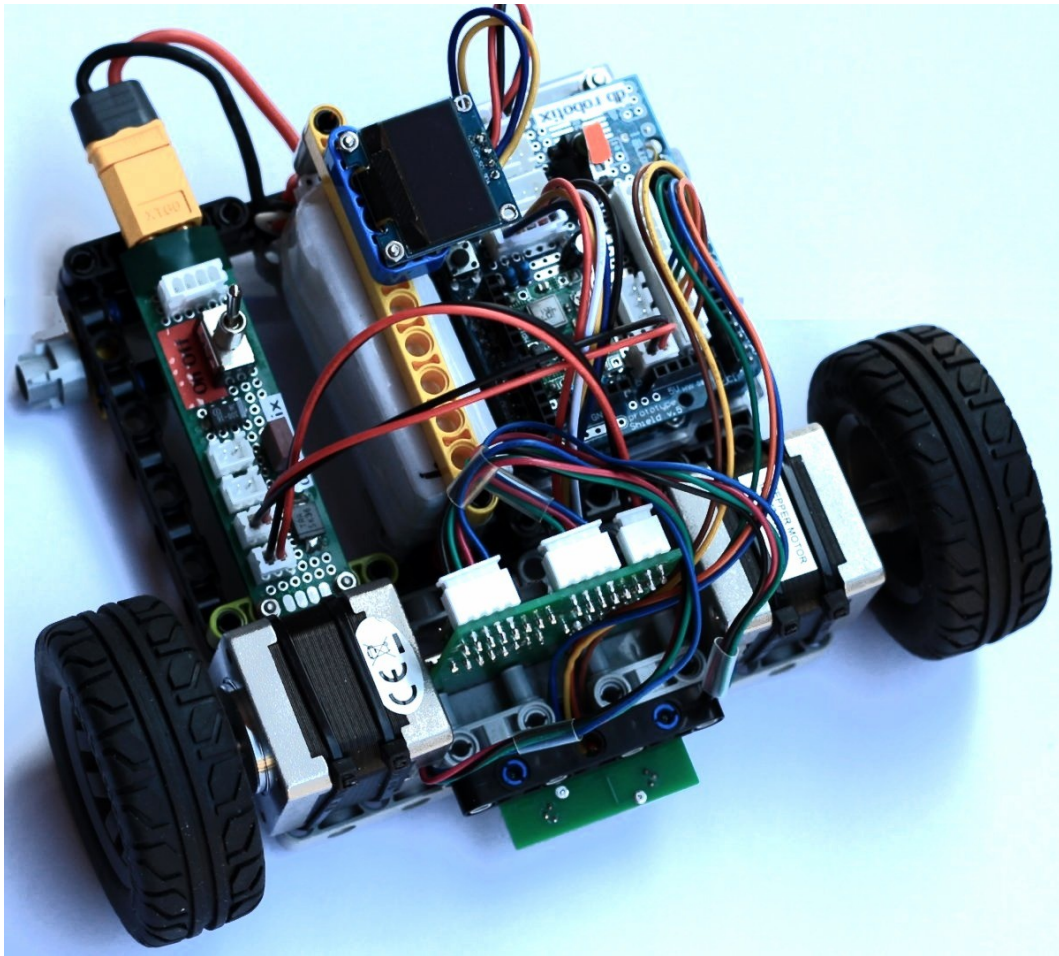


db robotix

Das innovative System für kreative Spiel-Roboter



Mit Hilfe dieser Anleitung wirst du Schritt für Schritt lernen, interessante Roboter-Konstruktionen zu bauen und zu programmieren.

Um einfach aufzubauende und trotzdem variationsreiche Konstruktionen zu ermöglichen, lassen sich alle Bausteine des Systems mit Lego-Technic-Komponenten (meist Lochstangen) verbinden. Allerdings nicht mit den Lego-Motoren und -Sensoren, aber dafür haben wir ja eigene Bausteine von db robotix, die zudem noch besser funktionieren.

In den folgenden Kapiteln lernst du, wie du sie einsetzen und programmieren kannst. Wir fangen mit einfachen Techniken an und im Kapitel „Fortgeschrittene Techniken“ kommen auch die „Profis“ unter euch richtig zum Zuge. Zusätzliche Infos befinden sich in grauen Kästen. Im Anhang sind noch weitere Informationen zusammen gefasst, die auch eher die Fortgeschrittenen interessieren.

Wichtige Sicherheitshinweise sind gelb hinterlegt.

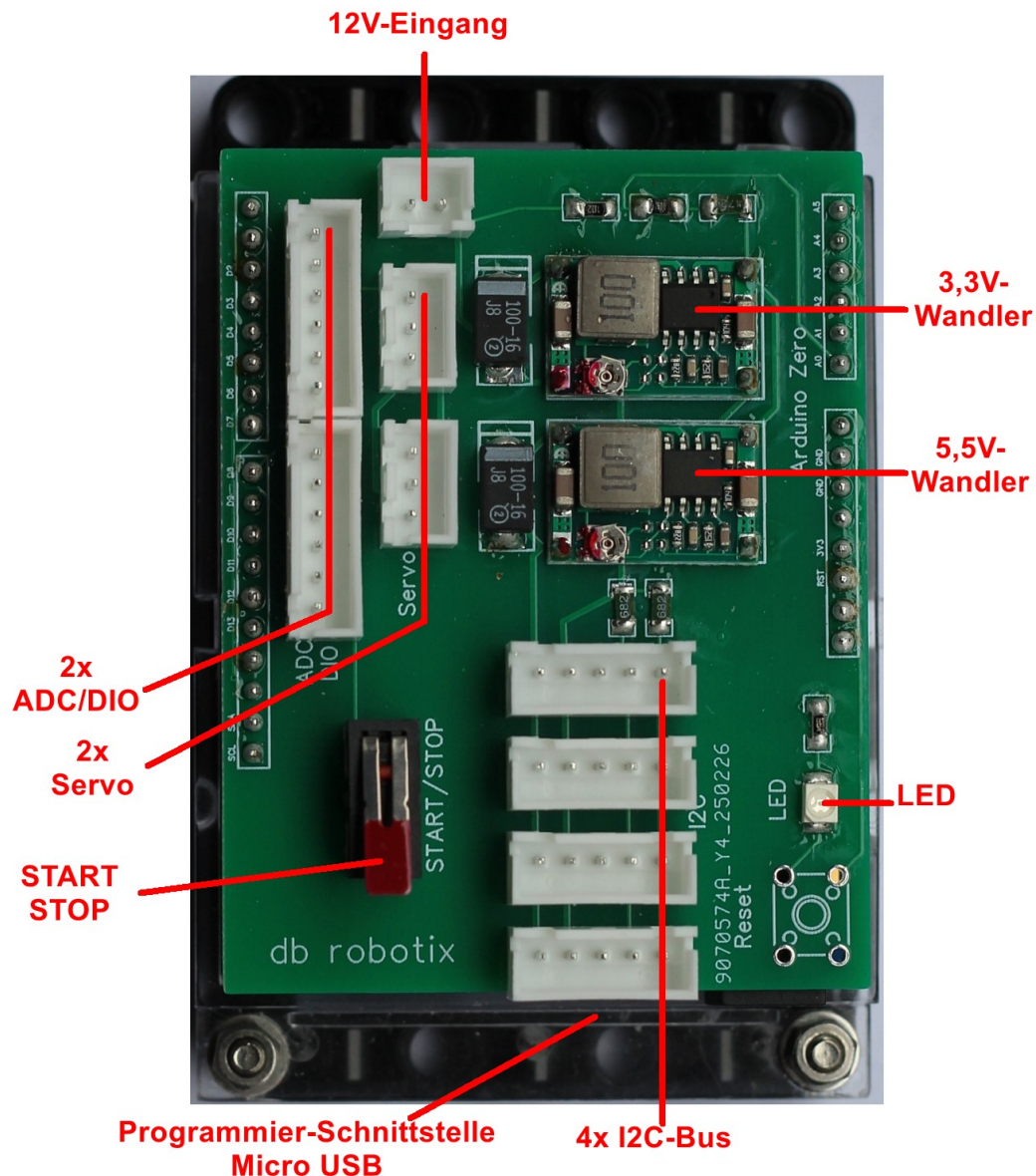
Kontakt: db-robotix@web.de

Erste Schritte

Die Steuerzentrale

Unser zukünftiger Roboter braucht einen Meister, der die verschiedenen Bestandteile bedient und ihnen sagt, was sie im einzelnen zu tun haben. Hierzu dient die Steuerzentrale oder auf englisch der Master Controller.

Schaust du von oben darauf, siehst du folgenden Aufbau:



Die einzelnen Teile und Funktionen werden später noch genau beschrieben.

Von der Seite betrachtet, siehst du zwei Ebenen. Die untere Ebene (im Bild nicht sichtbar) ist ein so genannter Arduino Zero, eine leistungsfähige Variante der weit verbreiteten Arduino-Microcomputer. Und Zero bedeutet keineswegs, dass er nichts kann. Im Gegenteil: Es handelt sich wirklich um eine Art Computer, der programmiert werden kann.

Trenne die beiden Ebenen möglichst nicht, weil eventuell die Kontaktstifte, die die beiden verbinden, verbogen werden. Du brauchst auch immer genau diese Kombination. Nur falls mal was kaputt ist, müssen sie getrennt werden.

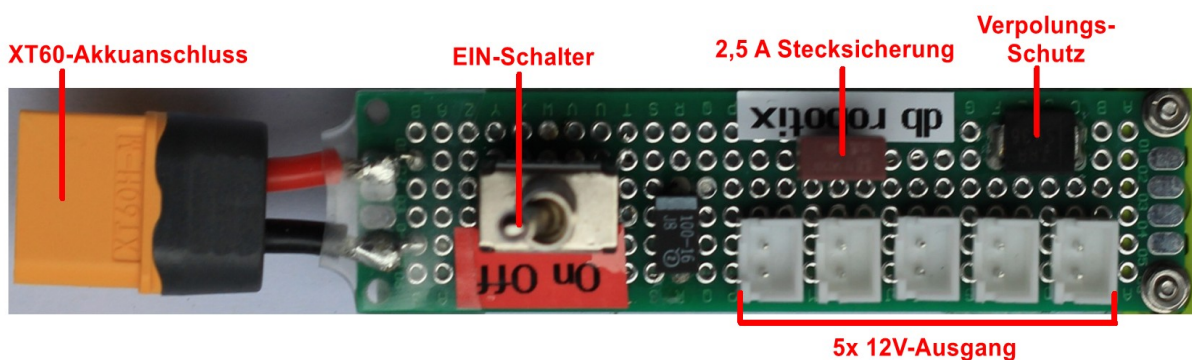
Du hast sicherlich beobachtet, dass sich unter dem Arduino noch eine durchsichtige Bodenplatte befindet. An die beiden Löcher (im Bild am unteren Rand) kann mit Hilfe zweier Gewindeschrauben und Muttern ein Lego-Technik-Rahmen der Größe 88 mm x 56 mm montiert werden. Somit lässt sich eine stabile Konstruktion aufbauen.

