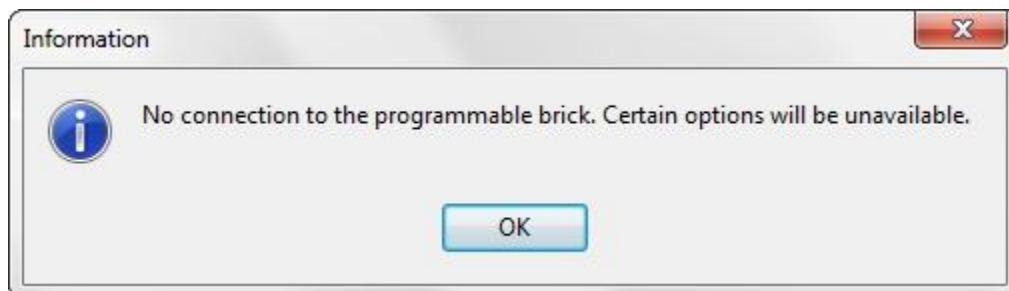
1.3.5 Einführung in BricxCC

BricxCC ist derzeit nur auf Englisch erhältlich, dies stellt jedoch in der Bedienung keine Probleme dar.

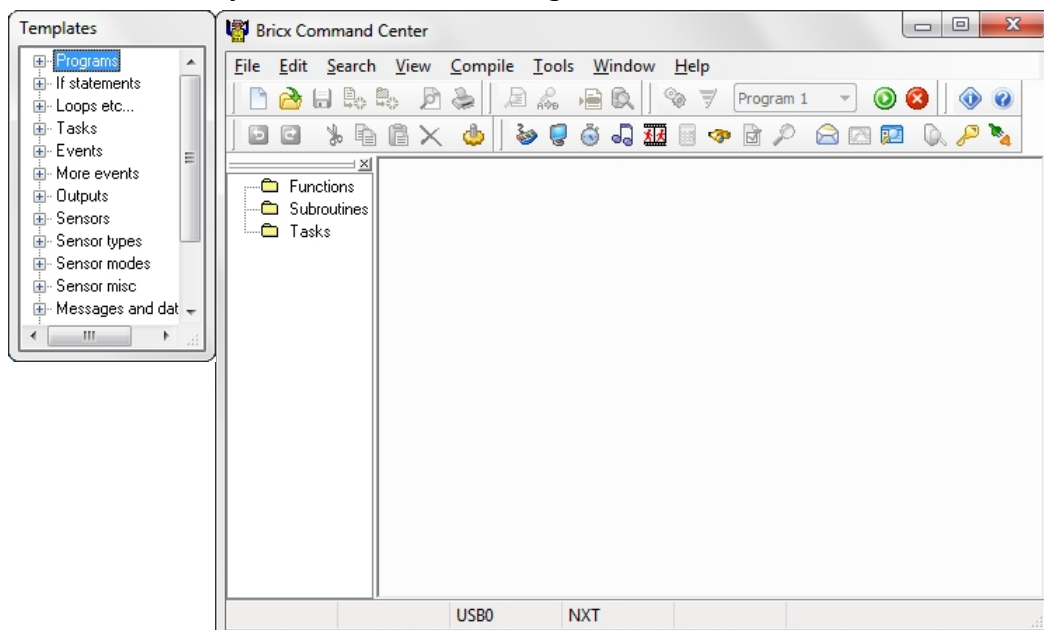
Beim Starten fragt BricxCC nach, für welchen LEGO Brick die Software erstellt werden soll und wie der LEGO Brick an den Computer angeschlossen ist. In diesem Fall wurde der Port „usb“ und der Brick Type „NXT“ gewählt. Bei Firmware bleibt „Standard“ ausgewählt.



Falls der Brick ausgeschaltet oder nicht angesteckt ist, erscheint folgende Fehlermeldung: „Keine Verbindung zum programmierbaren Brick. Bestimmte Optionen werden nicht verfügbar sein.“

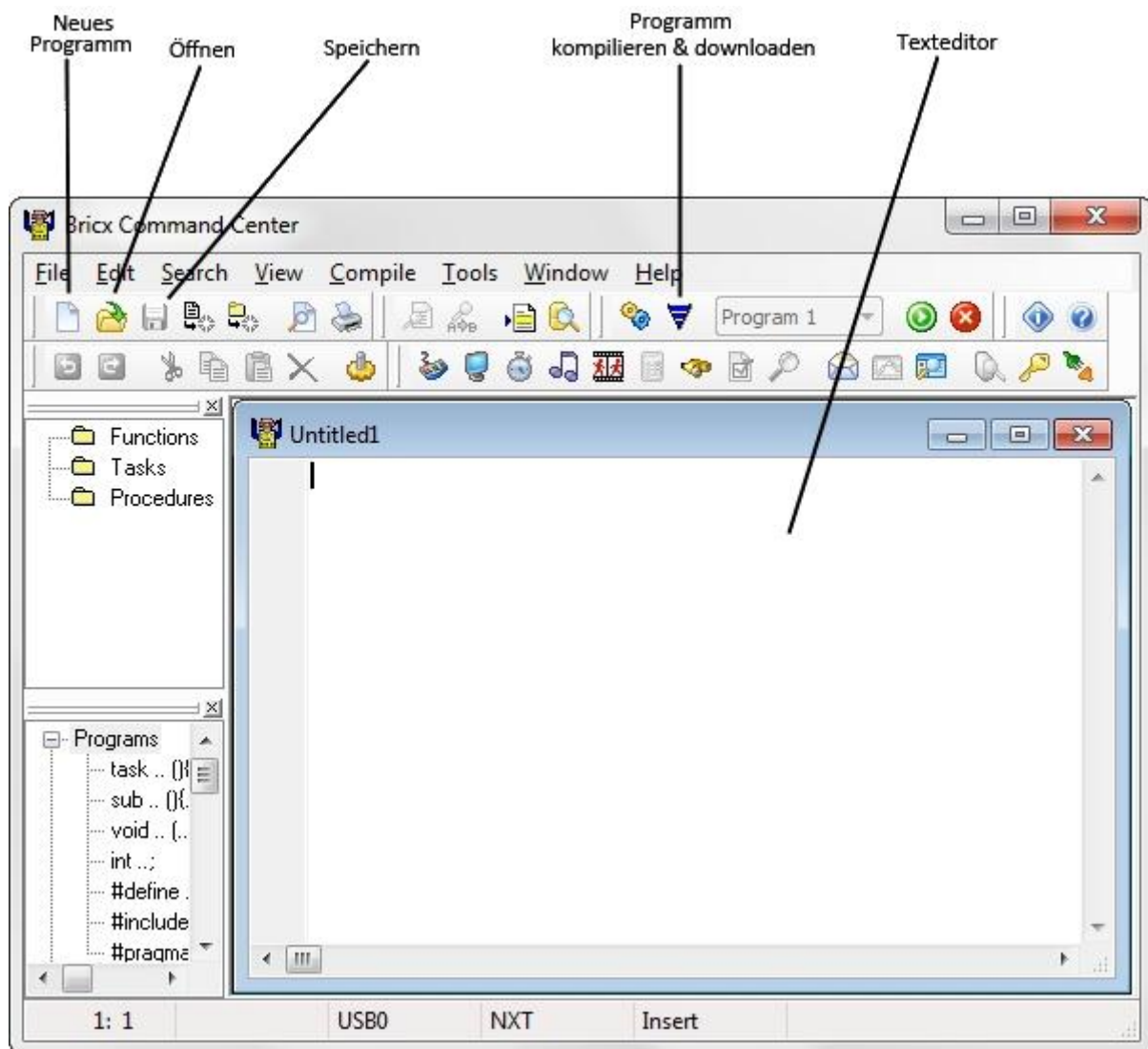


Danach startet BricxCC jedoch trotzdem und zeigt seine Oberfläche.



Das Hauptaugenmerk liegt auf dem größeren Fenster der Oberfläche – dem Quelltext-Editor. Das kleinere Fenster (links) beinhaltet lediglich sog. Templates (engl. „Vorlagen“). Es ermöglicht uns, z.B. die Grundstrukturen verschiedener Funktionen einzufügen, jedoch ist es nicht weiter interessant und kann vernachlässigt werden.

In der folgenden Grafik sind die wichtigsten Icons markiert:

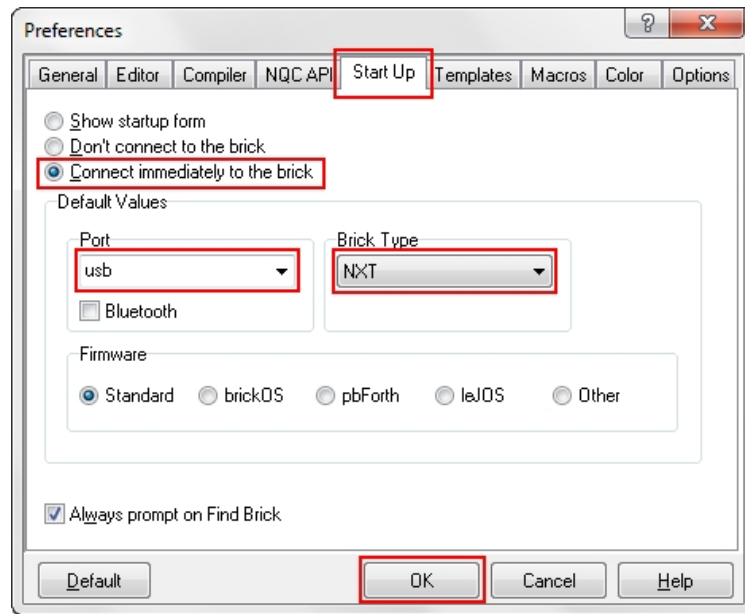


Um das Bricx Command Center noch funktioneller zu machen, sollten ein paar weitere kleine Änderungen vorgenommen werden. Im Menüpunkt **Edit** → **Preferences** (engl. „Vorzüge“) kann man die Einstellungen vornehmen.

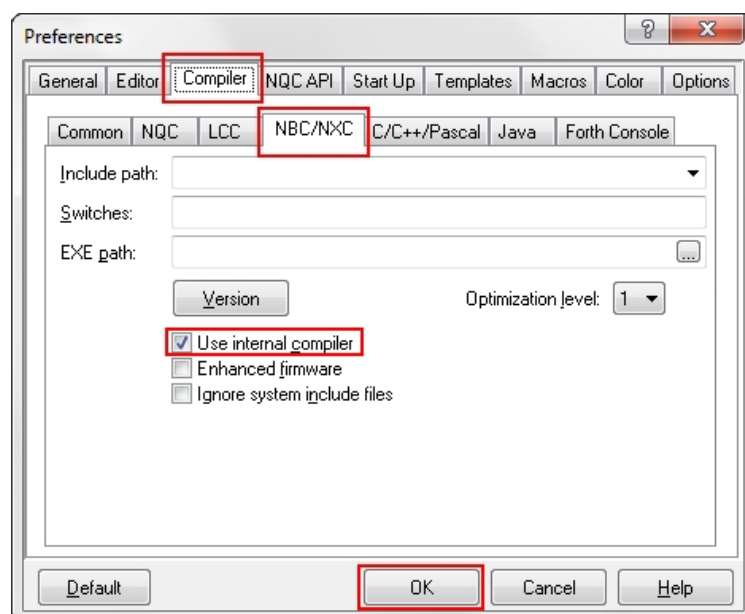
Zum Beispiel im Preferences-Reiter **Start Up**: in diesem Reiter setzt man Standardwerte beim Startdialog, in dem nach dem verwendeten NXT Brick und der Verbindung gefragt wird. Somit wird der Startdialog deaktiviert und die Verbindung findet automatisch statt.

Dazu muss man die Einstellungen **Connect immediately to the brick** (engl. „Verbinde sofort zum Brick“) auswählen und dann die Standardverbindung setzen – in diesem Fall mit dem Port **usb** und dem Brick Type **NXT**. Mit **OK** bestätigen.

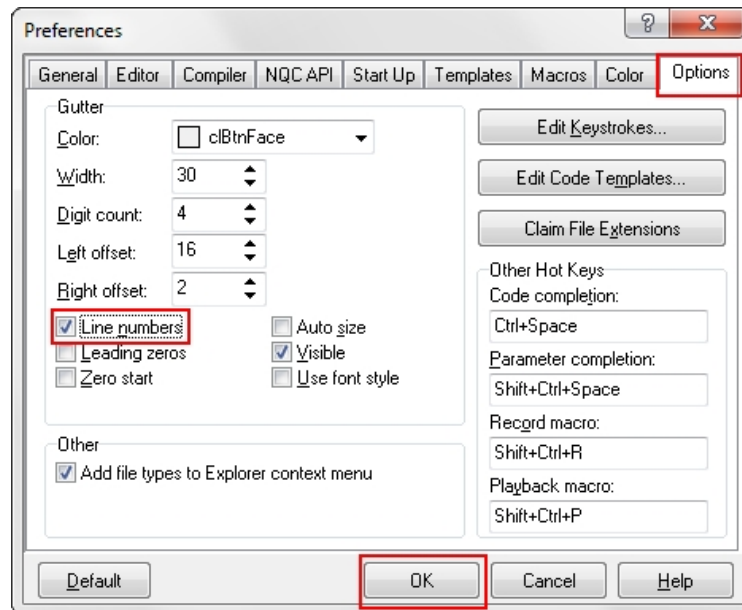
Von nun an verbindet sich der PC automatisch mit dem NXT (sofern dieser per USB angeschlossen und eingeschaltet ist).



Eine weitere wichtige Einstellung sollte im Reiter **Compiler** und Unterreiter **NBC/NXC** vorgenommen werden. Hier kann der interne Compiler – ein Compiler übersetzt den geschriebenen NXC-Code in eine für den NXT Brick verständliche Sprache – aktiviert werden. Dieses Häkchen bringt wieder Zeit und Speicherplatz. Dafür muss **Use internal compiler** angehakt und wieder mit **OK** bestätigt werden.