Woher kommt die Energie?

Jedes elektronische Teil und vor allem die später hinzu gefügten Motoren benötigen für ihre Funktion elektrische Energie. Da es für den Roboter unpraktisch wäre, ein Stromkabel hinter sich her zu ziehen, benutzen wir einen Akku, der auf dem Roboter mittransportiert wird. Hierzu ist ein so genannter Lithium-Polymer-Akku (abgekürzt LiPo), wie er auch in Notebooks und Handys eingesetzt wird, vorteilhaft. Er liefert viel Energie bei wenig Gewicht. Mit unserem 100g-Akku könnte der Roboter mehr als eine Stunde ununterbrochen fahren!



Also schließen wir den Akku an die Steuerzentrale an. Doch warte: Wir nehmen noch eine Schutzschaltung hinzu, die verhindert, dass der Akku falsch herum angeschlossen oder überlastet werden kann. Außerdem verfügt die Schutzschaltung über einen Schalter, mit dem man alles ein- und wieder ausschalten kann, sowie mehrere Ausgangs-Anschlüsse für die vielen Komponenten, die der Roboter braucht:



Der verwendete Akku hat eine „Nennspannung“ von 11,1 Volt. Voll geladen hat er sogar eine Spannung von etwa 12,5 Volt. Daher sprechen wir im Folgenden immer vom 12V-Anschluss.

Wichtiger Hinweis: Du hast sicherlich schon mal davon gehört, dass in seltenen Fällen ein Handy, E-Bike oder auch Elektroauto Feuer fängt, weil der Akku defekt war. Das kann deinem Roboter im Normalfall nicht passieren, wenn du folgende Regeln beachtest:

- Gehe niemals mit einem metallischen Gegenstand direkt an die gelben Anschlüsse oder den weißen Stecker des Akkus.
- Achte darauf, dass die Akku-Schutzschaltung nicht eine andere Elektronik berührt.
- Schalte deinen Roboter aus, wenn du ihn einige Zeit nicht benutzt, denn der Akku mag es nicht, wenn er zu weit entladen wird.

- Benutze zum Aufladen ausschließlich das vorgesehene Ladegerät und nur unter deiner Beaufsichtigung (nicht einfach über Nacht angeschlossen lassen).
- Wenn du feststellst, dass das Kabel oder die Kunststoff-Umhüllung beschädigt ist oder der Akku heiß wird, benutze den Akku nicht weiter und verständige sofort die Aufsichtsperson.
- Bringe den Akku nicht mit Flüssigkeiten in Berührung (Getränke gehören nicht auf den Arbeitstisch!)

Jetzt kann es endlich weiter gehen:

Stecke den gelben Stecker des Akkus in die passende Buchse der Akku-Schutzschaltung. Kippe den Schalter auf „Off“, und verbinde mit einem schwarz-roten Kabel den Master Controller mit einem Ausgang der Schutzschaltung (welchen der fünf Ausgänge du benutzt, spielt keine Rolle).

Wenn du jetzt den Schalter betätigst (auf „On“), werden auf den Controller-Platten verschiedene Leuchtdioden aufleuchten.

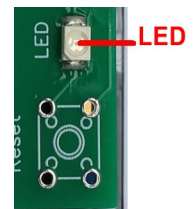
Sonst passiert nichts, denn wir haben ja noch kein Programm. Also vorsichtshalber wieder ausschalten!

Unser erstes Programm: Ein Licht geht auf

Auf dem Master Controller befindet sich eine Leuchtdiode. Sie kann vom Arduino angesteuert werden und leuchtet dann gelb. Damit das funktioniert, muss in das Programm eine so genannte „Bibliothek“ (englisch library) eingebunden werden.

Zur Programmierung benutzen wir die Arduino IDE. IDE steht für Integrated Development Environment, zu deutsch: integrierte Entwicklungsumgebung.

Lass diese IDE von einem erfahrenen Computernutzer auf deinem Notebook oder Computer installieren. Wie das gemacht wird, ist im Anhang beschrieben.



Zuerst Allgemeines zu Bibliotheken:

Spezielle Funktionen, die der Roboter benötigt, befinden sich in einer Bibliothek, so wie in einer echten Bücherei verschiedene Bücher ausgeliehen werden können. In jedem dieser „Bücher“ sind die Funktionen beschrieben und definiert, die du in das Programm für einen Baustein einsetzen kannst. So gibt es zum Beispiel Bücher über die verschiedenen Motoren, für den Farbsensor und eben für die Leuchtdiode (LED).

Wir brauchen nun die Bibliothek `anadigMaster` (steht für analog und digital; die LED wird „digital“ angesteuert.) In dieser Bibliothek sind mehrere „Klassen“ festgelegt, z.B. eine mit dem Namen `Led`.

Folge diesen Schritten:

1. Starte die Arduino IDE.
2. Erzeuge ein neues Projekt mit File – New Sketch (oder Strg-N).
3. Gib dem Projekt mittels File – Save (oder Strg-S) einen beliebigen neuen Namen, z.B. Robotertest. Verwende in dem Namen keine Leerzeichen!
4. Es öffnet sich die Programmieroberfläche der Datei „Robotertest.ino“ mit den Grundfunktionen `setup` und `loop`.
5. Ganz am Anfang des Programms schreibe: `#include <anadigMaster.h>`
Achte auf die Sonderzeichen und korrekte Groß-/Kleinschreibung!
Damit steht nun die Bibliothek `anadigMaster` zur Verfügung.

6. Schreibe in die nächste Zeile: `Led LED;` (Semikolon am Ende nicht vergessen!)
Warum nun zweimal der gleiche Name? Es ist nicht der gleiche Name, weil das Programm zwischen Groß- und Kleinschreibung unterscheidet. `Led` ist der festgelegte Name der Klasse aus der Bibliothek, während `LED` der Name für das „Objekt“ in deinem Programm ist (er könnte auch anderes lauten). Um die Bibliothek nutzen zu können, muss ein „Kind“ der benötigten Klasse definiert werden. Und dieses erhält hier den Namen `LED`.
7. Damit nun die `LED` leuchten darf, musst du in den vorbereiteten Programmteil „loop“ den folgenden Befehl einbauen:

```
void loop() {
  LED.on();
}
```
8. Schalte den Akku ein und lade nun dein Programm in den Arduino. Lass dir von jemandem, der Erfahrung damit hat, zeigen, wie es geht, oder schau im Anhang „Programmierung des Arduino Zero“ nach.
9. Die `LED` leuchtet nun gelb.

Falls es nicht funktioniert, überprüfe nochmals dein Programm:

```
#include <anadigMaster.h>
Led LED;
void setup() {
}
void loop() {
  LED.on();
}
```

Hinweis: Leerzeichen und leere Zeilen dürfen außerhalb von Wörtern an beliebigen Stellen hinzu gefügt werden.

Zwischen `LED` und `on()` befindet sich ein Punkt! Links davon steht der Objektname und rechts die gewünschte Funktion zum Einschalten.

Für die Programmier-„Experten“ unter euch: Die von der Arduino-IDE verwendete Programmiersprache ist „C“. Diese wird ergänzt durch Elemente aus „C++“ zur so genannten „Objektorientierten Programmierung“ (OOP). Von dieser stammen auch die Begriffe „Klasse“, „Objekt“ und „Vererbung“ (hier anschaulicher als „Kind“ bezeichnet).

In Wartestellung

Damit es interessanter wird, möchten wir die `LED` nun zum Blinken bringen. Damit unsere Augen das Ein und Aus der `LED` verfolgen können, muss etwas Zeit zwischen diesen Vorgängen vergehen. Eine Möglichkeit, von der wir immer wieder Gebrauch machen werden, ist der Befehl „delay“ (Verzögerung). Wir müssen ihm noch mitteilen, wie lang diese Verzögerung sein soll. In Klammern dahinter wird die Zeit in Millisekunden (1/1000 Sekunde) angegeben, z.B. `delay(500);`

Das Programm wartet dann an dieser Stelle eine halbe Sekunde.

Danach soll die `LED` wieder ausgeschaltet werden: `LED.off();`

Wie der Funktionsteil „loop“ (Schleife) schon erwarten lässt, wird dieser automatisch immer wiederholt.

Schreiben wir in dieses Loop-Programm:

```
void loop() {  
  LED.on();  
  delay(500);  
  LED.off();  
  delay(500);  
}
```

und schalten das System ein, blinkt die LED im Sekundenrhythmus solange, bis die Spannungsversorgung (Akku) wieder ausgeschaltet wird.

Hast du an die Semikolons (;) am Ende jeden Befehls gedacht? Ansonsten wird sich die Arduino IDE bei der Überprüfung des Programms (Compilieren) beschweren.

Zur Übung kannst du mal probieren, die LED immer wieder 3-mal blinken zu lassen und dann 3 Sekunden bis zum nächsten Mal zu warten.

Aktion auf Tastendruck

Auf dem Master Controller befindet sich ein größerer Taster, der mit einer roten Markierung gekennzeichnet ist. Dieser kann benutzt werden, um den Ablauf des Programms zu beeinflussen, z.B. das Programm an einer bestimmten Stelle anzuhalten, bis der Taster betätigt wird.

In der schon vorgestellten Bibliothek `analogMaster` gibt es eine weitere Klasse namens `Button` (Knopf). Auch hiermit muss ein Kind benannt werden, das den Knopf „bewacht“, z.B.:

```
Button knopf;
```

Es gibt in der Klasse ein Programm `wait`, das solange wartet, bis der Taster gedrückt und wieder losgelassen wird:

```
knopf.wait();
```

Damit können wir ein Programm schreiben, dass erst dann das Blinken der LED beginnt, wenn der Taster gedrückt wurde. Wir nehmen dazu den Funktionsteil „`setup`“. Dieser wird nur einmal ausgeführt, wenn der Arduino durch Einschalten (oder durch den „Reset“-Taster) neu startet.

```
void setup() {  
  knopf.wait();  
}
```

Wird in der Klammer von `wait` eine Zahl `z` eingetragen, wartet das Programm noch weitere `z` Millisekunden. Das wird später nützlich sein, wenn der Roboter erst losfahren soll, nachdem du sicher den Finger vom Taster entfernt hast. Du kannst das testen, indem du `knopf.wait(2000);` programmierst: Erst zwei Sekunden, nachdem du den Taster loslässt, blinkt die LED.

Was will uns der Roboter mitteilen?

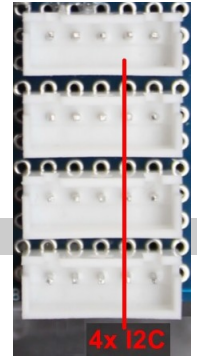
Der Roboter kann auch mit uns kommunizieren, indem er Texte oder Zahlen auf einer kleinen Anzeige (englisch `display`) ausgibt. Die Anzeige muss mit einem der „I2C“-Anschlüsse verbunden werden. Die Kommunikation, welche Texte dort erscheinen sollen, geschieht nämlich über den so genannten I2C-Bus, der in der Microcomputer-Technik weit verbreitet ist. Wir werden diesen „Bus“ (eine Verbindung mit mehreren Kabeln) wieder treffen beim Farbsensor und sogar bei der Motorsteuerung.



Das Anschlusskabel der Anzeige wird einfach in einen der vier I2C-Anschlüsse des Master Controllers eingesteckt. Das geht nur in eine Richtung (beachte die beiden Aussparungen in den 5-poligen Buchsen).

Über den Bus wird auch Energie zum angeschlossenen Gerät geführt. Eine Verbindung zur 12V-Versorgung (siehe Akku-Schutzschaltung) wird nicht benötigt.

Die Spannung auf dem Bus beträgt außerdem nur 3,3 Volt.



Für alle Geräte, die an den I2C-Bus angeschlossen werden, existiert eine eigene Bibliothek `i2cMaster`. Wie wir es schon mit der `anadigMaster`-Bibliothek getan haben: `#include <i2cMaster.h>` (am besten ganz oben im Programm)

Die nun benötigte Klasse in dieser Bibliothek heißt `Display`. Wir benennen danach ein Kind dieser Klasse: `Display anzeige;`

Den Namen „anzeige“ des Kindes kannst du (fast) beliebig wählen. Nicht erlaubt ist natürlich „Display“, weil die Klasse schon so heißt. „display“ (kleingeschrieben) wäre dagegen möglich. Namen von Standard-Unterprogrammen, wie „delay“ sind auch nicht zulässig. (Wer möchte sein Kind auch schon delay nennen? :-)

Für alles, was den I2C-Bus erfordert, musst du in den `setup`-Teil des Arduino-Programms diesen Befehl einsetzen: `Wire.begin();`

Im folgenden Testprogramm lernen wir gleich mehrere Unterprogramme dieser Klasse kennen, die alle mit „anzeige.“ beginnen:

| | |
|---|--|
| <code>anzeige.start();</code> | die Anzeige wird aktiviert |
| <code>anzeige.clear();</code> | es wird alles gelöscht, was eventuell vorher auf der Anzeige stand |
| <code>anzeige.println("Hallo");</code> | die Anzeige begrüßt euch mit diesem Wort |
| <code>anzeige.print(„Ich bin „);</code> | dieses kommt in die nächsten Zeile und direkt dahinter |
| <code>anzeige.println(12);</code> | diese Zahl |
| <code>anzeige.setRow(4);</code> | Weiter geht es in der 4. Zeile mit |
| <code>anzeige.println("Tschuess");</code> | der Verabschiedung |

Einige Erklärungen:

- Symbolisch wird auf der Anzeige etwas „gedruckt“ (`print`).
- Bei Verwendung von `println` wird das Nachfolgende in der nächsten Zeile ausgegeben (steht für `print line`).
- Auszugebende Texte müssen in Anführungszeichen stehen (`"Hallo"`).
- Sonderbuchstaben, dazu gehören auch ä,ö,ü,ß, lassen sich nicht auf das Display bringen.
- Zahlen (und später Variablen) stehen nicht in Anführungszeichen.
- Das Display ist so eingestellt, dass 4 Zeilen „bedruckt“ werden können. Durch den Befehl `setRow` kann die Zeile festgelegt werden.
- Wurde zuletzt in der 4. Zeile `println` eingesetzt, werden weitere Ausgaben ignoriert, da es keine 5. Zeile gibt. In diesem Fall muss dann für weitere Ausgaben ein `clear` oder `setRow` erfolgen.
- `anzeige.start();` ist immer notwendig, wenn sie benutzt wird. Das gilt auch für alle nachfolgenden Programme.

Farbe und Helligkeit erkennen

Schon kommt das nächste I2C-Gerät zum Einsatz. Mit Hilfe des Farbsensors A lassen sich die Farbe und die Helligkeit eines Gegenstands, der sich vor dem Farbsensor befindet, komfortabel messen.

Nach dem Anschluss an einer weiteren I2C-Buchse des Master Controllers leuchten die beiden weißen LEDs sehr hell. Somit ist der Sensor nicht auf Umgebungslicht oder eine externe Lichtquelle angewiesen.

Nach Benennung eines Kindes der Klasse ColorSensorA aus der i2cMaster-Bibliothek, z.B. durch

```
ColorSensorA farbsensor;
```

lassen sich mit dem Kindsnamen farbsensor verschiedene Messfunktionen nutzen, z.B.:

```
farbsensor.start();  
farbsensor.getRGB();  
farbsensor.color();  
farbsensor.intens();
```

start() und getRGB() leiten die Messung ein.

Die Funktion color() liefert eine Zahl als Code für bestimmte Farben zurück:

1 = rot ; 2 = gelb ; 3 = grün ; 4 = blau ; 5 = weiß; 0 = keine Farbe erkannt (z.B. schwarz)

Die Funktion intens() liefert eine Zahl als Maß für die gemessene Helligkeit (Intensität) des reflektierten Lichtes. Die Werte bewegen sich im Bereich von 0 bis etwa 2500.

Zur Ausgabe der Ergebnisse wird die Anzeige benutzt:

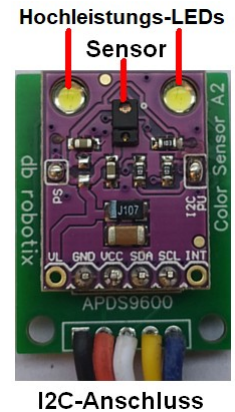
```
anzeige.println(farbsensor.color());  
anzeige.println(farbsensor.intens());
```

Baue diese innerhalb von setup() ein, wenn eine einmalige Messung erfolgen soll oder innerhalb von loop(), wenn die Messung fortlaufend erfolgen soll.

Im letzteren Fall sollte zusätzlich ein anzeige.clear(); und delay(...); im loop stehen.

Baue jetzt dein Messprogramm zusammen! Du kannst die Ausgaben auch noch mit zusätzlichen Texten versehen, damit man weiß, was die Zahlen bedeuten. Falls es auf Anhieb noch nicht funktioniert, keine Panik: Vor dem Anhang gebe ich eine Lösung dazu.

Weitere Möglichkeiten des Farbsensors findest du in der Bibliotheken-Befehlsreferenz im Anhang.



Abfragen und Schleifen

In diesem Abschnitt beschäftigen wir uns mit einigen grundlegenden Programmier-Techniken und wenden sie dann auf den Farbsensor an.

Soll ein Programmabschnitt nur dann ausgeführt werden, wenn eine bestimmte Bedingung erfüllt ist (z.B. eine bestimmte Farbe erkannt wurde), wird eine if-Abfrage helfen (if = wenn):

```
if (farbsensor.color() == 1) anzeige.println(„rot“);
```


Das doppelte Gleichheitszeichen wird für die Gleichheit zweier Zahlen verwendet.

Soll ein anderer Programmschritt nur dann folgen, wenn das nicht der Fall ist, wird ein else-Zweig hinzugefügt (else = anderenfalls):

```
else anzeige.println(„nicht rot“);
```

if und else lassen sich mehrfach kombinieren:

```
if (farbsensor.color() == 1) anzeige.println(„rot“);
```

```
else if (farbsensor.color() == 2) anzeige.println(„gelb“);
```

```
else if (farbsensor.color() == 3) anzeige.println(„gruen“);
```

```
else anzeige.println(„andere Farbe“);
```

Eine Schleife ist ein Programmabschnitt, der mehrfach in gleicher Weise ausgeführt wird.

Soll diese Schleife eine bestimmte Anzahl mal durchlaufen werden, nimmt man den for-Befehl, z.B. 10 Durchläufe mit:

```
for (int i = 0; i < 10; i++) {Programmblock}
```

Erläuterungen:

- i ist eine so genannte Variable (siehe späteres Kapitel) mit ganzen Zahlen (int)
- Diese Variable wird zunächst auf den Wert 0 gesetzt (i = 0).
- Der Programmblock wird solange durchlaufen, wie i kleiner als 10 ist.
- Damit der Wert 10 überhaupt erreicht wird, muss i in der Schleife verändert werden.
- Das geschieht durch Hochzählen am Ende der Schleife mit Hilfe von i++ .
- Beachte, dass die drei Angaben in der for-Klammer mit Semikolons getrennt werden.
- Ein Programmblock mit mehreren Befehlen wird mit geschweiften Klammern { } umfasst.
- Nur wenn ein einziger Befehl innerhalb der Schleife notwendig ist, kann { } entfallen.