Die letzte wirklich praktische Einstellung ist die Zeilennummerierung. Dazu wieder auf **Edit** → **Preferences** und dort den Reiter **Options** (engl. „Optionen“) anklicken. Danach die Option **Line numbers** (engl. „nummeriere Zeilen“) anhaken und mit **OK** bestätigen.



2. Die Programmiersprache NXC

2.1 Das erste Programm programmieren und abspielen

2.1.1 Das Programm schreiben

Jedes Programm besteht aus sogenannten Tasks (engl. „Aufgaben“). Ein NXC-Programm benötigt zumindest einen main-Task (engl. „Haupt-Aufgabe“). Alle Anweisungen eines Tasks werden mit geschwungenen Klammern {...} zusammengefasst. Genau wie in der Programmiersprache C wird auch in NXC **jede** Anweisung mit einem Strichpunkt ; beendet.

Für das einfache erste Programm werden drei Funktionen verwendet: **OnFwd()**, **Wait()** und **Off()**. Jeder Funktion werden in Klammern sogenannte *Parameter* übergeben.

Die Funktion **OnFwd(AUSGANG_MOTOR, LEISTUNG)** (engl. Fwd = „forward“ = „nach vorne“) besitzt zwei verschiedene Parameter – **AUSGANG_MOTOR** und **LEISTUNG** – und lässt den Motor nach vorne drehen.

Der Parameter **AUSGANG_MOTOR** kann mehrere Werte annehmen. Als Ausgang kann entweder ein einzelner Motor (**OUT_A**, **OUT_B** oder **OUT_C**) oder es können gleichzeitig auch mehrere Motoren (**OUT_AB**, **OUT_AC**, **OUT_BC** oder **OUT_ABC**) angesteuert werden.

Der Parameter **LEISTUNG**, der die Drehgeschwindigkeit des Motors angibt, ist eine ganze Zahl aus dem Bereich zwischen 0 und 100.

Die Funktion **Wait(MILLISEKUNDEN)** (engl. „warten“) besitzt den Parameter **MILLISEKUNDEN**. Der Funktion **Wait()** wird die Zeit übergeben, die das Programm den aktuellen Zustand behalten soll – es wartet. Der Parameter **MILLISEKUNDEN** wird in ganzen Zahlen angegeben (1000 Millisekunden = 1 Sekunde).

Die Funktion **Off(AUSGANG_MOTOR)** (engl. „aus“) bremst die Motoren abrupt ab. Welche weitere Möglichkeiten es gibt, einen Motor auszuschalten, wird im Kapitel „2.2.6 Bremsen“ behandelt.

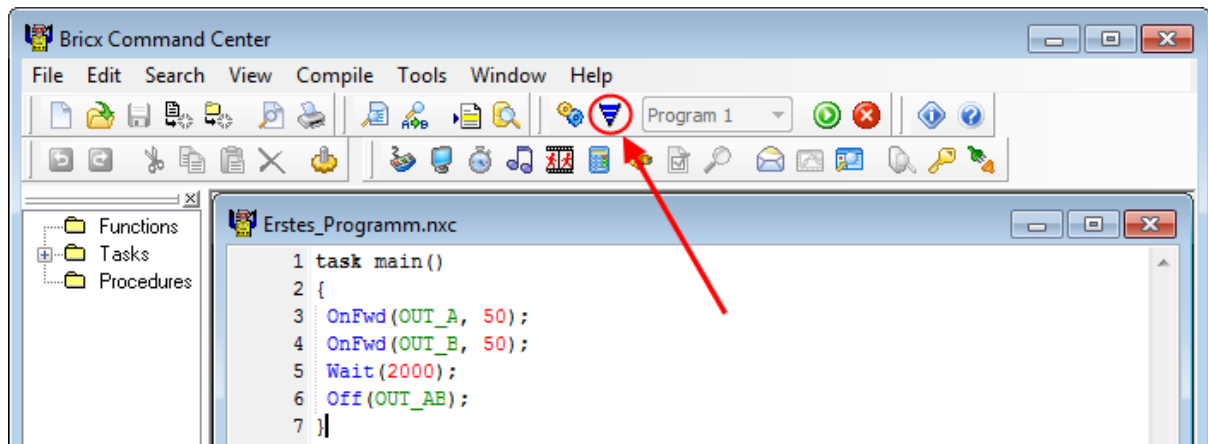
Für das erste Programm wird ein Roboter benötigt, der zwei angeschlossene Motoren besitzt (Ausgang A und Ausgang B). Der Code würde demnach so aussehen:

```
1 task main()
2 {
3   OnFwd(OUT_A, 50);
4   OnFwd(OUT_B, 50);
5   Wait(2000);
6   Off(OUT_AB);
7 }
```

Es ist üblich, Anweisungen, die mit geschwungenen Klammern zusammengefasst sind, einzurücken!

2.1.2 Das Programm kompilieren und downloaden

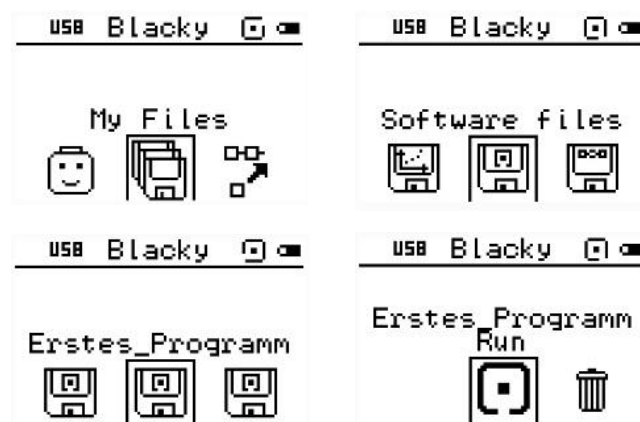
Der Code ist nun fertiggestellt, und das Programm muss noch auf den NXT übertragen werden. Dies geschieht mit einem Klick auf das Pfeilsymbol:



Der integrierte Compilers (engl. „Übersetzer“) übersetzt nun den Quellcode in ein für den NXT verständliches Programm, dass dann am NXT Brick ausgeführt werden kann.

2.1.3 Das Programm abspielen

Sobald der NXT piepst, ist das Programm fertig übertragen. Jetzt kann man über das Menü das Programm starten:



Im main-Task passiert folgendes:

- 3. Zeile Motor A fährt mit 50% Leistung nach vorne
- 4. Zeile Motor B fährt mit 50% Leistung nach vorne
- 5. Zeile Das Programm „wartet“ für zwei Sekunden im gleichen Zustand, die Motoren fahren also zwei Sekunden vorwärts
- 6. Zeile Die Motoren A und B werden gebremst

Sobald das Programm fertig ist, liefert es am LCD **DONE** (engl. „gemacht“) zurück.



Aufgabe 1

2.2 Die Motoren

Es gibt drei mögliche Outputs (engl. „Ausgabe“) beim NXT Brick: Ton (Lautsprecher am Brick), die Anzeige (LC-Display) sowie die Motorengänge A, B und C. Über die Motorengänge werden dann die einzelnen Motoren angesteuert.

Für kaum etwas gibt es beim NXT so viele verschiedene Funktionen wie für die Servomotoren.

2.2.1 Kommentare

Um im Quellcode den Überblick zu behalten bieten sich Kommentare besonders an. Kommentare sind kleine Notizen im Quellcode, die jedoch nicht das Programm beeinflussen. Ein Kommentar kann nur eine Zeile lang sein, er kann sich aber auch über mehrere Zeilen hinwegstrecken. Man kann ein Kommentar je nach Belieben hinzufügen und entfernen, ohne den Quellcode in irgendeiner Weise zu beeinflussen.

Hier wurde das erste Programm mit Kommentaren versehen:

```
1 task main()
2 {
3   // Ein Kommentar über eine Zeile
4   OnFwd(OUT_A, 50);
5   OnFwd(OUT_B, 50);
6   /* Dieses Kommentar geht
7   über mehrere Zeilen
8   und es verändert den Code nicht! */
9   Wait(2000);
10  Off(OUT_AB);
11 }
```

Es gibt also zwei Arten von Kommentaren: einzeilige und mehrzeilige. Für einzeilige Kommentare müssen zwei Schrägstriche vor den Kommentar gesetzt werden (siehe Zeile 3). Mehrzeilige Kommentare fangen mit einem Schrägstrich und einem Stern an und enden mit einem Stern und Schrägstrich.

Einzeilig:

// Kommentar

// Kommentar

Mehrzeilig:

/* Kommentar

Kommentar

Kommentar */

2.2.2 Die Funktionen OnFwd() und OnRev()

Die Funktion **OnFwd()** ist bereits im Kapitel „2.2.1 Das Programm schreiben“ genau erklärt worden. Die Funktion **OnRev(AUSGANG_MOTOR, LEISTUNG)** (engl. „reverse“ = „zurück“) entspricht **OnFwd()** mit dem einzigen Unterschied, dass sich die Motoren bei **OnRev()** rückwärts drehen.

Aufgabe 2.1

2.2.3 Die Funktion RotateMotor()

Die Funktionen **OnFwd()** oder **OnRev()** haben einen kleinen Nachteil. Sie sind abhängig von der Genauigkeit des Motors und somit vor Störfaktoren wie z.B. niedrigem Akkulevel nicht geschützt. Mithilfe der Funktion **RotateMotor(AUSGANG_MOTOR, LEISTUNG, WINKEL)** (engl. „rotieren“) lässt sich der Motor aber auf das Grad genau steuern.

Der Roboter führt nun alle Befehle nacheinander aus:

```
1 task main()
2 {
3   RotateMotor(OUT_A, 75, 180);
4   RotateMotor(OUT_B, 50, 180);
5   RotateMotor(OUT_A, 100, -400);
6   RotateMotor(OUT_B, 100, -400);
7   RotateMotor(OUT_AB, 50, 360);
8 }
```

Der Parameter **AUSGANG_MOTOR** entspricht dem bei **OnFwd()** oder **OnRev()**. Somit ist es wieder möglich die Motoren einzeln oder gleichzeitig anzusteuern.

Der Parameter **LEISTUNG** ist ident mit dem Parameter **LEISTUNG** bei **OnFwd()** oder **OnRev()**. Die Leistungspanne von 0% bis 100% kann angegeben werden.

Der einzige neue Parameter **WINKEL** beschreibt die Gradanzahl, die sich der jeweilige Motor drehen soll. Ist die Angabe positiv, dreht sich der Motor solange nach vorne bis er den angegebenen Drehwinkel erreicht hat. Ist die Angabe negativ, dreht sich der Motor rückwärts. 360° entsprechen einer ganzen Umdrehung.

Im main-Task passiert folgendes:

- 3. Zeile Motor A dreht sich mit 75% Leistung 180° nach vorne
- 4. Zeile Motor B dreht sich mit 50% Leistung 180° nach vorne – der Roboter fährt leicht nach links
- 5. Zeile Motor A dreht sich mit voller Leistung 400° nach hinten
- 6. Zeile Motor B dreht sich mit voller Leistung 400° nach hinten – der Roboter fährt gerade zurück
- 7. Zeile Motor A und B drehen sich mit 50% Leistung genau eine Umdrehung (bzw. 360°) nach vorne – der Roboter fährt exakt gerade aus

Aufgabe 2.2

2.2.4 Die Funktionen OnFwdReg() und OnRevReg()

Wenn man länger mit einem NXT Roboter arbeitet, wird man früher oder später feststellen, dass er bei noch so genauer Programmierung ab und zu vom Kurs abkommt. Geht man davon aus, dass kein z.B. asymmetrischer Konstruktionsfehler gemacht wurde, liegt es meist daran, dass der Grip (engl. „*Haftung, Reibung*“) der Reifen oder andere unerwünschte Unreinheiten am Boden den Weg des Roboters beeinflussen.

Um zu garantieren, dass der Roboter bei noch so schlechter Unterlage geradeaus fährt, gibt es die Funktionen **OnFwdReg**(*AUSGANG_MOTOR*, *LEISTUNG*, *REGMODE*) und **OnRevReg**(*AUSGANG_MOTOR*, *LEISTUNG*, *REGMODE*) (engl. Reg = „*Regulation*“ = „*Regulation*“). Diese beinhalten die bereits bekannten Parameter *AUSGANG_MOTOR* (*Wichtig: Der Parameter AUSGANG_MOTOR muss diesmal mehrere Motoren angeben!*) und *LEISTUNG*, sowie den neuen Parameter *REGMODE* (engl. „*Regulierungsmodus*“).

```
1 task main()
2 {
3   OnFwdReg(OUT_AB, 50, OUT_REGMODE_SYNC);
4   Wait(8000);
5   Off(OUT_AB);
6 }
```

REGMODE wird auf den Modus *OUT_REGMODE_SYNC* gesetzt. Der Modus *OUT_REGMODE_SYNC* bezieht sich auf die Rotationen der Motoren. Hält man einen der beiden (laufenden) Motoren an, so stoppt auch der zweite, um die beiden Motoren synchron laufen zu lassen.

Analog dazu funktioniert **OnRevReg()** – der einzige Unterschied ist, dass der Roboter zurück fährt.

Im main-Task passiert folgendes:

- 3. Zeile Motor A und B fahren mit 50% Leistung nach vorne und korrigieren sich dabei gegenseitig
- 4. Zeile Das Programm „wartet“ für acht Sekunden, die Motoren fahren also acht Sekunden vorwärts
- 5. Zeile Die Motoren A und B werden gebremst

Aufgabe 2.3

2.2.5 Die Funktionen OnFwdSync() und OnRevSync()

Da der Roboter aber nicht nur immer geradeaus fahren soll, gibt es die Funktionen **OnFwdSync()** (**AUSGANG_MOTOR**, **LEISTUNG**, **RADIUS**) und **OnRevSync()** (**AUSGANG_MOTOR**, **LEISTUNG**, **RADIUS**). Diese beiden sehr praktischen Funktionen erleichtern das Fahren von Kurven. Bis jetzt war es nur möglich mit den Standardfunktionen **OnFwd()** und **OnRev()** sich umständlich in eine Kurve hineinzulenken, das ändert sich nun.