Mit dem Befehl for (; ;) wird die Schleife unendlich oft ausgeführt. Das Programm bleibt dadurch stehen und kann nur durch den Reset-Knopf oder Neu-Einschalten wieder gestartet werden. Sollte nur in Ausnahmefällen eingesetzt werden.

Schreibe ein kleines Programm, das mit dem Farbsensor 20-mal im Sekundentakt die Helligkeit misst und das Ergebnis in die erste Zeile der Anzeige schreibt.

Eine andere Schleifenart benutzt das Schlüsselwort „while“ (= während). Dahinter muss eine Bedingung in Klammern gesetzt werden. Beispiel:

```
while (farbsensor.color() != 5) { delay(10); }
```

```
LED.on();
```

Das Zeichen != wird für die Prüfung auf Ungleichheit verwendet. Also: während die gemessene Farbe nicht 5 (weiß) ist, wird die Schleife mit einer Verzögerung von 10 ms immer wieder durchlaufen. Erst wenn du ein weißes Blatt Papier vor den Sensor hältst, endet die Schleife und die gelbe LED leuchtet auf.

Die geschweiften Klammern sind hier nicht unbedingt erforderlich (weil innerhalb nur ein Befehl existiert), werden aber häufig der Übersichtlichkeit halber dennoch gesetzt.

Roboter in Action

Jetzt soll der Roboter in Bewegung kommen. Dazu brauchen wir Motoren. Sehr zuverlässig arbeiten so genannte Schrittmotoren. Während man an einen normalen Motor einfach eine Spannung anlegt und er läuft dann kontinuierlich,

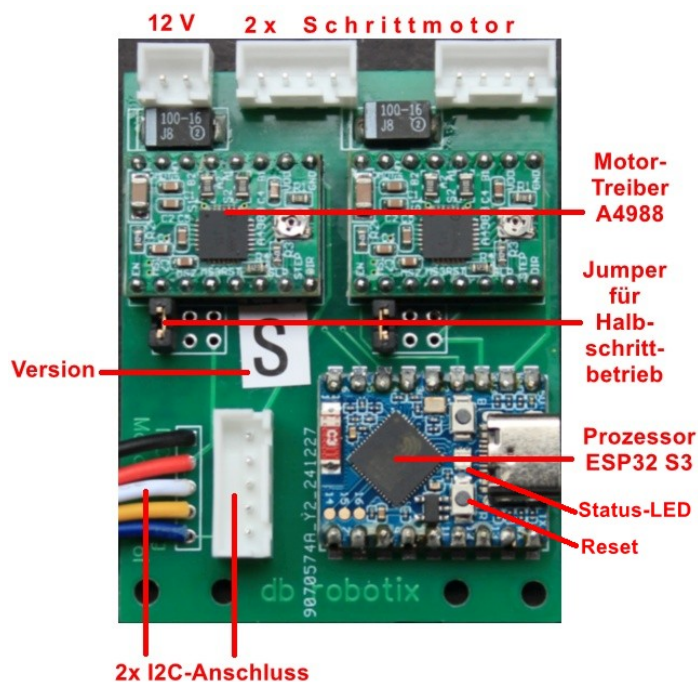


braucht ein Schrittmotor eine spezielle Ansteuerung. Für jeden kleinen Drehwinkel – einen „Schritt“ (meist 0.9 Grad oder 1.8 Grad) muss ein elektrisches Signal an ihn übertragen werden. Dabei dreht er exakt diese Schritte in eine bestimmte Richtung weiter, falls er nicht blockiert ist. Selbst eine schwankende Akkuspannung ändert daran nichts. Beim „normalen“ Motor weiß man dagegen ohne zusätzliche Einbauten nicht, wie weit er gedreht hat. Der Schrittmotor ist ihm hinsichtlich der Präzision weit überlegen.

Um dem Motor die Schritte vorzugeben, brauchen wir eine zusätzliche Elektronik (Bild rechts), den Motor Controller oder die Motorsteuerung, die wieder über einen I2C-Bus (im Bild links unten) verfügt. Es gibt am Motor-Controller Anschlüsse für zwei Motoren (im Bild oben). Für den Betrieb der Motoren ist die 12V-Spannung vom Akku-Schutz notwendig (oben links), zugeführt über ein 2-adriges Verbindungskabel (schwarz – rot).

Achte darauf, dass dieses Modul nicht die Motoren oder andere Module berührt. Verschraube dazu die Befestigungslöcher (im Bild unten) mit einer Lochstange.

Um zwei Fahrmotoren (linkes Rad und rechtes Rad) zu betreiben, brauchst du die Version „S“ der Motorsteuerung, erkennbar am Aufkleber. Damit lassen sich zwei Schrittmotoren gleichmäßig (synchron) ansteuern, damit der Roboter exakt geradeaus fährt.



Als Bibliothek dient wieder i2cMaster mit der neuen Klasse Drivetrain (zu deutsch: Antriebsstrang). Bei der Definition des Objekts (nennen wir es Robert) musst du in Klammern noch den Wert 4 angeben. Das ist die I2C-Adresse der Motorsteuerung S.

```
Drivetrain Robert(4);
```

Bevor du Robert los schickst, muss er noch wissen, wie weit er fahren soll, mit welcher Geschwindigkeit (speed) und ob er lenken (steering) soll, um eine Kurve zu fahren (siehe nächsten Abschnitt). Die Fahrstrecke muss in Schritte der Motoren umgerechnet werden, denn Robert weiß noch nicht, wie viele Schritte er für ein gewisse Strecke braucht. Das hängt vom Durchmesser der Reifen ab. Dreht sich das Rad genau um eine Umdrehung, was mit der Standardeinstellung des Motor Controllers genau 400 Schritte sind, entspricht die zurückgelegte Strecke der Länge der Lauffläche des Reifens, also dessen Umfang. Aus der Geometrie des Kreises weißt du sicherlich, dass diese Formel gilt: $\text{Umfang} = \pi \cdot \text{Durchmesser}$. π (pi) ist die Kreiszahl 3,1415.

Der Fischertechnik-Reifen hat einen Durchmesser von etwa 64 mm und damit einen Umfang von etwa 200 mm. Also bei 400 Schritten fährt er 200 mm weit.

Damit Robert endlich startet, sagst du einfach „go“.

Ein Programm sieht dann beispielsweise so aus:

```
Robert.setAccelerations();
Robert.setSpeed(50);
Robert.setSteering(0);
```

```
Robert.setTargetSteps(400);  
Robert.go();  
delay(2000);  
Robert.coast();
```

Erklärung der Befehle:

- Mit setAccelerations() werden für die Beschleunigung (engl. acceleration) und das Abbremsen Standardwerte festgelegt, die für die meisten Roboterbewegungen optimal sind.
- Mit setSpeed(50) wird die Geschwindigkeit des Roboters auf 50 cm/s festgelegt.
- Mit setSteering(0) fährt der Roboter ohne Lenkung, also geradeaus.
- Mit setTargetSteps(400) wird das Ziel (engl. target) der Fahrt definiert (400 Schritte = 1 Umdrehung = 200 mm Strecke).
- Durch go() startet der Roboter seine Fahrt.
- delay(2000): Damit er auch Gelegenheit hat, diese Fahrt zu Ende zu führen, warten wir vorsichtshalber 2 Sekunden ab, bevor das Programm endet.
- Robert.coast() bewirkt, dass am Ende der Strom für die Motoren abgeschaltet und so der Akku geschont wird.

Jetzt musst du nur noch mit Lego Technic ein Roboter-Grundmodell mit zwei Schrittmotoren (am besten vom Typ NEMA 14) und montierten Reifen bauen. Achte darauf, dass die Reifen parallel zueinander stehen.

Schließe die Schrittmotoren an den Motor Controller an (unbedingt im ausgeschalteten Zustand; sonst könnte die Elektronik zerstört werden!), verbinde den 12V-Anschluss über ein Kabel mit der Akku-Schutzschaltung, stecke die I2C-Verbindung auf den Master Controller, lade das Programm in den Arduino und schalte dann ein. Los geht's!

Falls er nicht vorwärts, sondern rückwärts fährt, vertausche einfach die beiden Motor-Stecker. Aber vorher ausschalten!

Soll der Roboter bewusst rückwärts fahren, setze in setSpeed einen negativen Wert ein, z.B.

```
Robert.setSpeed(-50);
```

Eine Kurvenfahrt

Vielleicht möchte Robert sich nicht immer nur geradeaus bewegen, sondern auch mal eine Kurve fahren. Da er keine Lenkachse besitzt, wie bei einem Auto, müssen wir das anders bewerkstelligen: Dreht das linke Rad schneller als das rechte, fährt er eine Rechtskurve. Und dreht das rechte Rad schneller, fährt er links herum. Diese Art Lenkung (englisch: steering) erreichen wir durch den Programmbefehl setSteering(...), der an den Motor Controller gesendet wird.

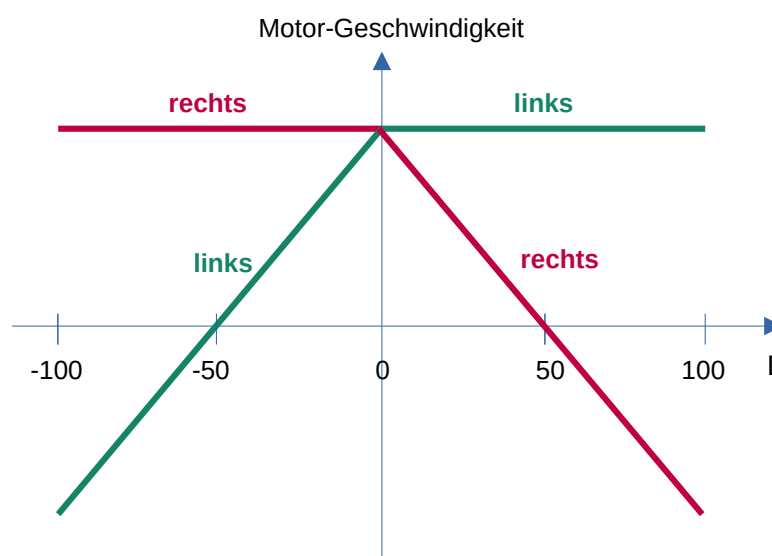
In der Klammer steht der Lenkwert L, der – wie bei anderen Robotersystemen auch – wie folgt definiert ist:

- $L = 0$: Roboter fährt geradeaus
- $L > 0$: Roboter fährt eine Rechtskurve
- $L < 0$: Roboter fährt eine Linkskurve
- $L = 50$: Das rechte Rad bleibt stehen und der Roboter dreht sich im Uhrzeigersinn
- $L = -50$: Das linke Rad bleibt stehen und der Roboter dreht sich gegen den Uhrzeigersinn

- $L = 100$: Das linke Rad dreht nach hinten und das rechte Rad nach vorn, so dass sich der Roboter um den Achsenmittelpunkt im Uhrzeigersinn dreht.
- $L = -100$: Das linke Rad dreht nach vorn und das rechte Rad nach hinten, so dass sich der Roboter um den Achsenmittelpunkt gegen den Uhrzeigersinn dreht.
- Zwischenwerte für L führen dazu, dass sich die Räder unterschiedlich schnell drehen und damit eine Kurve gefahren wird.

Probiere es direkt mal aus, indem du in dem bisherigen Programm `Robert.setSteering(0)` durch `Robert.setSteering(20)` oder andere Lenkwerte ersetzt.

Die eingestellten Werte für Geschwindigkeit und Rad-Drehwinkel sind immer für das schnellere Rad maßgeblich, das andere Rad wird nach folgender Grafik langsamer bewegt und dreht damit weniger weit:



Für die Bewegungsrichtung der Räder gilt zusammengefasst:

| Lenkwert L | Richtung linkes Rad | Richtung rechtes Rad | Drehrichtung Fahrzeug |
|--------------|---------------------|----------------------|-----------------------|
| -100 ... -51 | Rückwärts | Vorwärts | Gegen Uhrzeigersinn |
| - 50 | Steht | Vorwärts | Gegen Uhrzeigersinn |
| -49 ... -1 | Vorwärts | Vorwärts | Gegen Uhrzeigersinn |
| 0 | Vorwärts | Vorwärts | Geradeaus |
| 1 ... +49 | Vorwärts | Vorwärts | Im Uhrzeigersinn |
| +50 | Vorwärts | Steht | Im Uhrzeigersinn |
| +51 ... +100 | Vorwärts | Rückwärts | Im Uhrzeigersinn |

Variablen

Eine Variable ist wie eine Schublade, in der z.B. Murmeln aufbewahrt werden: Du kannst eine bestimmte Anzahl Murmeln dort lagern, nachzählen wie viele sich dort befinden oder deren Anzahl

verringern bzw. vergrößern. Außerdem bekommt die Schublade eine Beschriftung, das ist der Variablenname.

In den Programmiersprachen wird eine Variable, in der ganze Zahlen gespeichert werden, integer (abgekürzt int) genannt und wird wie folgt deklariert:

```
int murmeln;
```

Sollen am Programmanfang schon 100 Murmeln enthalten sein, schreibst du:

```
int murmeln = 100;
```

Hiermit werden 20 Murmeln hinzugefügt:

```
murmeln = murmeln + 20;
```

Für die Erhöhung der Anzahl um eins gibt es eine Abkürzung

(kennen wir schon von der for-Schleife):

```
murmeln++;
```

Um die Anzahl zu ermitteln, benutzt du einfach den Variablennamen:

```
anzeige.println(murmeln);
```

Mit Variablen lässt sich auch rechnen:

```
int a = 3, b = 5, c = 10;
```

$a+b+c$ ergibt dann folgerichtig 18

$a*b + c$ ergibt 25

$a + b*c$ ergibt 53 (auch hier gilt die Regel: Punktrechnung vor Strichrechnung)

$(a + b) * c$ ergibt 80

$c - b$ ergibt 5

c / b ergibt 2 (das Zeichen / steht für eine Division)

b / a ergibt jedoch 1 und nicht 1.6666, denn es wird bisher nur mit Ganzzahlen gerechnet.

Die integer-Division ist eine ganzzahlige Division, der Rest wird abgeschnitten.

$b \% a$ ergibt 2 (das Zeichen % - modulo genannt - steht für den Rest der ganzzahligen Division)

Mit den folgenden Programmzeilen kannst du die Division mit Restangabe anzeigen lassen:

```
int a = 3, b = 5;
```

```
anzeige.print(b);
```

```
anzeige.print(" / ");
```

```
anzeige.print(a);
```

```
anzeige.print(" = ");
```

```
anzeige.print(b / a);
```

```
anzeige.print(" Rest ");
```

```
anzeige.println(b % a);
```

Bei der Division musst du natürlich aufpassen, dass der Divisor (oder Nenner eines Bruches) nicht Null wird. In diesem Fall würde das Programm sich selber beenden. Achte also sorgsam darauf, dass die Divisor-Variable niemals 0 werden kann!

Bei der Kombination von Division und Multiplikation kann man böse Überraschungen erleben.

Weißt du, was $a / b * c$ ergibt? Möglicherweise hat du 6 errechnet, was auch richtig wäre, wenn du mit gebrochenen (rationalen) Zahlen zwischenrechnest. Das Ergebnis ist aber seltsamerweise 0 ! Warum? Die Berechnung des Ausdrucks wird von links nach rechts ausgeführt, also zunächst $a / b = 3 / 5 = 0$ ($3 / 5$ ergibt 0 Rest 3). Die nachfolgende Multiplikation mit 10 ändert das Zwischenergebnis nicht mehr.

Um die Rechnung wie gewollt „richtig“ durchzuführen, schreibe stattdessen: $a * c / b$. Dann erhältst du wirklich 6. (Regel: zuerst die Multiplikationen und dann die Divisionen)

Aber auch bei der Multiplikation oder Addition gibt es Fallstricke, wenn das Ergebnis zu groß wird, denn die Schublade kann nur eine bestimmte Anzahl Murmeln aufnehmen.

Rechnest du z.B.:

```
int murmeln = 32767;  
murmeln = murmeln + 3;  
anzeige.println(murmeln);
```

siehst du auf der Anzeige das seltsame Ergebnis -32766.

Die Ursache liegt darin, dass die Marmor-Schublade höchstens 32767 Kugeln aufnehmen kann.

Beim Überschreiten des erlaubten Wertebereichs (Fachbegriff: overflow) fängt die Zählung beim größten negativen Wert wieder an, das ist -32768. Willst du größere Zahlen als 32767 verarbeiten, benutze statt dessen den Variablentyp long (der geht dann bis 2 147 483 647).

Das Rechnen mit gebrochenen Zahlen wird ermöglicht, wenn du die Variable als float (floating point = „fließender Punkt“) deklarierst. Entsprechende float-Zahlen, die du ins Programm eingibst, müssen den Dezimalpunkt enthalten (nicht das deutsche Dezimalkomma). Beispiel:

```
float a = 3.0, b = 5.0, c = 10.0; a / b * c ergibt 6.0
```

Benutze den Typ float möglichst nur, wenn du mit integer-Variablen nicht auskommst, denn float-Variablen verbrauchen doppelt so viel Speicherplatz wie int-Variablen und viele Programmfunktionen erfordern int-Werte.

Informationen über alle möglichen Typen für Variablen findest du im Anhang.

Hilfsmotoren

Für weitere Aufgaben des Roboters (einen Gegenstand wegschieben, festhalten oder anheben) brauchst du zusätzliche Motoren. Das können weitere Schrittmotoren sein, die ähnlich wie die Fahrmotoren angesteuert werden. Dazu gibt es einen anders programmierten Motor Controller der Version D und die Bibliotheks-Klasse MotorsX.

Die Beschreibungen zur Klasse MotorsX findest du im Anhang.

Komfortabler in der Anwendung und Programmierung ist jedoch ein so genannter Servo-Motor, wie er auch in ferngesteuerten Modellautos für die Lenkachse eingesetzt wird. Willst du für deinen Roboter Lego-Technic Bauteile verwenden, ist der „Geek-Servo“ besonders interessant: Er hat Befestigungslöcher für Lego-Lochstangen, zwei Achsanschlüsse für Lego-Kreuzstangen und kann bis zu 360° weit drehen.



Schließe den Servo einfach an die 3-polige Buchse „8“ auf dem Master Controller an. Zur Bedienung brauchst du wieder die Bibliothek anadigMaster, jetzt mit der Klasse ServoMotor. Zur Nutzung ist der Befehl `ServoMotor servo(GEEK, 8);` einzubinden.

Dann gibt es folgende Funktionen:

```
servo.turnTo(winkel);           Bewege die Achse schnell bis zum angegebenen Winkel in Grad  
servo.slowTo(winkel, speed);    Bewege die Achse langsam bis zum angegebenen Winkel in Grad
```

Mit dem Parameter speed gibst du die Drehgeschwindigkeit in Grad/Sekunde vor. Willst du z.B. eine 180° Drehung in 2 Sekunden ausführen, schreibe: `servo.slowTo(180, 90);`

Der Servo weiß immer genau, auf welchem Winkel er steht und dreht mit dem nächsten Befehl von dort aus bis zum benannten Winkel weiter.

Experimentiere einfach mit unterschiedlichen Befehlen und Werten.

Fortgeschrittene Techniken

Unterprogramme

Brauchst du bestimmte Befehlsfolgen öfter in deinem Programm, lohnt es sich, diese in ein „Unterprogramm“ zu stecken. Zum Beispiel wurde im Abschnitt „Roboter in Action“ die Fahrt mit den Befehlen `setAccelerations`, `setSpeed`, `setSteering`, `setTargetSteps`, `go` und `wait`, jeweils mit bestimmten Werten ausgeführt.

Kommen in deinem gesamten Programm mehrere solche Abläufe vor, müssten sie auch entsprechend oft eingefügt werden. Statt dessen programmieren wir außerhalb von `setup` und `loop` ein Unterprogramm mit einem neuen Namen und verwenden für jede Teilstrecke nur den Unterprogramm-Namen mit Werten für Strecke, Geschwindigkeit und Lenkung.