Die Funktionen **OnFwdSync()** und **OnRevSync()** besitzen drei verschiedene Parameter, wieder die bereits bekannten **AUSGANG_MOTOR** (*Wichtig: Der Parameter **AUSGANG_MOTOR** muss auch hier mehrere Motoren angeben!*) und **LEISTUNG** und zusätzlich den Parameter **RADIUS**. **OnFwdSync()** und **OnRevSync()** unterstützen auch die automatische Geschwindigkeitskorrektur wie **OnFwdReg()** und **OnRevReg()**.

```
1 task main()
2 {
3   OnFwdSync(OUT_AB, 50, 10);
4   Wait(8000);
5   Off(OUT_AB);
6 }
```

Der Parameter **RADIUS** beschreibt die Krümmung der Kurve und kann Werte zwischen **-127** und **127** annehmen. Beim obigen Beispiel fährt der Roboter eine Kurve nach rechts. Je nach Bauweise des Roboters ergeben die verschiedenen Werte verschieden große Radien. Setzt man **RADIUS** auf 0, fährt der Roboter geradeaus. Bei **127** bzw. **-127** drehen sich beide Räder in die entgegengesetzte Richtung – der Roboter dreht sich auf der Stelle.

Analog dazu funktioniert **OnRevSync()**, der einzige Unterschied liegt darin, dass der Roboter rückwärts fährt.

Im main-Task passiert folgendes:

- 3. Zeile Motor A und B fahren mit 50% Leistung nach vorne und leicht nach rechts
- 4. Zeile Das Programm „wartet“ für acht Sekunden, die Motoren fahren also acht Sekunden vorwärts und weiterhin nach rechts
- 5. Zeile Die Motoren A und B werden gebremst

Aufgabe 2.4

2.2.6 Bremsen

Bremsen ist nicht gleich bremsen. Der NXT verfügt über zwei verschiedene Möglichkeiten die Motoren zu bremsen, diese werden hier erklärt.

2.2.6.1 Bremsen mit Off()

Die Funktion **Off(AUSGANG_MOTOR)** (engl. „aus“) bremst die Motoren aktiv ab. Nach der Bewegung werden die Motoren abrupt angehalten.

2.2.6.2 Bremsen mit Float()

Die Funktion **Float(AUSGANG_MOTOR)** (engl. „gleiten“) lässt die Motoren auslaufen. Nach der Bewegung werden die Motoren nicht mehr mit Strom versorgt, sie laufen nach und der Roboter legt somit noch eine gewisse Strecke zurück.

Am besten sieht man den Unterschied in einem praktischen Beispiel. Die Motoren werden zweimal völlig ident angesteuert, jedoch unterschiedlich gebremst. Bei der zweiten Bewegung ist eine längere Fahrstrecke zu erkennen:

```
1 task main()
2 {
3   OnFwd(OUT_AB, 100);
4   Wait(1000);
5   Off(OUT_AB);
6
7   Wait(2000);
8
9   OnFwd(OUT_AB, 100);
10  Wait(1000);
11  Float(OUT_AB);
12 }
```

Im main-Task passiert folgendes:

- 1. Zeile Motor A und B fahren mit 100% Leistung nach vorne
- 2. Zeile Das Programm „wartet“, die Motoren fahren eine Sekunde vorwärts
- 3. Zeile Die Motoren A und B werden gebremst – die Motoren werden abrupt angehalten
- 7. Zeile Der Roboter steht für zwei Sekunden still, damit eine eindeutige Trennung zwischen den beiden Bewegungen entsteht
- 9. Zeile Motor A und B fahren mit 100% Leistung nach vorne
- 10. Zeile Das Programm „wartet“, die Motoren fahren eine Sekunde vorwärts
- 11. Zeile Die Motoren A und B werden nicht mehr mit Strom versorgt – der Roboter rollt aus

Aufgabe 2.5

2.3 Konstanten einführen

Die Programme waren bis jetzt alle noch nicht sehr komplex. Die meisten hatten kaum mehr als zwei Bewegungsabläufe. Ein längeres Programm wird ein Vielfaches solcher Anweisungen haben. Wenn ein langes Programm geschrieben ist, und dann etwas doch nicht passt und verändert gehört, müssten alle als Parametern übergebenen Werte einzeln ausgetauscht werden. Eine Lösung für dieses Problem stellt die Verwendung von Konstanten dar.

Einer Konstante wird ein Wert fest zugewiesen, dieser kann während des Programms nicht mehr verändert werden (siehe Kap. „2.7 Variablen“). Sie ist sozusagen ein „Platzhalter“ für den eigentlichen Wert.

In den folgenden beiden Grafiken ist der Unterschied deutlich erkennbar:

```
1 // OHNE Konstanten
2
3 task main()
4 {
5   OnFwd(OUT_AB, 100);
6   Wait(500);
7   Off(OUT_AB);
8
9   OnFwd(OUT_A, 50);
10  Wait(1000);
11  Off(OUT_AB);
12
13  OnFwd(OUT_AB, 100);
14  Wait(500);
15  Off(OUT_AB);
16 }
```

```
1 // MIT Konstanten
2
3 #define VOLLGAS 100
4 #define DREHEN 50
5
6 task main()
7 {
8   OnFwd(OUT_AB, VOLLGAS);
9   Wait(500);
10  Off(OUT_AB);
11
12  OnFwd(OUT_A, DREHEN);
13  Wait(1000);
14  Off(OUT_AB);
15
16  OnFwd(OUT_AB, VOLLGAS);
17  Wait(500);
18  Off(OUT_AB);
19 }
```

Im rechten Quelltext werden zwei Konstanten eingeführt, **VOLLGAS** und **DREHEN**. Die Definition einer Konstante erfolgt immer **vor** dem main-Task nach diesem Schema:

#define NAME WERT

Auf **#define** (engl. „definiere“) folgt der beliebige Name der Konstante und dann der Wert, der eine Zahl oder ein Text sein kann. **Wichtig! Die Zuweisung wird nicht mit einem Strichpunkt beendet!**

Der Name einer Konstante muss mit einem Buchstaben beginnen und darf keine Umlaute enthalten. Um eine deutliche Unterscheidung vom restlichen Quelltext zu erreichen, ist es üblich, Konstanten mit reinen Großbuchstaben zu benennen.

In den Zeilen im rechten Beispielcode passiert folgendes:

- 3. Zeile Die Konstante „**VOLLGAS**“ wird mit dem Wert „100“ definiert
- 4. Zeile Die Konstante „**DREHEN**“ wird mit dem Wert „50“ definiert
- 8. Zeile Das Programm holt sich mit Hilfe der Konstante „**VOLLGAS**“ den definierten Wert „100“ – der Roboter fährt also mit 100% Leistung vorwärts
- 12. Zeile Das Programm holt sich mit Hilfe der Konstante „**DREHEN**“ den definierten Wert „50“ – der Roboter fährt also mit 50% Leistung vorwärts
- 16. Zeile siehe Zeile 8

Aufgabe 2.6

2.4 Ausgabe von Text und Zahlen am LCD

Mit zwei einfachen Funktionen ist es möglich, Text und Zahlen am LC-Display auszugeben. So kann der NXT dann z.B. den Status, Zähler oder den Wert eines Lichtsensors anzeigen. Das Display verfügt über 100 x 64 Pixel.

2.4.1 Ausgabe von Text

Die Textausgabe erfolgt mit der Funktion **TextOut()**. Dieser Funktion müssen 3 Parameter **TextOut(X_KOORDINATE, ZEILE, "TEXT")** übergeben werden. Mit dieser Funktion erfolgt auch die Ausgabe von Sonderzeichen.

2.4.2 Ausgabe von Zahlen

Auch Zahlen müssen als Zeichenkette (String) ausgegeben werden. Das übernimmt die Funktion **NumOut()** (engl. Num = „number“ = „Nummer, Zahl“). Der Funktion müssen 3 Parameter **NumOut(X_KOORDINATE, ZEILE, ZAHL)** übergeben werden.

```
1 task main()
2 {
3   TextOut(0, LCD_LINE1, "Links" );
4   NumOut(35, LCD_LINE2, 1248);
5   TextOut(60, LCD_LINE3, "Rechts" );
6   Wait(10000);
7 }
```



Links 1248 Rechts

X_KOORDINATE beschreibt bei welchem Pixel die Ausgabe des Strings starten soll. Am Display stehen 100 Pixel für die **X_KOORDINATE** zur Verfügung. Die Nummerierung der 100 Pixel fängt jedoch bei 0 an – somit kann dieser Parameter die Werte 0 bis 99 annehmen.

ZEILE gibt an, in welcher der acht möglichen Zeilen der Text ausgegeben werden soll. Insgesamt gibt es acht Möglichkeiten: **LCD_LINE1**, **LCD_LINE2**, **LCD_LINE3**, **LCD_LINE4**, **LCD_LINE5**, **LCD_LINE6**, **LCD_LINE7** und **LCD_LINE8**.

Der auszugebende **"TEXT"** wird als Zeichenkette, als sog. *String* übergeben und muss zwischen zwei Anführungszeichen stehen.

Die auszugebende **ZAHL** wird auch als Zahl übergeben und von der Funktion **NumOut()** automatisch in einen String umgewandelt.

Im main-Task passiert folgendes:

- 3. Zeile Der String „Links“ wird in Zeile 1 ab der x-Koordinate 0 ausgegeben
- 4. Zeile Der String 1248 wird in Zeile 2 ab der x-Koordinate 35 ausgegeben
- 5. Zeile Der String „Rechts“ wird in Zeile 3 ab der x-Koordinate 60 ausgegeben
- 6. Zeile **Wichtig!** Das **Wait()** ist entscheidend – ohne **Wait()** würde sich das Programm sofort wieder schließen und die Anzeige gleich wieder verschwinden

Aufgaben 3

2.5 Schleifen und Verzweigungen

Schleifen und Verzweigungen sind Programmstrukturen, die das Programmieren verkürzen und den Roboter erst komplexere Aufgaben bewältigen.

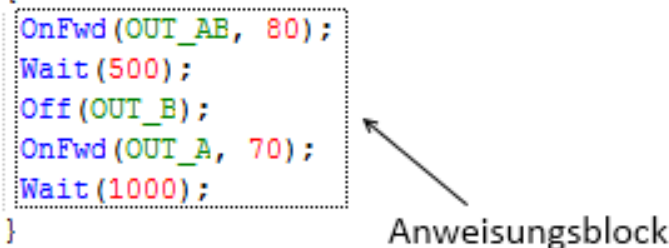
2.5.1 Die repeat-Schleife

Die einfachste Schleife ist die **repeat**-Schleife (engl. „wiederholen“). Sie gibt an, wie oft sich der angegebene Programmteil wiederholen soll. Sie erspart oft viel Tipparbeit.

Struktur: **repeat**(Anzahl der Wiederholungen)
 {
 Anweisung1;
 Anweisung2;
 Anweisung3;
 ...
 }

Auf ein Programm umgelegt sieht dies so aus:

```
1 task main()
2 {
3   repeat(4)
4   {
5     OnFwd(OUT_AB, 80);
6     Wait(500);
7     Off(OUT_B);
8     OnFwd(OUT_A, 70);
9     Wait(1000);
10  }
11  Off(OUT_AB);
12 }
```



Anweisungsblock

Im main-Task passiert folgendes:

- 3. Zeile Die **repeat**-Schleife leitet den Bereich ein, der (in diesem Fall) vier Mal wiederholt werden soll
- 5. Zeile Der Anweisungsblock, der zwischen den beiden geschwungenen Klammern bis 9. Z. steht, wird vier Mal ausgeführt
- 11. Zeile Nachdem die Schleife vier Mal durchgelaufen ist, setzt das Programm nach der Schleife fort und bremst die Motoren A und B

Aufgabe 4.1

2.5.2 Die while-Schleife

Die **while**-Schleife (engl. „während, solange“) enthält eine Bedingung. Solange diese Bedingung erfüllt ist, wird die Schleife ausgeführt.

Nach dem Schlüsselwort **while** wird zuerst in runden Klammern die Bedingung angegeben und danach der Anweisungsblock, der solange die Bedingung erfüllt ist ausgeführt werden soll, in geschweiften Klammern.

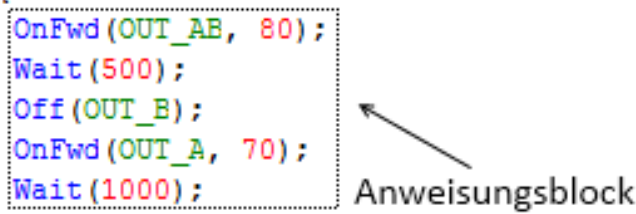
Struktur:

```
while (Bedingung)
{
    Anweisung1;
    Anweisung2;
    Anweisung3;
    ...
}
```

Im folgenden Beispiel handelt es sich um eine Endlosschleife, da die Bedingung IMMER erfüllt ist:

(Für weitere Möglichkeiten komplexere Bedingungen zu erstellen siehe Kap. „2.5.3.1 Vergleichsoperatoren“ und „2.5.3.2 Verknüpfungsoperatoren“.)

```
1 task main()
2 {
3   while(1==1)
4   {
5       OnFwd(OUT_AB, 80);
6       Wait(500);
7       Off(OUT_B);
8       OnFwd(OUT_A, 70);
9       Wait(1000);
10  }
11 }
```



Im main-Task passiert folgendes:

- 3. Zeile Die **while**-Schleife mit der Bedingung (1==1) ist immer erfüllt, sie läuft endlos
- 5. Zeile Der Anweisungsblock, der zwischen den beiden geschweiften Klammern
- bis 9. Z. steht, wird ausgeführt, bis der Akku leer ist oder der Stopp-Button gedrückt wird

Aufgabe 4.2

2.5.3 Die if-Verzweigung

Mit der **if**-Verzweigungen (engl. „wenn“) kann der Roboter selbstständig Entscheidungen treffen. Nach dem Schlüsselwort **if** wird in runden Klammern mindestens eine Bedingung angegeben, danach folgt ein Anweisungsblock in geschweiften Klammern. Optional kann ein **else** (engl. „sonst“) hinzugefügt werden, dass immer in Aktion tritt, wenn die Bedingung **nicht** erfüllt ist. **if**-Verzweigungen nehmen vor allem in der Auswertung von Sensoren einen wichtige Funktion ein (siehe Kap. „2.6 Die Sensoren“).