Ein Unterprogramm wird für unseren Zweck wie folgt eingeleitet:

```
void Robert_fahre (int strecke, int geschwindigkeit, int lenkung) {...}
```

Das erste Wort „`void`“ besagt, dass kein Wert zurückgeliefert wird (anders später bei Funktionen). Innerhalb des Klammerpaares `{ }` schreiben wir dann die Befehle zu dessen Ausführung, z.B.:

```
Robert.setAccelerations();  
Robert.setSpeed(geschwindigkeit);  
Robert.setSteering(lenkung);  
Robert.setTargetSteps(strecke);  
Robert.go();  
delay(2000);  
Robert.coast();
```

Um das Unterprogramm zu starten, schreibst du in dein Roboterprogramm in `setup()` oder `loop()` einfach:

```
Robert_fahre (550, 80, 3);
```

und er bewegt sich dann 550 Motor-Schritte weit mit der Geschwindigkeit 80 cm/s und leicht nach rechts gelenkt (Lenkwert = 3).

Möchtest du, dass die Strecke direkt in mm eingesetzt werden kann, und hast z.B. vorher ermittelt, dass für einen mm Fahrt zwei Motor-Schritte notwendig sind, kannst du auch direkt schreiben:

```
Robert.setTargetSteps(2 * strecke);
```

Vielleicht möchtest du die Geschwindigkeit in den meisten Fällen auf einen Wert 100 setzen und meistens auch nicht lenken, dann kannst du diese Werte als Vorgaben in die Unterprogramm-Einleitung schreiben und brauchst diese nur noch zu beziffern, wenn diese Parameter davon abweichen sollen:

```
void Robert_fahre (int strecke, int geschwindigkeit = 100, int lenkung = 0) {...}
```

Der Aufruf kann dann einfach durch `Robert_fahre(275);` erfolgen.

Oder mit verringerter Geschwindigkeit: `Robert_fahre(275, 50);`

In beiden Fällen fährt er geradeaus, da die Vorgabe `lenkung = 0` gesetzt wird.

Eine weitere Verbesserung innerhalb des Unterprogramms betrifft die abschließende Wartezeit (delay) in Millisekunden. Soll Robert recht weit fahren, reichen vielleicht die 2 Sekunden nicht aus, oder bei einer kurzen Strecke wartet er unnötig lange.

Um diese Zeit an die Fahrt anzupassen, benutzen wir aus der Klasse Drivetrain den Befehl wait(). Er hält das Programm solange an, bis die Motoren stehen.

Anstatt delay(2000) wird also der Befehl Robert.wait(); verwendet

So: Deine Aufgabe besteht nun darin, das Unterprogramm Robert_fahre mit allen Teilen, die du hier kennengelernt hast, zusammen zu fügen und zu testen.

Nun kommen wir zu den selbst definierten Funktionen. Solche hast du schon oben mit farbsensor.color() und farbsensor.intens() kennen gelernt. Sie lieferten eine Zahl zurück, mit der man die Farbe und Intensität ermitteln und abfragen konnte.

Solche Funktionen kannst du selber schreiben. Sie sind auch eine Art Unterprogramme. Aber anstatt mit void zu beginnen, schreibt man den Variablen-Typ der Zahl, die zurück geliefert wird, z.B. int. Die Lieferung des Ergebnisses erfolgt innerhalb des Unterprogramms mit dem Wort return.

Nehmen wir ein einfaches Beispiel aus der Mathematik. Die so genannte Signum-Funktion (sgn) liefert einem das Vorzeichen einer Zahl: +1 wenn sie positiv ist, -1 wenn sie negativ ist und 0, wenn sie null ist:

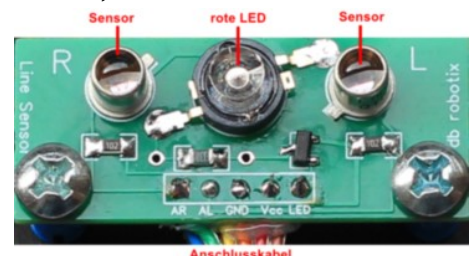
```
int sgn(int n) {  
    if (n > 0) return 1;  
    else if (n < 0) return -1;  
    else return 0;  
}
```

In der objektorientierten Programmierung werden Unterprogramme gewöhnlich “Methoden” genannt, ich verwende hier jedoch die traditionellen Begriffe „Funktionen“ und „Unterprogramme“.

Linienverfolgung

Um an einer dunklen Linie auf hellem Untergrund entlang zu fahren, braucht Robert Licht und Augen. Diese stellt das Modul „Linienfolger“ zur Verfügung. Das Licht wird durch eine rote Leuchtdiode erzeugt, die beiden Augen (Sensoren) sind lichtempfindliche Bauelemente (hier Foto-Transistoren). Die Sensoren haben einen Abstand von 2 cm zueinander, so dass der eine etwas nach links schaut, der andere etwas nach rechts. Die Leuchtdiode wird immer nur kurz eingeschaltet, solange die Messung stattfindet, daher blinkt sie im Betrieb.

Das Linienverfolgungs-Modul hat jedoch keinen I2C-Anschluss, sondern ein Kabel mit einem 6-poligen Stecker, der mit einem ADC/DIO-Anschluss des Master Controllers verbunden wird.



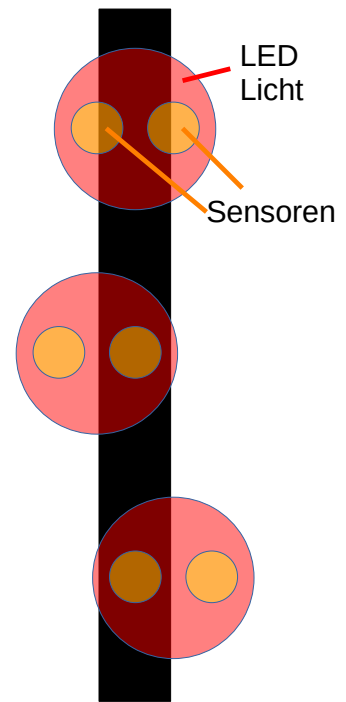
ADC steht für Analog Digital Conversion (für die Fototransistoren) und DIO für Digital Input Output (für die Leuchtdiode).

Nehmen wir eine schwarze Linie auf weißem Untergrund an, sieht der Linienfolger je nach Position des Roboters diese Bilder.

Roboter befindet sich auf der Linien-Mitte:
Beide Sensoren erhalten gleich viel Licht, die Differenz ist Null.
Der Roboter braucht nicht zu lenken, Lenkwert = 0

Roboter steht links von der Linie:
Der linke Sensor erhält mehr Licht als der rechte,
die Differenz $L - R$ ist positiv.
Der Roboter muss nach rechts lenken, Lenkwert > 0

Roboter steht rechts von der Linie:
Der rechte Sensor erhält mehr Licht als der linke,
die Differenz $L - R$ ist negativ.
Der Roboter muss nach links lenken, Lenkwert < 0



In der Bibliothek anadigMaster gibt es eine Klasse LineSensor mit der Funktion getDiff(), die uns genau diese Lichtdifferenz zurück gibt:

```
LineSensor liniensensor;  
d = liniensensor.getDiff();
```

liniensensor ist der selbst gewählte Name des Klassen-Objekts und d die Variable, in der die Differenz gespeichert wird.

Wie du an den drei Beispielen erkennen kannst, brauchst du einen Lenkwert, der das gleiche Vorzeichen hat wie die Differenz der Lichtsignale. Und je größer diese Differenz ist, desto mehr sollte der Roboter gegenlenken. Allerdings kann die Differenz $L - R$ mehrere Hundert betragen, zum Lenken brauchen wir aber vielleicht nur einen Wert von 10. Denn wenn Robert zu stark gegenlenkt, kommt er vielleicht ins Schleudern (wie bei einem Auto). Wir lösen das, indem die Differenz mit einem kleinen Faktor multipliziert wird, z.B. 0.05. Dazu brauchen wir jetzt eine Variable vom Typ float:

```
float kp = 0.05;  
int lenkung;  
lenkung = (int) (kp * liniensensor.getDiff());
```

Dieser verwendete Faktor wird meist kp genannt, was soviel bedeutet wie „Konstante proportional“. Weil kp float ist, lenkung jedoch integer, muss nach der Berechnung der Variablentyp umgewandelt werden, was mit dem voran gestellten (int) möglich ist. Es wird zunächst das Produkt $kp * liniensensor.getDiff()$ ausgerechnet und dann werden die Nachkommastellen abgeschnitten.

Fährt der Roboter jetzt in der Nähe der Linie, kann sich die Licht-Differenz ändern, sie wird z.B. kleiner, wenn der Roboter sich der Linie nähert. Dementsprechend muss sich auch der Lenkwert beim Fahren ändern.

Dazu benötigst du eine Schleife, die solange durchlaufen wird, bis Robert an seiner Zielposition angekommen ist. Weil bei jedem Schleifendurchgang der Lenkwert angepasst wird (je nach Ergebnis

des Linienfolgers), spricht man von einem „Regelkreis“. Die Lenkung wird so nachgeregelt, dass Robert versucht, die Mitte der Linie zu erreichen.

Das Fahr-Programm wird wieder mit Robert_fahre eingeleitet. Du musst allerdings aus dem bisherigen Unterprogramm die Befehle delay(2000); und Robert.coast(); heraus nehmen, damit während der Regelschleife die Motoren noch laufen.

Ein möglicher Programmteil sieht dann so aus:

```
Robert_fahre (500, 50); // starte eigenes Unterprogramm: fahre 500 mm weit mit 50 cm/s
delay(10); // warte etwas, damit Robert in Bewegung kommt
while (Robert.isRunning()) { // frage ab, ob Robert sich noch bewegt
    lenkung = (int) (kp * liniensensor.getDiff()); // berechne den Lenkwert als int-Zahl
    Robert.setSteering(lenkung); // verändere den Lenkwert
    delay(1); // warte 1 ms, damit die Schleife nicht zu schnell durchlaufen wird
} // Ende der Schleife
```

Hier sieht du ein neues Programmierelement:

Nach jedem Befehl steht // und dahinter ein erklärender Text. Der doppelte Schrägstrich wird in C verwendet, um einen Kommentar hinzu zu fügen. Damit erinnerst du dich später, was du hier getan hast, oder teilst anderen Programmierern mit, was hier geschieht.

Teste dein Programm, indem du Robert in der Nähe einer Linie auf einem Spielfeld startest. Führt er eine Schlangenlinie, lenkt er zu stark und musst du kp verkleinern.

Um sicher zu stellen, dass der Lenkwert nicht zu groß wird, sollte er z.B. auf das Intervall -10 bis +10 begrenzt werden. Arduino C stellt eine komfortable Funktion für diesen Zweck zur Verfügung:

```
lenkung = constrain(lenkung, -10, 10);
```

Wenn du den Roboter etwas schräg auf die Linie stellst, kann es vorkommen, dass er nicht schnell genug gegenlenkt und so von der Linie abkommt. Deshalb ergänzen wir die Regelung mit einem zusätzlichen Summanden, der davon abhängt, wie schnell sich die Liniensensor-Differenz ändert. Das nennt man differentiell Verhalten und der zugehörige Faktor ist kd (Konstante differentiell).

Das Programm wird jetzt etwas komplizierter: Wir brauchen mehrere neue Variablen, eine die am Anfang der while-Schleife die aktuelle Differenz zwischenspeichert (z.B. int sensor), und eine die am Ende der Schleife diesen Wert für den nächsten Durchlauf bewahrt (z.B. int sensor0).

Der Lenkwert wird dann zu:

```
lenkung = kp * sensor + kd * (sensor - sensor0);
```

Zusätzlich möchten wir, dass der I2C-Bus nicht zu stark belastet wird: Der Lenkwert wird bisher in jedem Schleifendurchlauf (etwa alle 1 ms) neu übertragen, auch wenn er sich nicht geändert hat. Daher speichern wir am Schleifenende den letzten Wert in einer Variablen lenkung0 und ergänzen Robert.setSteering(lenkung); mit der Abfrage:

```
if (lenkung != lenkung0) Robert.setSteering(lenkung);
```

Kannst du das Gesamt-Programm schon selber zusammen basteln? Starte mit einem Wert kd = 0.1. Wenn du noch Schwierigkeiten hast, schaue in das Musterprogramm vor dem Anhang.

Man spricht hier allgemein von einer PD-Regelung, da sich die Lenkung aus einem proportionalen Anteil und einem differentiellen Anteil zusammen setzt. In der Literatur wird häufig ein PID-Regler vorgeschlagen, der noch zusätzlich einen integrierenden Anteil enthält. Dieser ist jedoch für eine Lenk-Regelung unnötig und sogar kontraproduktiv.

Farberkennung

Die Bibliotheksfunktion `getColor` liefert vielleicht in bestimmten Umgebungen nicht zuverlässig das gewünschte Ergebnis. Möglicherweise sind auch noch weitere Farben (schwarz, orange, braun) oder mehrfarbige Objekte im Spiel.

In solchen Fällen wirst du möglicherweise selber ein Unterprogramm schreiben müssen, in dem du die Original-Messwerte des Sensors, die RGB-Anteile, geeignet verarbeitest. Nach einer Messung mittels `getRGB()` sind sie in den Klassen-Variablen `r` und `g` und `b` abgespeichert, z.B. `klassenname.r`

Manchmal helfen schon die zusätzlichen Funktionen der Bibliotheks-Klasse `ColorSensorA` weiter. Sie basieren auf dem HSI-Farbmodell:

| | |
|---------------------------|---|
| <code>hue()</code> | liefert den gemessenen Farbton als Winkel (-180° ... +180°) |
| <code>saturation()</code> | liefert den gemessenen Sättigungs-Wert (0 ... 100) |
| <code>intens()</code> | liefert die Helligkeit (0 ... ca. 1000) |

Mit Hilfe des Farbtons (`hue`) lassen sich viele Farben unterscheiden (Bild rechts). Das empfangene Licht darf jedoch nicht zu schwach und nicht zu weiß sein, sonst wird das Verfahren unzuverlässig.

Die Sättigung (`saturation`) gibt an (in Prozent), wie stark die eigentliche Farbe hervor sticht. Eine reine grelle Farbe hat theoretisch eine Sättigung von 100. Praktisch lässt sich dieser Wert mit unserem Farbsensor nicht ganz erreichen (typisch 50 ... 70). Ist die Farbe eher blass, sinkt die Sättigung deutlich (typisch 10 ... 30). Praktisch lässt sich mit `saturation` auch ermitteln, wie groß der Weißanteil ist. Ideales Weiß hat gleiche Farbanteile ($R=G=B$) und eine Sättigung = 0. Auch hierbei zeigt der reale Farbsensor Abweichungen (typisch 10 ... 20).



Auch mit Hilfe der Helligkeit (`intens`) kannst du über einen hohen Wert vielleicht ein weißes Objekt detektieren. Diese hängt jedoch stark vom Abstand zum Sensor ab.

Schreibe ein Programm, das die Werte `R`, `G`, `B` sowie `H`, `S` und `I` kontinuierlich misst und auf die Anzeige bringt.

Teste damit verschiedene Objekte in unterschiedlichen Abständen.

Zeitmessung

Vielleicht möchtest du wissen, wie lange Robert für eine gewisse Roboter-Aufgabe braucht. Das Arduino-C bietet dazu eine nützliche Funktion: `millis()`

Als Ergebnis gibt sie dir die Zeit in 1/1000 Sekunden zurück, die seit dem Einschalten des Roboters vergangen ist.

Da die Werte nach vielen Minuten sehr groß werden können, benutzt `millis()` den Rückgabetypp „unsigned long“. Dieser ist zwar auch begrenzt - auf etwa 4 Milliarden Millisekunden, aber keine Sorge: das wären fast 50 Tage! (Probiere es bitte nicht aus, da der Akku bestimmt nicht so lange durchhält und dann nicht mehr zu gebrauchen wäre.)

Willst du die Zeit vom Betätigen des Startknopfes bis zum Ablauf des Roboterprogramms messen, musst du den Startzeitpunkt abspeichern: `unsigned long startzeit = millis();`
und am Ende die Differenz `millis() - startzeit` bilden.

Baue die Zeitmessung in dein eigenes Roboterprogramm ein und zeige die Laufzeit auf dem Display in Sekunden an.

Ultraschall

Hierunter versteht man Schallwellen, die von Menschen nicht hörbar sind. Es lassen sich damit Abstände zu Objekten messen, die den Schall reflektieren. Zum Beispiel kannst du herausfinden, wie weit dein Roboter von einer Wand entfernt ist – ähnlich wie die Einparkhilfe beim Auto mit Ultraschall-Sensoren, die in der Stoßstange verbaut sind.

Fertige arduino-geeignete Ultraschall-Module besitzen vier Anschlüsse: Masse (Gnd), Versorgungsspannung (Vcc), Trigger und Echo. Mit einer Triggerspannung von einem Arduino-Ausgang wird ein Ultraschallsignal gesendet. Sobald das Mikrofon den reflektierten Ultraschall empfängt, sendet das Modul über den Echo-Anschluss eine Spannung, die von einem Arduino-Digitaleingang detektiert wird.

Zur Verbindung mit dem Master Controller verwendest du die untere 6-polige ADC/DIO-Buchse:

Zum Glück gibt es wieder etwas Passendes in der Arduino-Bibliothek `analogMaster`. Die Klasse heißt `UltrasonicSensor` und die Messfunktion (in mm) `getDistance();`

Mit dieser Information kannst du sicherlich schon ein kleines Testprogramm schreiben, das kontinuierlich die Messwerte auf dem Display anzeigt.



Lösungen der Programmier-Aufgaben

Diese Programme stellen nur eine Möglichkeit für die Realisierung vor. Abweichungen im Programmcode sind möglich, solange das Programm die Aufgabe richtig umsetzt.

Wartestellung

```
#include <anadigMaster.h>
Led LED;
void setup() {
}
void loop() {
    LED.on();
    delay(500);
    LED.off();
    delay(500);
    LED.on();
    delay(500);
    LED.off();
    delay(500);
    LED.on();
    delay(500);
    LED.off();
    delay(3000);
}
```

Farbe und Helligkeit

```
#include <i2cMaster.h>
Display anzeige;
ColorSensorA farbsensor;
void setup() {
    Wire.begin();
    anzeige.start();
    farbsensor.start();
}
void loop() {
    farbsensor.getRGB();
    anzeige.clear();
    anzeige.print("Farbcode: ");
    anzeige.println(farbsensor.color());
    anzeige.print("Helligkeit: ");
    anzeige.println(farbsensor.intens());
    delay(1000);
}
```

Schleifen

```
#include <i2cMaster.h>
Display anzeige;
ColorSensorA farbsensor;
void setup() {
    Wire.begin();
    anzeige.start();
    for (int i = 0; i < 20; i++) {
        farbsensor.getRGB();
        anzeige.clear();
        anzeige.print("Helligkeit: ");
        anzeige.println(farbsensor.intens());
        delay(1000);
    }
}
void loop() {}
```

Roboter in Action

```
#include <i2cMaster.h>
Drivetrain Robert(4);
void setup() {
    Wire.begin();
    Robert.setAccelerations();
    Robert.setSpeed(50);
    Robert.setSteering(0);
    Robert.setTargetSteps(400);
    Robert.go();
    delay(2000);
    Robert.coast();
}
void loop() {}
```

Hilfsmotoren

```
#include <anadigMaster.h>
ServoMotor servo(GEEK, 8);
void setup() {
    servo.turnTo(90);
    delay(1000);
    servo.turnTo(270);
    delay(1000);
    servo.turnTo(180);
    delay(1000);
    servo.turnTo(360);
    delay(1000);
    servo.slowTo(0, 90);
    delay(5000);
}
```

```
}  
void loop() {}
```

Unterprogramme

```
#include <i2cMaster.h>  
Drivetrain Robert(4);  
void Robert_fahre(int strecke, int geschwindigkeit=100, int lenkung=0) {  
    Robert.setAccelerations();  
    Robert.setSpeed(geschwindigkeit);  
    Robert.setSteering(lenkung);  
    Robert.setTargetSteps(2 * strecke);  
    Robert.go();  
    delay(10);  
    Robert.wait();  
}  
void setup() {  
    Wire.begin();  
    Robert_fahre(500);  
    Robert.coast();  
}  
void loop() {}
```