Struktur:

```
if (Bedingung)
{
    //Ist die Bedingung erfüllt, werden diese
    Anweisungen ausgeführt

    Anweisung1;
    Anweisung2;
    ...
}
else
{
    //Ist die Bedingung NICHT erfüllt, werden diese
    Anweisungen ausgeführt

    Anweisung3;
    Anweisung4;
    ...
}
```

2.5.3.1 Vergleichsoperatoren für Bedingungen

In den Bedingungen einer **if**-Verzweigung und einer **while**-Schleife können folgende Operatoren für den Vergleich von zwei Werten verwendet werden:

<u>Operator</u>	<u>Bedeutung</u>	<u>Beispiel</u>
==	ist gleich	if (Sensor(IN_1) == 1) { ... }
!=	ist ungleich	if (Sensor(IN_1) != 1) { ... }
<	kleiner als	if (Sensor(IN_4) < 4) { ... }
>	größer als	if (Sensor(IN_4) > 20) { ... }
<=	kleiner gleich	if (Sensor(IN_4) <= 10) { ... }
>=	größer gleich	if (Sensor(IN_4) >= 25) { ... }

2.5.3.2 Verknüpfungsoperatoren für Bedingungen

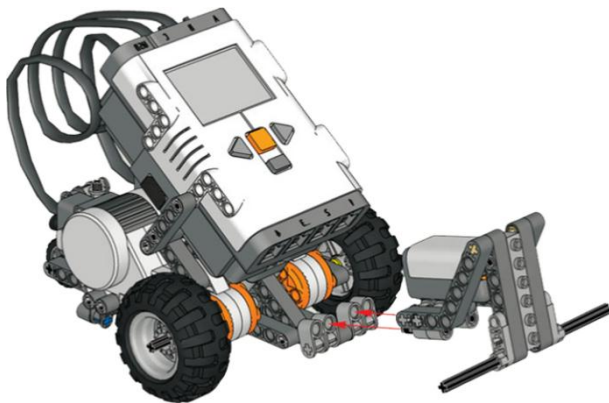
Weiters gibt es die Operatoren **&&** (*UND*) und **||** (*ODER*) falls mehrere Bedingungen erfüllt sein sollen. Bei einer Verknüpfung mit **&&** (*UND*) ist die Gesamtbedingung erst dann erfüllt, wenn beide Teilbedingungen erfüllt sind. Bei der Verknüpfung **||** (*ODER*) gilt die Gesamtbedingung auch dann als erfüllt, wenn mind. eine der Teilbedingungen erfüllt ist.

&&	<i>UND</i>	if ((Sensor(IN_1)==1) && (Sensor(IN_2)==5)) { ... }
 	<i>ODER</i>	if ((Sensor(IN_1)==1) (Sensor(IN_2)==5)) { ... }

2.6 Die Sensoren

Die Sensoren sind für den NXT die Verbindung zur Außenwelt bzw. zur Wettbewerbsstrecke und somit eine Art Sinnesorgane für den NXT Brick. Über die vier Eingänge (*an der Unterseite des Brick*) erhält dieser die Inputs (engl. „Eingabe“), auf die er dann reagieren kann. Auf diese Inputs soll dann mit Outputs reagiert werden z.B. mit einer Rückwärtsbewegung oder einer Schleife.

Bevor die Werte der Sensoren im Programm verwendet werden können, müssen die Eingänge den Sensoren erst zugeordnet werden.



2.6.1 Der Tastsensor

Der Tastsensor ist der einfachste aller Sensoren, weil er nur zwei verschiedene Werte liefert: gedrückt oder nicht gedrückt bzw. 1 oder 0. Er ist jedoch auch einer der wichtigsten bei Wettbewerben, da er sehr leicht und effektiv eingesetzt werden kann, Hindernisse zu erkennen und diesen auszuweichen.



Als erstes muss dem Eingang, an dem ein Sensor angesteckt ist, der Tastsensor zugeordnet werden. Dies geschieht mit der Funktion **SetSensorTouch(IN_X)** (engl. „set“ = „bestimmen“, „touch“ = „berühren“). Diese Funktion gilt **nur** für den Tastsensor und erklärt den Sensor am jeweiligen Eingang (**IN_1**, **IN_2**, **IN_3** oder **IN_4**) zum Tastsensor.

Mit der Funktion **Sensor(IN_X)** greift man auf die Werte, die die jeweiligen Eingänge bzw. Sensoren liefern, zu. Als Parameter übergibt man den jeweiligen Eingang (**IN_1**, **IN_2**, **IN_3** oder **IN_4**). Die Funktion **Sensor(IN_X)** liefert beim Tastsensor die Werte 0 (nicht gedrückt) oder 1 (gedrückt) zurück.

```
1 task main()
2 {
3   SetSensorTouch(IN_1);
4
5   while (Sensor(IN_1) != 1)
6   {
7     OnFwd(OUT_AB, 75);
8   }
9
10  if (Sensor(IN_1) == 1)
11  {
12    OnRev(OUT_AB, 50);
13    Wait(2000);
14    Off(OUT_AB);
15  }
16 }
```

Im main-Task passiert folgendes:

- 3. Zeile Eingang 1 wird ein Tastsensor zugeordnet
- 5. Zeile Durch die **while**-Schleife fährt der Roboter nach vorne, solange der Tastsensor den Wert 0, also nicht gedrückt, zurückliefert
- 10. Zeile Sobald der Tastsensor gedrückt wird, liefert der Sensor den Wert 1 zurück, die while-Schleife wird verlassen und die Bedingung der **if**-Verzweigung ist erfüllt
- 12. Zeile Der Roboter fährt für zwei Sekunden zurück und bleibt daraufhin stehen bis 14. Z.

Aufgabe 5.1

2.6.2 Der Lichtsensor

Der Lichtsensor kann die Intensität des reflektierten Lichts des Untergrunds messen. So wird es dem Roboter ermöglicht, verschiedene Helligkeitswerte zu erkennen. Der Sensor liefert Helligkeitswerte im Bereich zwischen 0 und 100. Besonders bei Wettbewerben ist diese Fähigkeit essentiell, da der Roboter oft einer Linie folgen muss.



Als erstes muss dem Eingang, an dem der Lichtsensor angesteckt ist, der Lichtsensor zugeordnet werden. Dies geschieht mit der Funktion **SetSensorLight(IN_X)** (engl. „light“ = „Licht“). Diese Funktion gilt **nur** für den Lichtsensor und erklärt den Sensor am jeweiligen Eingang (**IN_1**, **IN_2**, **IN_3** oder **IN_4**) zum Lichtsensor.

Die Funktion **Sensor(IN_X)** liefert beim Lichtsensor Werte zwischen 0 (geringste Reflektion = dunkel) und 100 (maximale Reflektion = hell) zurück.

```
1 task main()
2 {
3   SetSensorLight(IN_3);
4   OnFwd(OUT_AB, 30);
5
6   while(1==1)
7   {
8     NumOut(0, LCD_LINE1, Sensor(IN_3));
9
10    if (Sensor(IN_3)<40)
11    {
12      Off(OUT_AB);
13    }
14  }
15 }
```

Im main-Task passiert folgendes:

- 3. Zeile Eingang 3 wird ein Lichtsensor zugeordnet
- 4. Zeile Die Motoren fahren los
- 6. Zeile Es wird eine Endlosschleife gestartet
- 8. Zeile Die Werte des Sensors am Eingang 3, also des Lichtsensors, werden kontinuierlich ausgegeben
- 10. Zeile Der Roboter fährt so lange, wie die Bedingung der **if**-Verzweigung **NICHT** erfüllt ist
- 12. Zeile Ist sie erfüllt, werden die Motoren gebremst

Aufgabe 5.2

2.6.3 Der Tonsensor

Der Tonsensor ist ein kleines Mikrofon, das Geräusche in der Umgebung wahrnimmt und dabei die Lautstärke (in Dezibel) und die Tonhöhe messen kann. Er wird in der Praxis (mit Ausnahme von RoboCup Dance) jedoch relativ selten eingesetzt.



Als erstes muss dem Eingang, an dem der Tonsensor angesteckt ist, der Tonsensor zugeordnet werden. Dies geschieht mit der Funktion **SetSensorSound(IN_X)** (engl. „sound“ = „Ton“). Diese Funktion gilt **nur** für den Tonsensor und erklärt den Sensor am jeweiligen Eingang (**IN_1**, **IN_2**, **IN_3** oder **IN_4**) zum Tonsensor.

Die Funktion **Sensor(IN_X)** liefert beim Tonsensor Werte zwischen 0 (minimale Lautstärke) und 100 (maximale Lautstärke) zurück.

```
1 task main()
2 {
3   SetSensorSound(IN_4);
4
5   while(1==1)
6   {
7     NumOut(0, LCD_LINE1, Sensor(IN_4));
8     Wait(200);
9     ClearScreen();
10  }
11 }
```

Im main-Task passiert folgendes:

- 3. Zeile Eingang 4 wird ein Tonsensor zugeordnet
- 5. Zeile Es wird eine Endlosschleife gestartet
- 7. Zeile Die Werte des Sensors am Eingang 4, also des Tonsensors, werden kontinuierlich ausgegeben
- 8. Zeile Der aktuelle Wert wird für 0,2 Sekunden angezeigt (Zeitspanne, die ca. für das Auge nötig ist, den aktuellen Wert zu erkennen)
- 9. Zeile Mit der Funktion **ClearScreen()** (engl. „clear“ = „löschen“, „Screen“ = „Bildschirm“) wird der LCD geleert

Aufgabe 5.3

2.6.4 Der Ultraschallsensor

Der Ultraschallsensor ist der modernste und komplexeste Sensor des NXT Kits. Mit ihm können Entfernungen zu einem Objekt relativ genau gemessen werden. Dazu wird die Technik der Reflexion von Ultraschall benützt.



Als erstes muss dem Eingang, an dem der Ultraschallsensor angesteckt ist, der Ultraschallsensor zugeordnet werden. Dies geschieht mit der Funktion **SetSensorLowspeed(IN_X)** (engl. „low“ = „langsam“, „speed“ = „Geschwindigkeit“). Diese Funktion gilt **nur** für den Ultraschallsensor und erklärt den Sensor am jeweiligen Eingang (**IN_1**, **IN_2**, **IN_3** oder **IN_4**) zum Ultraschallsensor.

Die Funktion **SensorUS(IN_X)** liefert beim Ultraschallsensor Werte (in cm) zwischen 0 (wobei jedoch der Wert 4 der minimal messbare Abstand ist) und 255 (maximal messbarer Abstand) zurück.

```
1 task main()
2 {
3   SetSensorLowspeed(IN_2);
4
5   while (SensorUS(IN_2) >= 15)
6   {
7     OnFwd(OUT_AB, 50);
8   }
9   Off(OUT_AB);
10 }
```

Im main-Task passiert folgendes:

- 3. Zeile Eingang 2 wird ein Ultraschallsensor zugeordnet
- 5. Zeile Eine **while**-Schleife startet, die den Roboter solange vorwärts fahren lässt,
- bis 8. Z. solange der Sensor die Werte größer gleich 15 liefert
- 6. Zeile Ist die Bedingung der **while**-Schleife nicht mehr erfüllt, verlässt das Programm
- die **while**-Schleife und bremst beide Motoren ab

Aufgabe 5.4

2.7 Variablen einführen

In NXC gibt es zwei Möglichkeiten Werte zu speichern, mit Konstanten (siehe Kap. „2.3 Konstanten einführen“) und Variablen. Konstanten besitzen, einmal definiert, **immer** denselben Wert, während Variablen – wie aus der Mathematik bekannt - verschiedene Werte annehmen und auch **verändert** werden können.

2.7.1 Deklaration von Variablen

Zuerst müssen Variablen deklariert werden. *(Eine Deklaration ist eine Festlegung verschiedener Eigenschaften – in diesem Fall des Variablentyps.)* Wie die Konstanten werden auch Variablen vor dem main-Task deklariert. Am Anfang der Deklaration steht der Variablen-Typ, dann der Variablen-Name (bei den Namen hat man sich darauf geeinigt, Variablen mit Kleinbuchstaben zu benennen). Die Deklaration wird mit einem Strichpunkt ; beendet.

Die zwei am häufigsten Verwendeten Variablentypen heißen **Integer** (engl. „Ganzzahl“) und **String** (engl. „Zeichenkette“).

Eine **Integer**-Variable wird verwendet, wenn eine (ganze) Zahl abgespeichert werden soll. Deshalb bieten sich **Integer**-Variablen perfekt dafür an, Werte von Sensoren in ihnen abzuspeichern.

String-Variablen sind Zeichenketten (z.B. eine Folge von Buchstaben) und werden deshalb oft für die Ausgabe am Display eingesetzt. Die Schlüsselwörter für die Deklaration von *Integer*- und *String*-Variablen lauten **int** und **string**.

```
1 // Deklaration der Variablen
2
3 int licht;
4 int x;
5
6 string text;
7
8 task main()
9 {
10 // Programm
11 }
```

Im Programmcode passiert folgendes:

- 3. Zeile Eine Variable vom Typ Integer mit dem Namen „licht“ wird deklariert
- 4. Zeile Eine Variable vom Typ Integer mit dem Namen „x“ wird deklariert
- 5. Zeile Eine Variable vom Typ String mit dem Namen „text“ wird deklariert

2.7.2 Variablen einen Wert zuweisen

Die Variable existiert nun, doch wie bekommt sie einen Wert? Es besteht entweder die Möglichkeit den Wert gleich bei der Deklaration zuzuweisen oder später während des Programms, wann immer die Variable benötigt wird.

Eine Variable kann generell und auch im Verlauf des Programmes unterschiedliche Werte annehmen. So kann man einer Variable eine Zahl, einen Text oder sogar den Wert einer anderen Variable übergeben. Einer **Integer**-Variable kann somit direkt ein Eingang, bzw. ein Sensorwert, übergeben werden. Bei der **String**-Variable muss man die Zeichenkette zwischen zwei Anführungszeichen „“ setzen.

```
1 int tast;
2 int y;
3
4 int x=8; //Integer-Variable deklariert und zugewiesen
5 string text = "Hallo!"; //String-Variable deklariert und zugewiesen
6
7 task main()
8 {
9   SetSensorTouch(IN_1);
10
11   text = "Zeichenkette"; //Variable erneut zugewiesen/überschrieben
12   y = x; //x ist 8 (y ist x) also ist y auch 8
13
14   while(1==1)
15   {
16     tast = Sensor(IN_1);
17     NumOut(0, LCD_LINE1, y); //Jetzt muss man nur mehr die Variablen
18     NumOut(0, LCD_LINE2, tast); //angeben. Sie fungieren als Platzhalter
19     TextOut(0, LCD_LINE3, text); //für den zugewiesenen Wert, geben diesen aus
20   }
21 }
```

Aufgabe 5.5

2.7.3 Rechnen mit Variablen

Man kann, wie in der Mathematik, mit den Variablen rechnen, als wären sie ganz normale Zahlen. In NXC sind für Integer-Variablen nicht nur die Grundrechenarten enthalten, sondern es stehen auch andere Rechenoperationen wie Wurzel oder Winkelfunktionen zur Verfügung. Der einzige Nachteil bei der Division: Der NXT kann **keine** Kommazahlen, er verwendet nur den ganzzahligen Anteil des Ergebnisses (Bsp. $12:5 = 2$).

```
1 int a, b, c, d;
2 int zahl1, zahl2, zahl3, zahl4;
3
4 //Variablen können auch nebeneinander deklariert werden
5
6 task main()
7 {
8   string wort1 = "Das ist" ;
9   string wort2 = "ein Test" ;
10  string text;
11
12  a=1; b=2; c=3; d=4;
13
14  zahl1 = a + b;
15  zahl2 = d - c;
16  zahl3 = b * c;
17  zahl4 = d / b;
18
19  text = wort1 + " " + wort2;
20 }
```

Aber auch Strings kann man „addieren“. Dabei werden die Zeichenketten zusammengefügt, jedoch sollte man einen Leerstring (" ") in der Mitte nicht vergessen, da im String **text** sonst „Das istein Test“ enthalten wäre.

Weiters gibt es noch zwei „berühmte“ Abkürzungen, die C++ seinen Namen gegeben haben:

```
1 int c, i;
2
3 task main()
4 {
5   c = c + 1;
6   //ist dasselbe wie:
7   c++;
8
9   i = i - 1;
10  //ist dasselbe wie:
11  i--;
12 }
```

2.7.4 Vergleichen von Variablen

Genau wie Zahlen können auch Variablen mit Vergleichsoperatoren verglichen werden (bereits bekannt aus „2.5.3 Die if-Verzweigung“):

<u>Operator</u>	<u>Bedeutung</u>	<u>Beispiel</u>
<code>==</code>	ist gleich	<code>if (a == 1) { ... }</code>
<code>!=</code>	ist ungleich	<code>if (a != 1) { ... }</code>
<code><</code>	kleiner als	<code>if (x < 4) { ... }</code>
<code>></code>	größer als	<code>if (x > 20) { ... }</code>
<code><=</code>	kleiner gleich	<code>if (x <= 10) { ... }</code>
<code>>=</code>	größer gleich	<code>if (x >= 25) { ... }</code>

Auch bei Verknüpfungsoperatoren können Variablen wie Zahlen verwendet werden:

<code>&&</code>	UND	<code>if ((a > 1) && (i == 5)) { ... }</code>
<code> </code>	ODER	<code>if ((a != 1) (i <= 5)) { ... }</code>

2.7.4.1 Zuweisendes und vergleichendes „Ist gleich“

```
1 task main()
2 {
3   int x;
4
5   x = 8;
6
7   if (x == 8)
8   {
9     //Anweisungen
10  }
11 }
```

Was ist der Unterschied?

Was ist der Unterschied zwischen dem einfachen und dem doppelten „Ist gleich“?

Einfaches „Ist gleich“

Das einfache „Ist gleich“ ist das **zuweisende** „Ist gleich“. Es schreibt in die Variable den Wert hinein, in diesem Fall in die Integer-Variable x den Wert 8.

Doppeltes „Ist gleich“

Das doppelte „Ist gleich“ ist das **vergleichende** „Ist gleich“. Es vergleicht die beiden Werte. Das vergleichende „Ist gleich“ (in diesem Fall in der if-Verzweigung) sieht nach, ob auf beiden Seiten der gleiche Wert steht, in diesem Fall, ob in der Variable x der Wert 8 steht.

Aufgabe 5.6

2.8 Subroutinen und Makros erstellen und einsetzen

2.8.1 Subroutinen

Subroutinen (=Unterprogramme) dienen der Übersichtlichkeit des Programms. Bis jetzt sind sämtliche Anweisungen (abgesehen von der Variablen-Deklaration) im main-Task geschehen. Da der main-Task mit der Zeit immer länger und somit unübersichtlicher wird, bedient man sich sog. Subroutinen bzw. Unterprogrammen. Mithilfe von Subroutinen können oftmals benötigte Anweisungs-Abfolgen viel kürzer und schneller aufgerufen werden.

Struktur:

```
sub name()
{
    Anweisung;
}

task main()
{
    Anweisung;
    name();
    Anweisung;
}
```

Auffällig ist, dass Subroutinen **vor** dem main-Task erstellt werden. Subroutinen werden mit dem Schlüsselwort **sub** eingeleitet. Danach folgt ein frei wählbarer Name, der keine Sonderzeichen (bis auf den Grundstrich **_**) enthalten darf. Weiters ist es üblich, Subroutinen mit Kleinbuchstaben zu benennen. Nach dem Namen folgt eine leere Klammer **()**. Wie beim main-Task üblich, stehen die Anweisungen der jeweiligen Subroutine dann zwischen geschwungenen Klammern. Subroutinen werden im main-Task wie eine Funktion mit **name()**; aufgerufen und können immer wieder im Programm aufgerufen werden.

```
1 sub tast()           //Subroutine wird erstellt
2 {
3   OnFwd(OUT_A, 50);    //Anweisungen, die in der Subroutine
4   OnRev(OUT_B, 50);    //ausgeführt werden
5   Wait(1550);
6   Off(OUT_AB);
7 }
8
9 task main()
10 {
11   SetSensorTouch(IN_1);
12   int x = Sensor(IN_1);
13
14   while(x == 0)        //Solange der Tastsensor nicht gedrückt ist,
15   {                    //geradeaus fahren
16     OnFwd(OUT_AB, 60);
17     x = Sensor(IN_1);
18   }
19
20   tast();             //Sobald x=1 ist, springt das Programm aus
21 }                     //der while-Schleife und ruft die Subroutine
22                     //tast() auf und führt sie aus
```

Aufgabe 6.1

2.8.2 Makros

Ein Makro ist eine festgelegte Folge von Befehlen, die man mit nur einem Wort aufrufen und somit das Programm verkürzen kann. Ein Makro kann nur aus einer Zeile bestehen oder es kann auch mehrere Anweisungen über mehrere Zeilen enthalten.

Struktur: **#define name Anweisung;**

```

    #define name Anweisung; Anweisung; \
        Anweisung; \
        Anweisung;

    task main()
    {
        Anweisung;
        name;
        Anweisung;
    }
```

Auch Makros werden **vor** dem main-Task festgelegt. Sie werden (wie schon von den Konstanten bekannt) mit **#define** eingeleitet. Darauf folgt ein frei wählbarer Name, standardmäßig in Kleinbuchstaben. Die Anzahl der zugehörigen Anweisungen ist nicht begrenzt. Um mehrere Anweisungen einem Makro zuzuordnen müssen sie entweder in derselben Zeile (von einem Strichpunkt ; getrennt) stehen oder durch ein Backslash \ jeweils die nächste Zeile verlängert werden. Makros können noch einfacher als Subroutinen aufgerufen werden, nämlich nur über den Namen des Makros.

```

1  #define gas OnFwd(OUT_AB, 75); Wait(1500); Off(OUT_AB);
2  #define rechts OnFwd(OUT_A, 50); Wait(1000); Off(OUT_A);
3
4  #define links OnFwd(OUT_B, 50); \
5          Wait(1000); \
6          Off(OUT_B);
7
8  /*
9      Die ersten beiden Makros werden normal erstellt, das dritte
10     wird über drei Zeilen immer wieder mit Backslash \ verlängert
11 */
12
13 task main()
14 {
15     gas;           //Makro gas wird aufgerufen,
16     rechts;        //Makro rechts wird aufgerufen,
17     gas;           //usw. Makros kann man also auch
18     links;         //so oft man will einsetzen
19     gas;
20 }
```

Aufgabe 6.2

3. Anhang A – Befehlsübersicht

3.1 Motoren

Float (AUSGANG_MOTOR);	
Abschalten der Motoren ohne aktiv zu bremsen	
AUSGANG_MOTOR	OUT_A, OUT_B, OUT_C, OUT_AB, OUT_BC, OUT_AC, OUT_ABC

Off (AUSGANG_MOTOR);	
Abschalten der Motoren mit aktiver Bremsung	
AUSGANG_MOTOR	OUT_A, OUT_B, OUT_C, OUT_AB, OUT_BC, OUT_AC, OUT_ABC

OnFwd (AUSGANG_MOTOR, LEISTUNG);	
Vorwärtsbewegung der Motoren mit variabler Geschwindigkeit zwischen 0% und 100%	
AUSGANG_MOTOR	OUT_A, OUT_B, OUT_C, OUT_AB, OUT_BC, OUT_AC, OUT_ABC
LEISTUNG	ganze Zahl zwischen 0 und 100

OnFwdReg (AUSGANG_MOTOR, LEISTUNG, REGMODE);	
Synchronisierte Vorwärtsbewegung der angegebenen Motoren mit Ausglei- chung von Unterschieden durch Pausieren eines Motors	
AUSGANG_MOTOR	OUT_AB, OUT_BC, OUT_AC, OUT_ABC
LEISTUNG	ganze Zahl zwischen 0 und 100
REGMODE	OUT_REGMODE_SYNC

OnFwdSync (AUSGANG_MOTOR, LEISTUNG, RADIUS);	
Vorwärtsbewegung der Motoren mit variabler Geschwindigkeit. Durch den RADIUS- Parameter drehen sich die Motoren unterschiedlich schnell und fahren somit eine Kurve	
AUSGANG_MOTOR	OUT_AB, OUT_BC, OUT_AC, OUT_ABC
LEISTUNG	ganze Zahl zwischen 0 und 100
RADIUS	ganze Zahl zwischen -127 und 127

OnRev (AUSGANG_MOTOR, LEISTUNG);	
Rückwärtsbewegung der Motoren mit variabler Geschwindigkeit zwischen 0% und 100%	
AUSGANG_MOTOR	OUT_A, OUT_B, OUT_C, OUT_AB, OUT_BC, OUT_AC, OUT_ABC
LEISTUNG	ganze Zahl zwischen 0 und 100

OnRevReg (AUSGANG_MOTOR, LEISTUNG, REGMODE);	
Synchronisierte Rückwärtsbewegung der angegebenen Motoren mit Ausgleich von Unterschieden durch Pausieren eines Motors	
AUSGANG_MOTOR	OUT_AB, OUT_BC, OUT_AC, OUT_ABC
LEISTUNG	ganze Zahl zwischen 0 und 100
REGMODE	OUT_REGMODE_SYNC

OnRevSync (<i>AUSGANG_MOTOR</i> , <i>LEISTUNG</i> , <i>RADIUS</i>);	
Rückwärtsbewegung der Motoren mit variabler Geschwindigkeit. Durch den RADIUS-Parameter drehen sich die Motoren unterschiedlich schnell und fahren somit eine Kurve	
<i>AUSGANG_MOTOR</i>	<i>OUT_AB, OUT_BC, OUT_AC, OUT_ABC</i>
<i>LEISTUNG</i>	ganze Zahl zwischen <i>0</i> und <i>100</i>
<i>RADIUS</i>	ganze Zahl zwischen <i>-127</i> und <i>127</i>

RotateMotor (<i>AUSGANG_MOTOR</i> , <i>LEISTUNG</i> , <i>WINKEL</i>);	
Drehung der Motoren um den angegebenen Winkel	
<i>AUSGANG_MOTOR</i>	<i>OUT_A, OUT_B, OUT_C, OUT_AB, OUT_BC, OUT_AC, OUT_ABC</i>
<i>LEISTUNG</i>	ganze Zahl zwischen <i>0</i> und <i>100</i>
<i>WINKEL</i>	ganze Zahl; positiver Wert → Vorwärtsdrehung negativer Wert → Rückwärtsdrehung

3.2 Timer

Wait (<i>MILLISEKUNDEN</i>);	
Das Programm bleibt für die angegebene Zeit im selben Zustand	
<i>MILLISEKUNDEN</i>	ganze Zahl ≥ 0

3.3 LC-Display

ClearScreen();

Diese Funktion löscht alles, was zuvor im Display angezeigt wurde

NumOut(X_KOORDINATE, ZEILE, ZAHL);

Ausgabe einer Zahl an einer beliebigen Position des Displays

X_KOORDINATE	ganze Zahl zwischen 0 und 99
ZEILE	LCD_LINE1, LCD_LINE2, LCD_LINE3, LCD_LINE4, LCD_LINE5, LCD_LINE6, LCD_LINE7, LCD_LINE8
ZAHL	Zahlenwert oder Variable, die den Zahlenwert beinhaltet

TextOut(X_KOORDINATE, ZEILE, "TEXT");

Ausgabe einer Zeichenkette an einer beliebigen Position des Displays

X_KOORDINATE	ganze Zahl zwischen 0 und 99
ZEILE	LCD_LINE1, LCD_LINE2, LCD_LINE3, LCD_LINE4, LCD_LINE5, LCD_LINE6, LCD_LINE7, LCD_LINE8
"TEXT"	Zeichenkette oder Variable, die eine Zeichenkette beinhaltet, begrenzt von doppelten Anführungszeichen „“

3.4 Schleifen & Verzweigungen

Entscheidungsabfrage mit der if-Verzweigung

```
if (x==1)
{
    Anweisung1;
    Anweisung2;
    Anweisung3;
    ...
}

else
{
    Anweisung4;
    Anweisung5;
    Anweisung6;
    ...
}
```

Wenn die Bedingung der if-Verzweigung erfüllt ist, wird Anweisungsblock 1 ausgeführt (Anweisung1, Anweisung2, ...). Immer, wenn die Bedingung **NICHT** erfüllt ist, springt das Programm in den else-Anweisungsblock. Dieser Anweisungsblock wird dann **genau einmal** ausgeführt

Wiederholung eines Anweisungsblocks mit der repeat-Schleife

```
repeat (x)
{
    Anweisung1;
    Anweisung2;
    Anweisung3;
    ...
}
```

x

Anzahl der gewünschten Wiederholungen

Schleifen mit while

```
while (1==1)
{
    Anweisung1;
    Anweisung2;
    Anweisung3;
    ...
}
```

Der Anweisungsblock der while-Schleife wird immer wieder von vorn ausgeführt, solange die Bedingung der while-Schleife erfüllt ist. Sie kann somit nur einmal oder auch unendlich oft durchlaufen werden

Logische Vergleichsoperatoren		
<u>Operator</u>	<u>Bedeutung</u>	<u>Beispiel</u>
<code>==</code>	ist gleich	<code>if (a == 1) { ... }</code>
<code>!=</code>	ist ungleich	<code>if (a != 1) { ... }</code>
<code><</code>	kleiner als	<code>if (x < 4) { ... }</code>
<code>></code>	größer als	<code>if (x > 20) { ... }</code>
<code><=</code>	kleiner gleich	<code>if (x <= 10) { ... }</code>
<code>>=</code>	größer gleich	<code>if (x >= 25) { ... }</code>
Logische Vergleichsoperatoren können in allen Schleifen und Verzweigungen verwendet werden. Mit ihnen ist es erst möglich, komplexe Programmstrukturen zu erstellen		

Logische Verknüpfungsoperatoren		
<code>&&</code>	<i>UND</i>	<code>if (a > 1) && (i == 5) { ... }</code>
<code> </code>	<i>ODER</i>	<code>if (a != 1) (i <= 5) { ... }</code>
Logische Verknüpfungsoperatoren dienen dazu, beliebig viele Bedingungen zu verknüpfen. Mit ihnen können Bedingungen angegeben werden, bei denen alle Bedingungen (<code>&&</code>) oder mindestens eine der Bedingungen (<code> </code>) erfüllt sein müssen		

3.5 Sensoren

Sensor(IN_X);	
Mit dieser Funktion greift man auf die Werte, die die jeweiligen Eingänge bzw. Sensoren liefern, zu. Sie liefert, abhängig vom gewählten Sensor, unterschiedliche Werte zurück	
<i>IN_X</i>	<i>IN_1, IN_2, IN_3, IN_4</i>
Ist an einem Eingang des <i>Tastsensors</i> angeschlossen...	...liefert die Funktion die Werte 0 (=nicht gedrückt) und 1 (=gedrückt) zurück
Ist an einem Eingang des <i>Lichtsensors</i> angeschlossen...	...liefert die Funktion Werte zwischen 0 (=dunkel) und 100 (=hell) zurück
Ist an einem Eingang des <i>Tonsensors</i> angeschlossen...	...liefert die Funktion Werte zwischen 0 (=leise) und 100 (=laut) zurück

SensorUS(IN_X);	
Nur für den Ultraschallsensor benötigt man eine andere Funktion. Sie liefert die Werte (in cm) zwischen 0 (wobei jedoch der Wert 4 der minimal messbare Abstand ist) und 255 (maximal messbarer Abstand) zurück.	
<i>IN_X</i>	<i>IN_1, IN_2, IN_3, IN_4</i>

SetSensorLight(IN_X);	
Diese Funktion ist nur für den Lichtsensor relevant. Sie legt fest, an welchem Eingang der Lichtsensor angeschlossen ist	
<i>IN_X</i>	<i>IN_1, IN_2, IN_3, IN_4</i>

SetSensorLowSpeed(IN_X);	
Diese Funktion kommt nur bei der Verwendung des Ultraschallsensors zum Einsatz. Sie legt fest, an welchem Eingang der Ultraschallsensor angeschlossen ist	
<i>IN_X</i>	<i>IN_1, IN_2, IN_3, IN_4</i>

SetSensorSound(IN_X);	
Diese Funktion funktioniert nur für den Tonsensor und legt fest, an welchem Eingang der Tonsensor angeschlossen ist	
<i>IN_X</i>	<i>IN_1, IN_2, IN_3, IN_4</i>

SetSensorTouch(IN_X);	
Diese Funktion findet nur beim Tastsensor Verwendung. Sie legt fest, an welchem Eingang der Tastsensor angeschlossen ist	
<i>IN_X</i>	<i>IN_1, IN_2, IN_3, IN_4</i>

3.6 Variablen

int x;

Eine Variable x vom Typ Integer (=ganze Zahl) wird deklariert

string text;

Eine Variable text vom Typ String (=Zeichenkette) wird deklariert

3.7 Präprozessoren

<i>// Kommentar</i>	
Leitet eine einzeilige Bemerkung im Code ein, die keinen Einfluss auf die Funktion des Programmcodes hat. Kommentare dienen nur der Übersicht	

<i>/* Kommentar Kommentar Kommentar */</i>	
Eine mehrzeilige Bemerkung im Code, die keinen Einfluss auf die Funktion des Programmcodes hat	

#define NAME WERT	
Erstellt eine im Programm nicht mehr veränderbare Konstante	
NAME	frei wählbarer NAME der Konstante, üblicherweise in Großbuchstaben
WERT	frei wählbarer ganzzahliger WERT, den die Konstante beinhalten soll

#define name Anweisung1; Anweisung2; Anweisung3; ...;	
Erstellt ein einzeiliges Makro	
name	frei wählbarer name des Makro, üblicherweise in Kleinbuchstaben
Anweisung1; Anweisung2; Anweisung3; ...;	Alle Anweisungen, die in dieser Zeile Platz finden, werden diesem Makro zugeordnet und beim Aufruf des Makros ausgeführt

#define name Anweisung1; \ Anweisung2; \ Anweisung3; \ ...;	
Erstellt ein mehrzeiliges Makro	
name	frei wählbarer name des Makro, üblicherweise in Kleinbuchstaben
Anweisung1; \ Anweisung2; \ Anweisung3; \ ...;	Durch einen Backslash \ wird die Zeile des Makros verlängert. Alle Anweisungen werden so diesem Makro zugeordnet und beim Aufruf des Makros ausgeführt

name;	
Aufruf des Makros mit angegebenen Namen	

3.8 Subroutinen

Subroutine erstellen

<pre>sub name() { Anweisung1; Anweisung2; Anweisung3; ... }</pre>

Erstellt eine Subroutine unter dem angegebenen Namen
--

Subroutine aufrufen

<pre>name();</pre>

Aufruf der Subroutine unter dem festgelegten Namen
--

4. Anhang B – Aufgabensammlung

Aufgabe 1

Roboter mit 2 Motoren



Erstelle ein Programm, mit dem der Roboter 4 Sekunden mit 70% Motorleistung vorwärtsfährt und danach stoppt.



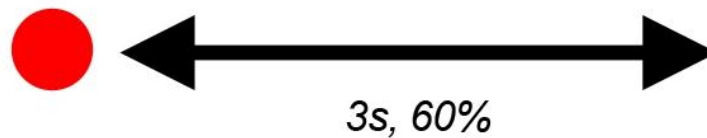
Aufgaben 2.1

Roboter mit 2 Motoren



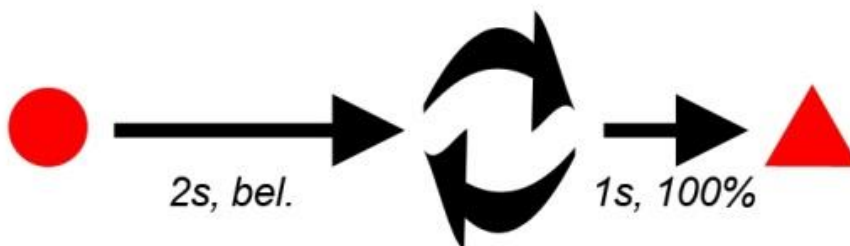
1.

Erstelle ein Programm, mit dem der Roboter 3 Sekunden mit 60% Motorleistung vorwärtsfährt, wieder die gleiche Strecke rückwärtsfährt und danach stoppt.



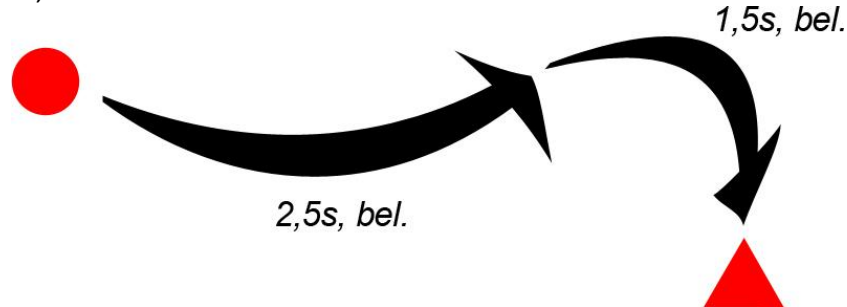
2.

Erstelle ein Programm, mit dem der Roboter 2 Sekunden mit beliebiger Motorleistung vorwärtsfährt, sich auf der Stelle dreht, dann nochmals eine Sekunde mit maximaler Leistung vorwärtsfährt und danach stoppt.



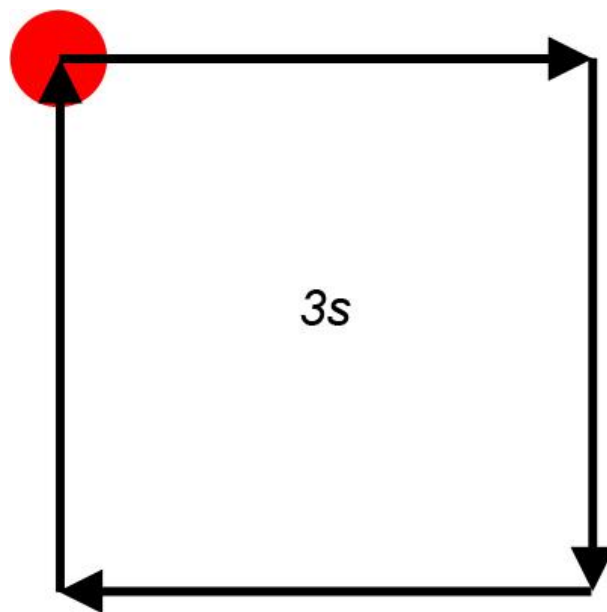
3.

Erstelle ein Programm, mit dem der Roboter 2,5 Sekunden eine leichte Kurve nach links fährt, dann für 1,5 Sekunden eine starke Kurve nach rechts fährt und danach stoppt.



4.

Erstelle ein Programm, mit dem der Roboter 3 Sekunden geradeaus fährt, sich viermal um 90° nach rechts dreht, wieder 3 Sekunden geradeaus fährt und so schließlich am Start wieder zum Stehen kommt.



Aufgaben 2.2

Roboter mit 2 Motoren



1.

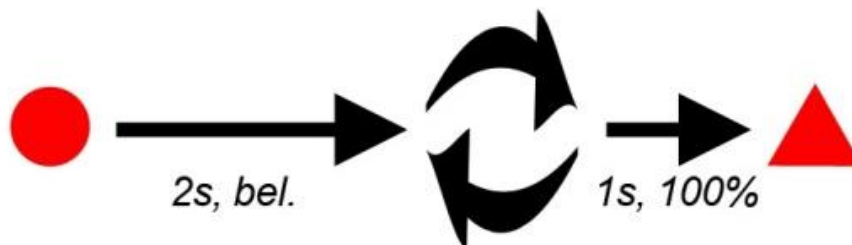
Erstelle ein Programm, mit dem der Roboter 3 Sekunden mit 60% Motorleistung vorwärtsfährt, wieder die gleiche Strecke rückwärtsfährt und danach stoppt. *Verwende dabei die neue Funktion RotateMotor() !*



2.

Erstelle ein Programm, mit dem der Roboter 2 Sekunden mit beliebiger Motorleistung vorwärtsfährt, sich auf der Stelle dreht, dann nochmals eine Sekunde mit maximaler Leistung vorwärtsfährt und danach stoppt.

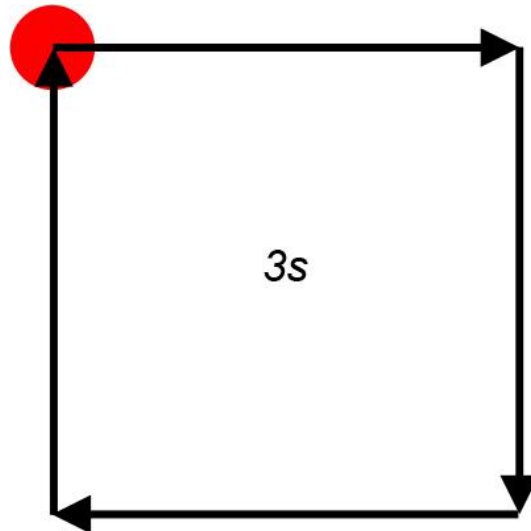
Verwende dabei die neue Funktion RotateMotor() !



3.

Erstelle ein Programm, mit dem der Roboter 3 Sekunden geradeaus fährt, sich viermal um 90° nach rechts dreht, wieder 3 Sekunden geradeaus fährt und so schließlich am Start wieder zum Stehen kommt.

Verwende dabei die neue Funktion `RotateMotor()` !

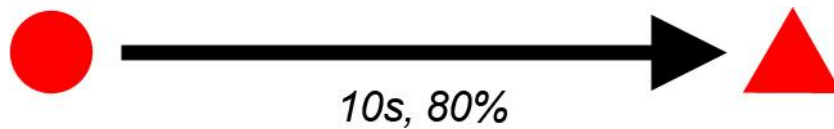


Aufgabe 2.3

Roboter mit 2 Motoren



Erstelle ein Programm, mit dem der Roboter 10 Sekunden mit 80% Motorleistung vorwärtsfährt und danach stoppt. Halte dabei eins der Räder fest und lass es wieder los, um zu sehen, wie der Roboter die Motoren wieder synchronisiert.



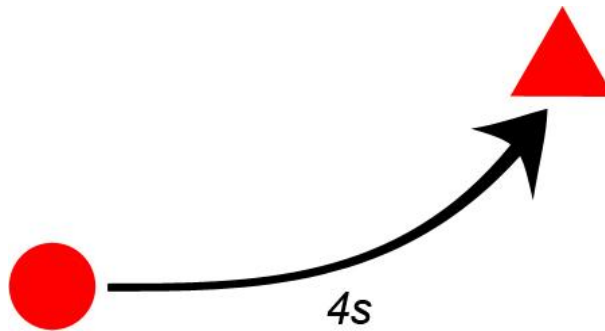
Aufgabe 2.4

Roboter mit 2 Motoren



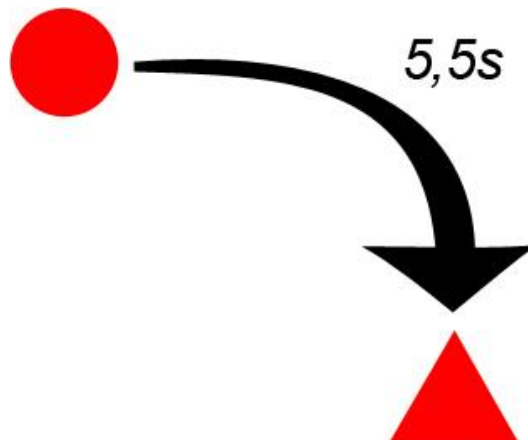
1.

Erstelle ein Programm, mit dem der Roboter mit der neuen Funktion (`OnFwdSync()`) für 4 Sekunden eine leichte Kurve nach links fährt und danach stoppt.



2.

Erstelle ein Programm, mit dem der Roboter mit der neuen Funktion (`OnFwdSync()`) für 5,5 Sekunden eine starke Kurve nach rechts fährt und danach stoppt.



3.

Erstelle ein Programm, mit dem der Roboter 2 Sekunden mit beliebiger Motorleistung vorwärtsfährt, sich genau einmal auf der Stelle dreht (rechts oder links), dann nochmals eine Sekunde mit maximaler Leistung vorwärtsfährt und danach stoppt.



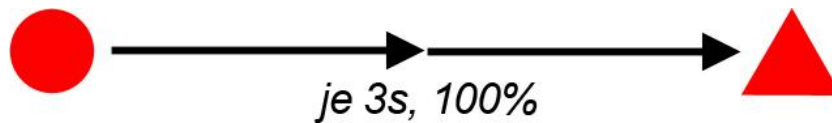
Aufgabe 2.5

Roboter mit 2 Motoren



Erstelle ein Programm, mit dem der Roboter zwei Mal 3 Sekunden mit 100% Leistung geradeaus fährt. Dabei wird der Roboter je einmal mit Off() und einmal mit Float() gebremst. Dazwischen soll eine Pause von 2 Sekunden gewartet werden, um die 2 Bewegungen eindeutig zu trennen.

Beobachte den Unterschied zwischen der Bremsung mit Off() und Float() !



Aufgabe 2.6

Roboter mit 2 Motoren



1.

- a) Führe in die Aufgabenstellung **Aufgabe 2.1.4** Konstanten ein!
- b) Ändere die Konstanten!

2.

- a) Führe in die Aufgabenstellung **Aufgabe 2.4.3** Konstanten ein!
- b) Ändere die Konstanten!

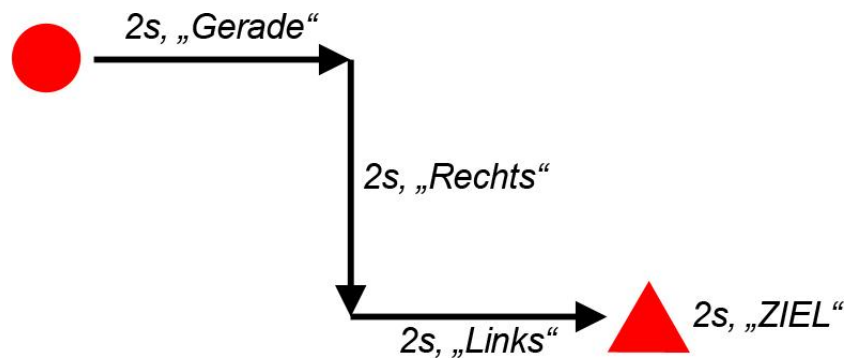
Aufgabe 3

Roboter mit 2 Motoren



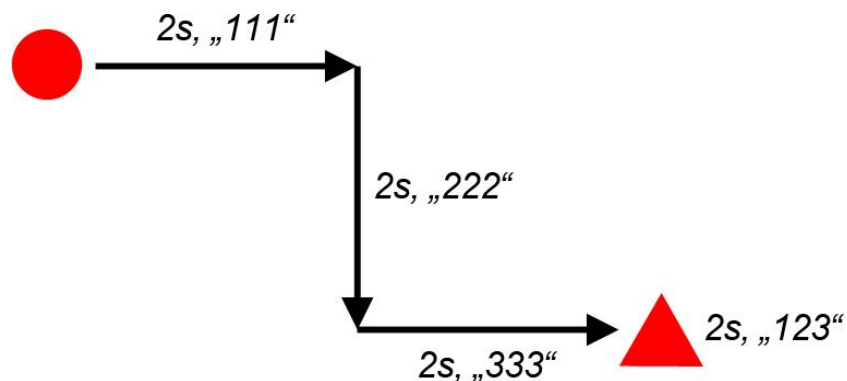
1.

Erstelle ein Programm, mit dem der Roboter 2 Sekunden vorwärtsfährt, sich 90° nach rechts dreht, wieder 2 Sekunden vorwärtsfährt, sich 90° nach links dreht, wieder 2 Sekunden vorwärtsfährt und danach stoppt und noch 2 Sekunden „ZIEL“ anzeigt. Während den einzelnen Bewegungen soll er auf dem Display „Gerade“, „Rechts“ und „Links“ ausgeben.



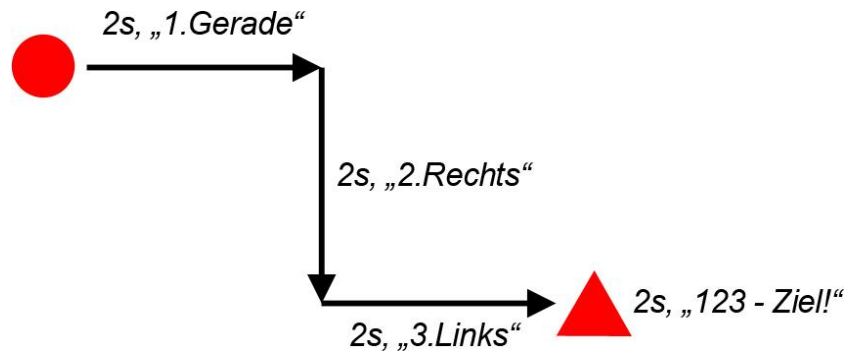
2.

Erstelle ein Programm, mit dem der Roboter 2 Sekunden vorwärtsfährt, sich 90° nach rechts dreht, wieder 2 Sekunden vorwärtsfährt, sich 90° nach links dreht, wieder 2 Sekunden vorwärtsfährt und danach stoppt und noch 2 Sekunden „123“ anzeigt. Während den einzelnen Bewegungen soll er auf dem Display „111“, „222“ und „333“ ausgeben.



3.

Erstelle ein Programm, mit dem der Roboter 2 Sekunden vorwärtsfährt, sich 90° nach rechts dreht, wieder 2 Sekunden vorwärtsfährt, sich 90° nach links dreht, wieder 2 Sekunden vorwärtsfährt und danach stoppt und noch 2 Sekunden „123 – Ziel!“ anzeigt. Während den einzelnen Bewegungen soll er auf dem Display „1.Gerade“, „2.Rechts“ und „3.Links“ ausgeben.



Aufgabe 4.1

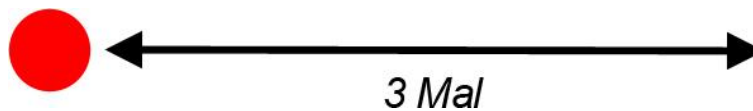
Roboter mit 2 Motoren



1.

Erstelle ein Programm, mit dem der Roboter dreimal 2 Sekunden mit 85% Motorleistung vorwärtsfährt, wieder die gleiche Strecke rückwärtsfährt und danach stoppt.

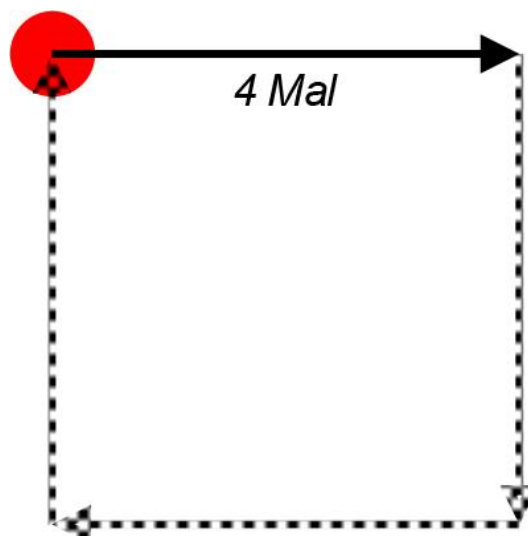
Verwende dabei das repeat()-Statement!



2.

Erstelle ein Programm, mit dem der Roboter viermal 3 Sekunden gradeaus fährt, sich um 90° nach rechts dreht und so sein Fahrweg ein Quadrat beschreibt.

Verwende dabei das repeat()-Statement!



Aufgabe 4.2

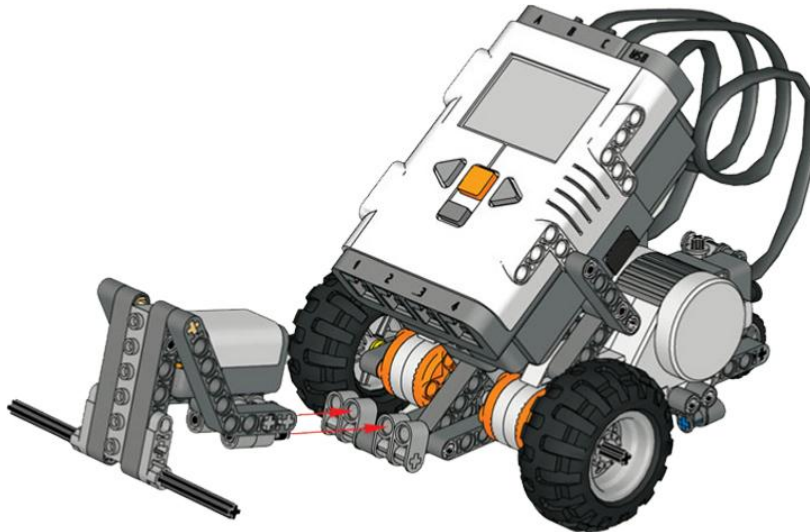
Roboter mit 2 Motoren



Erstelle ein Programm, mit dem der Roboter in einer Endlosschleife mit 80% Motorleistung geradeaus fährt!

Aufgabe 5.1

Roboter mit 2 Motoren, 1 Tastsensor, 1 Hindernis (z.B. Flasche)



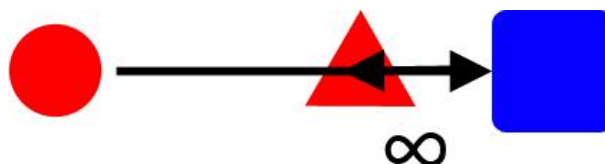
1.

Erstelle ein Programm, mit dem der Roboter solange geradeaus fährt, bis er auf ein Hindernis trifft. Dann soll er 1,5 Sekunden gerade zurück fahren und dann stehen bleiben.



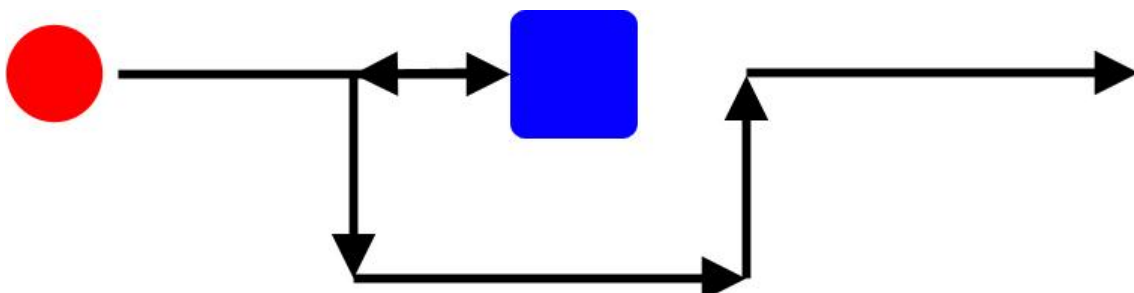
2.

Erstelle ein Programm, mit dem der Roboter in einer Endlosschleife solange geradeaus fährt, bis er auf ein Hindernis trifft, 1,5 Sekunden gerade zurück fährt, kurz stehen bleibt und dann wieder geradeaus startet bis er auf das nächste Hindernis trifft usw.



3.

Erstelle ein Programm, mit dem der Roboter gerade auf ein Hindernis zufährt und bei Berührung ein Ausweichmanöver startet, in dem das Hindernis rechts umfahren werden soll ohne dieses zu berühren. Der Roboter soll danach seinen Weg wie zuvor fortsetzen.



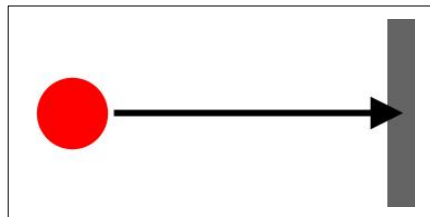
Aufgabe 5.2

Roboter mit 2 Motoren, 1 Lichtsensor, Druckvorlage 1



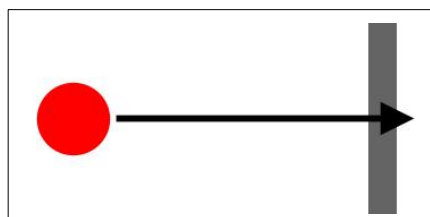
1.

Erstelle ein Programm, mit dem der Roboter solange geradeaus fährt, bis der Sensor die schwarze Fläche (DVL1) erreicht. Auf dieser soll er stehen bleiben.



2.

Erstelle ein Programm, mit dem der Roboter solange geradeaus fährt, bis er die schwarze Fläche (DVL1) passiert und, sobald er wieder die weiße Fläche erreicht, stehen bleibt.



3.

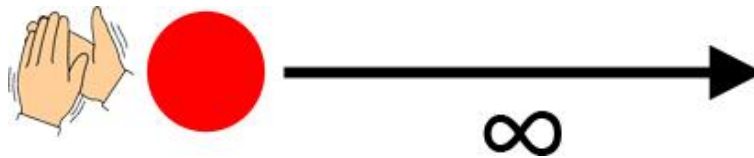
Erstelle ein Programm, mit dem der Roboter laufend den aktuellen Wert des Lichtsensors anzeigt.

Aufgabe 5.3

Roboter mit 2 Motoren, 1 Tonsensor



Erstelle ein Programm, mit dem der Roboter geradeaus fährt, sobald man klatscht.

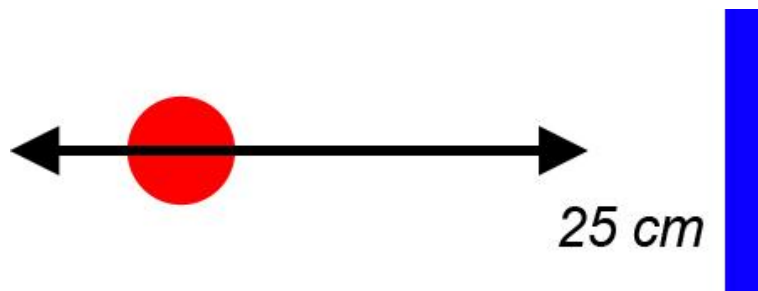


Aufgabe 5.4

Roboter mit 2 Motoren, 1 Ultraschallsensor, 1 Objekt

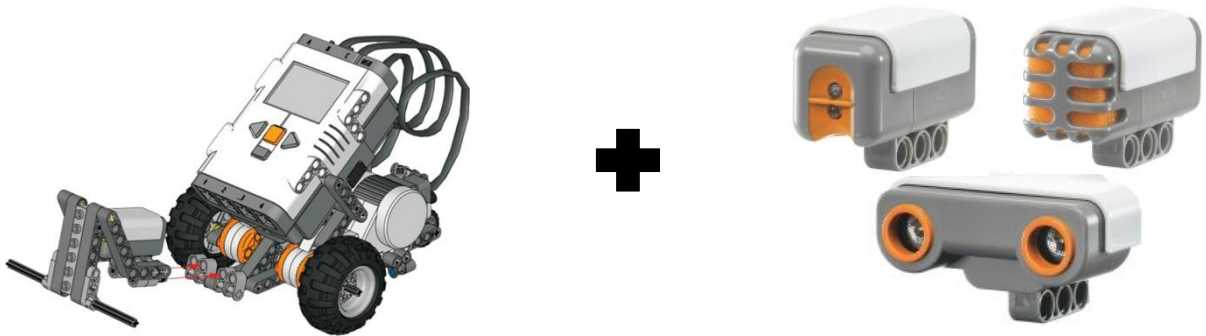


Erstelle ein Programm, mit dem der Roboter gradeaus fährt, bis er sich 25 cm vor einem Objekt befindet. Dort soll der Roboter für 3 Sekunden stehenbleiben und dann zurück fahren.



Aufgabe 5.5

Roboter mit 2 Motoren, 1 Tastsensor, 1 Lichtsensor, 1 Tonsensor, 1 Ultraschallsensor



Erstelle ein Programm, mit dem der Roboter geradeaus fährt und alle Sensoren angeschlossen sind. **Mit Hilfe von Variablen** sollen sämtliche Werte der Sensoren angezeigt werden.

Am Display soll diese Ansicht erscheinen:

```
Tastsensor: 0  
Lichtsensor: 64  
Tonsensor: 2  
USSensor: 0
```

Aufgabe 5.6

Roboter mit 2 Motoren, alle Sensoren, 1 Hindernis, Druckvorlage 1

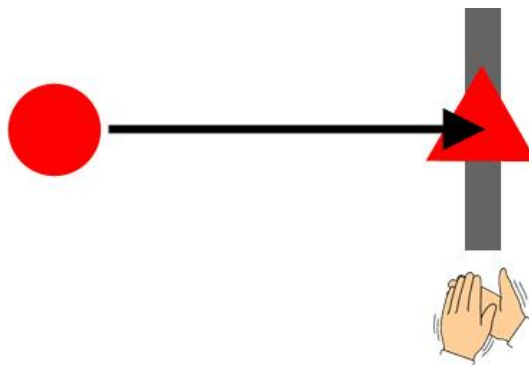


1.

Erstelle ein Programm, mit dem der Roboter geradeaus fährt und erst stoppt, wenn man entweder klatscht **ODER** er gegen ein Hindernis fährt.

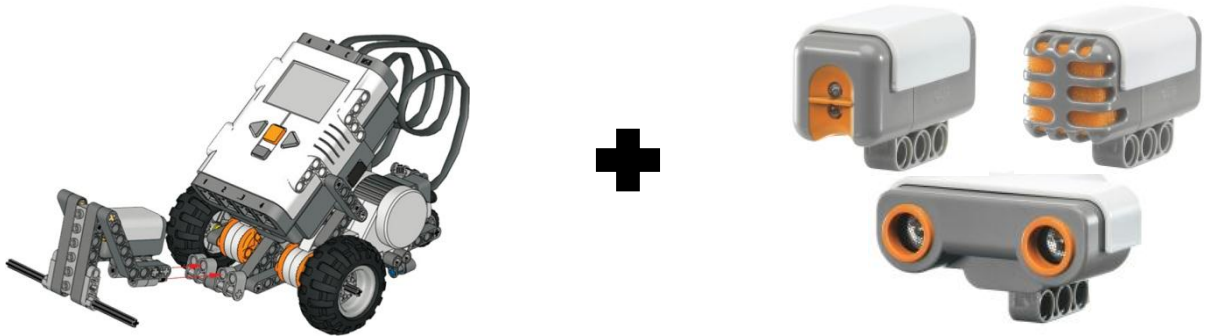
2.

Erstelle ein Programm, mit dem der Roboter langsam geradeaus fährt und erst stoppt, wenn er über die schwarze Linie fährt **UND** dabei gleichzeitig geklatscht wird.



Aufgabe 5.7

Roboter mit 2 Motoren, alle Sensoren, 1 Hindernis, Druckvorlage 2

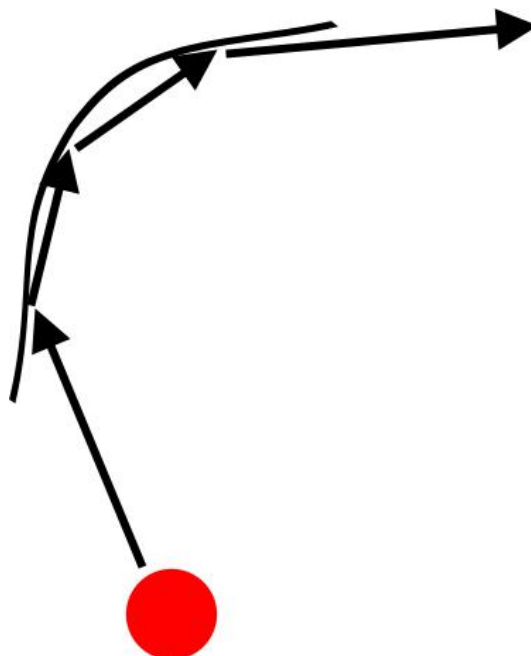


1.

Erstelle ein Programm, mit dem der Roboter geradeaus fährt und die Subroutine **tast()** enthält. In der Routine soll sich der Roboter drei Mal im Kreis drehen. Diese Subroutine soll immer ausgeführt werden, wenn der Roboter auf ein Hindernis stößt.

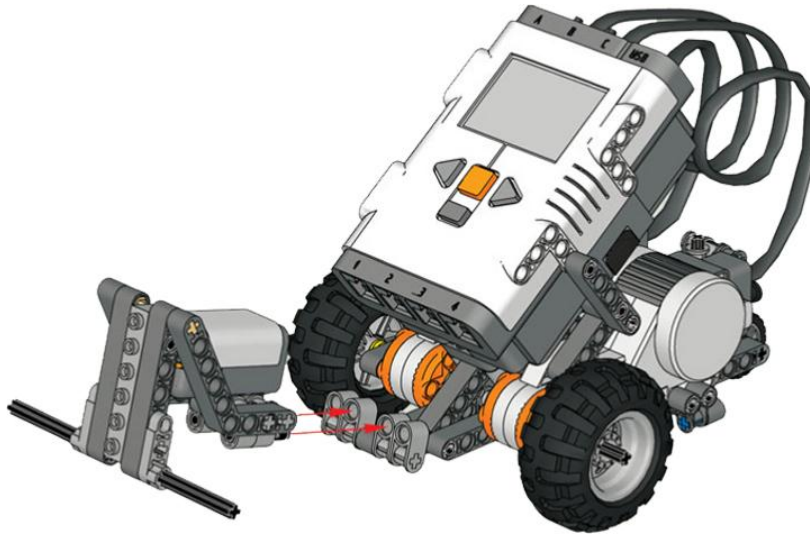
2.

Erstelle ein Programm, mit dem der Roboter geradeaus fährt und die Subroutine **licht()** enthält. In der Routine soll sich der Roboter immer ein Stück nach rechts drehen. Diese Subroutine soll immer ausgeführt werden, wenn der Roboter auf eine schwarze Linie (DVL2) trifft.



Aufgabe 5.7

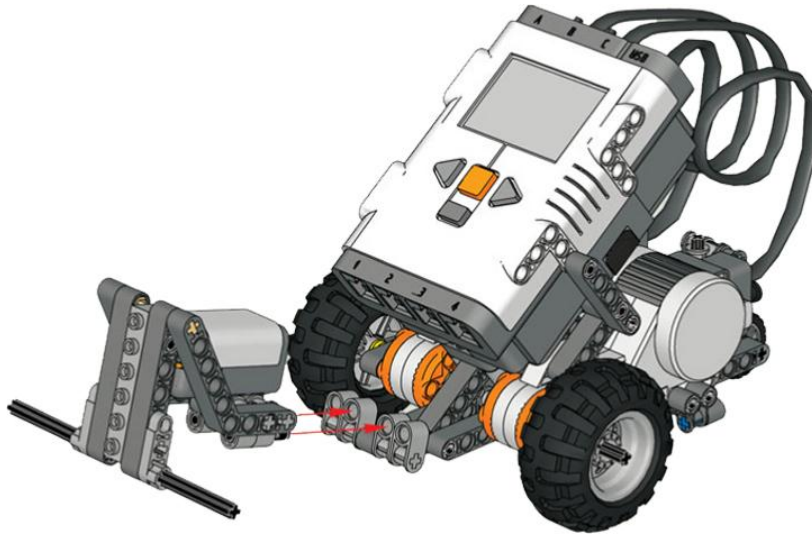
Roboter mit 2 Motoren, 1 Tastsensor, 1 Hindernis



Erstelle ein Programm, mit dem der Roboter geradeaus fährt und das Makro **tast** enthält. In dem Makro soll der Roboter rückwärts fahren und sich leicht nach links drehen. Dieses Makro soll immer ausgeführt werden, wenn der Roboter auf ein Hindernis stößt. Er weicht dem Hindernis somit intuitiv aus.

Aufgabe 6

Roboter mit 2 Motoren, 1 Tastsensor, 1 Hindernis



1.

Erstelle ein Programm, mit dem der Roboter geradeaus fährt und auf ein Hindernis trifft. Wenn er auf das Hindernis trifft, soll ein Makro ausgeführt werden, in dem er kurz rückwärts fährt und kurz stehen bleibt. Danach soll er wieder geradeaus fahren bis er wieder auf das Hindernis trifft. Wenn der Roboter das Hindernis vier Mal berührt hat, soll er endgültig stehen bleiben.

2.

Erstelle ein Programm, mit dem der Roboter geradeaus fährt und erst dann wieder stehen bleibt, nachdem man drei Mal geklatscht hat.