Linienverfolgung

```
#include <i2cMaster.h>  
#include <analogMaster.h>  
Drivetrain Robert(4);  
LineSensor liniensensor;  
void Robert_fahre(int strecke, int geschwindigkeit=100, int lenkung=0) {  
    Robert.setAccelerations();  
    Robert.setSpeed(geschwindigkeit);  
    Robert.setSteering(lenkung);  
    Robert.setTargetSteps(2 * strecke);  
    Robert.go();  
}  
void setup() {  
    Wire.begin();  
    float kp = 0.05;  
    float kd = 0.1;  
    int lenkung;  
    int lenkung0 = 0;  
    int sensor;  
    int sensor0 = 0;  
    Robert_fahre(500, 50);  
    delay(10);  
    while (Robert.isRunning()) {  
        sensor = liniensensor.getDiff();  
        lenkung = (int) (kp * sensor + kd * (sensor - sensor0));  
    }  
}
```



```

    lenkung = constrain(lenkung, -10, 10);
    if (lenkung != lenkung0) Robert.setSteering(lenkung);
    sensor0 = sensor;
    lenkung0 = lenkung;
    delay(1);
}
Robert.coast();
}
void loop() {}

```

Farberkennung

```

#include <i2cMaster.h>
Display anzeige;
ColorSensorA farbsensor;
void setup() {
    Wire.begin();
    anzeige.start();
    farbsensor.start();
}
void loop() {
    anzeige.clear();
    farbsensor.getRGB();
    anzeige.print("RGB: ");
    anzeige.print(farbsensor.r);
    anzeige.print(" ");
    anzeige.print(farbsensor.g);
    anzeige.print(" ");
    anzeige.println(farbsensor.b);
    anzeige.print("H: ");
    anzeige.println(farbsensor.hue());
    anzeige.print("S: ");
    anzeige.println(farbsensor.saturation());
    anzeige.print("I: ");
    anzeige.println(farbsensor.intens());
    delay(1000);
}

```

Zeitmessung

```

#include <i2cMaster.h>
Display anzeige;
Drivetrain Robert(4);
void Robert_fahre(int strecke, int geschwindigkeit=100, int lenkung=0) {
    Robert.setAccelerations();
    Robert.setSpeed(geschwindigkeit);
    Robert.setSteering(lenkung);
    Robert.setTargetSteps(2 * strecke);
    Robert.go();
}

```

```

    Robert.wait();
}
unsigned long startzeit;
void setup() {
    Wire.begin();
    anzeige.start();
    startzeit = millis();
    Robert_fahre(1000, 50);
    Robert_fahre(300, 30, 10);
    Robert_fahre(500, -20);
    anzeige.print("Zeit: ");
    anzeige.print((millis() - startzeit) / 1000);
    anzeige.println(" s");
}
void loop() {}

```

Ultraschall

```

#include <i2cMaster.h>
#include <analogMaster.h>
Display anzeige;
UltrasonicSensor ultraschall;
void setup() {
    Wire.begin();
    anzeige.start();
}
void loop() {
    anzeige.clear();
    anzeige.print("Abstand: ");
    anzeige.print(ultraschall.getDistance());
    anzeige.println(" mm");
    delay(1000);
}

```

Häufige Programmierfehler:

- Fehlendes Semikolon am Ende eines Befehls, jedoch nicht nach #include, #define oder schließender Klammer }
- Fehlende schließende Klammer } nach einem Block, { } existieren immer paarweise !
- Funktionen aus Bibliotheken ohne Nennung des Klassenobjekts + Punkt, z.B. robotDrive. oder display.
- Innerhalb einer for-Anweisung müssen Semikolons stehen, keine Kommas: for (i=0 ; i<10 ; i++)
- Nachkommastellen von Dezimalzahlen müssen mit einem Punkt abgetrennt werden, nicht mit Komma: 3.1415
- Falsche Ergebnisse bei der integer-Division (siehe Abschnitt „Variablen“)

Anhang

Einrichtung der Arduino IDE

Programmierung des Arduino Zero

C-Befehlsreferenz

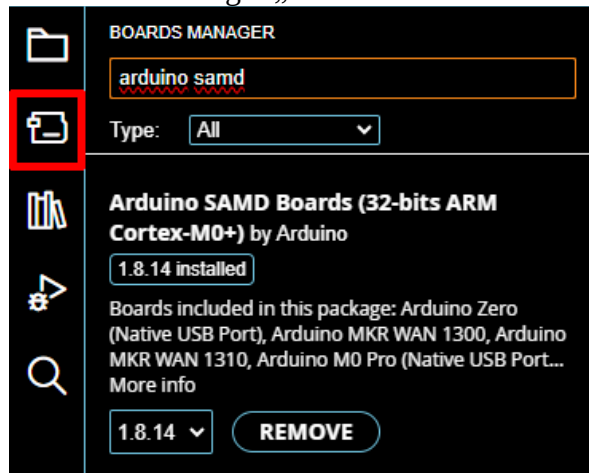
Bibliotheken-Befehlsreferenz

Einrichtung der Arduino IDE

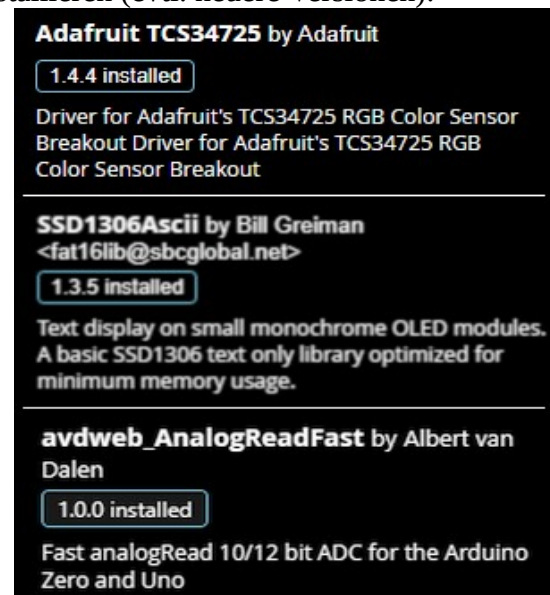
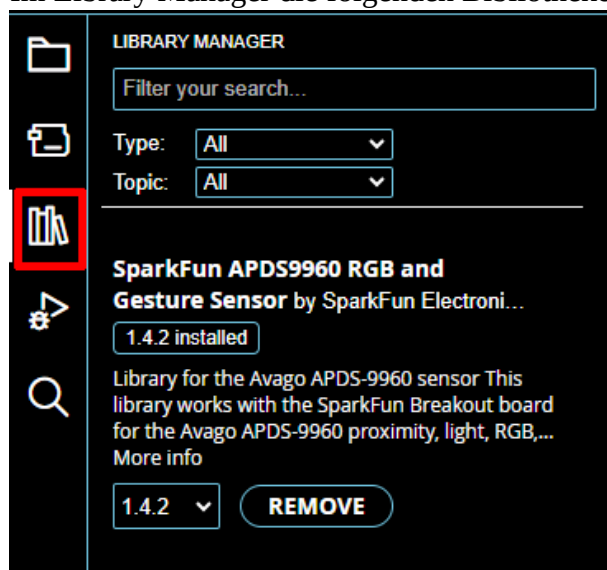
Die aktuelle Version der Arduino IDE (<https://www.arduino.cc/en/software>) aus dem Internet herunterladen und das Programm installieren.

Konfigurationen innerhalb der IDE (Internet-Verbindung erforderlich)

Im Boards Manager „Arduino SAMD Boards“ installieren:



Im Library Manager die folgenden Bibliotheken installieren (evtl. neuere Versionen):



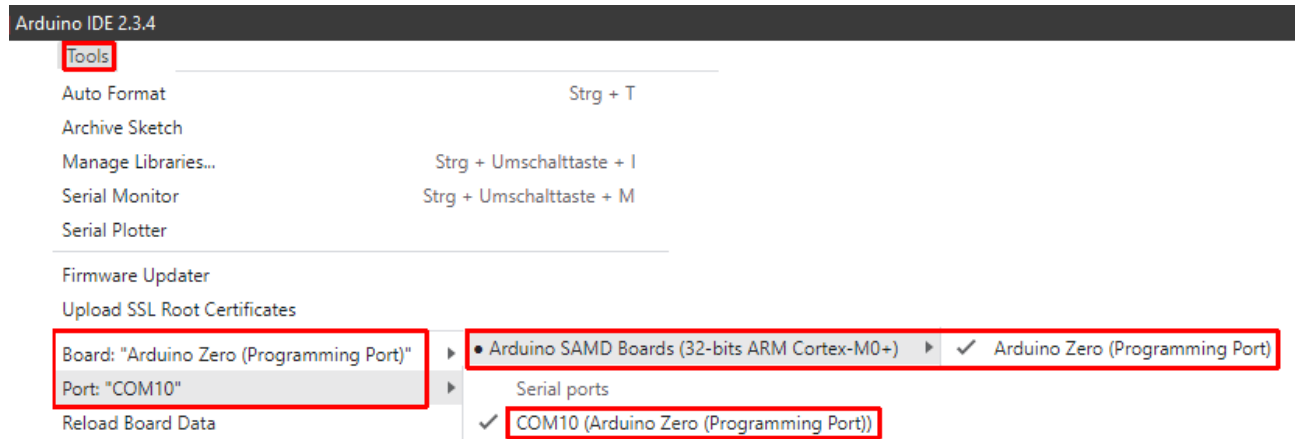
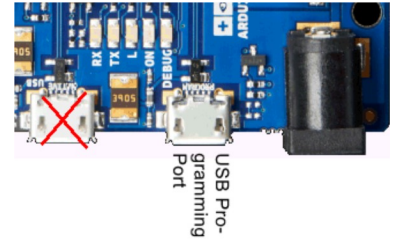
Außerdem sind in den Arduino-Programmordner unter „libraries“ noch zwei Bibliotheken einzubinden. Dazu zwei Unterordner generieren:

- i2cMaster
- anadigMaster

und die Dateien aus dem Internet (<https://github.com/db-robotix/Master-Controller>) herunterladen und dort hinein kopieren.


Programmierung des Arduino Zero


1. Den Controller am „USB Programming Port“ (Micro USB Buchse) über ein USB-Kabel an den Computer anschließen (s. Rechts) und den Akku einschalten.
2. Unter „Tools“ das Arduino Zero Board und den COM-Port (Nummer ist computer-abhängig) auswählen:

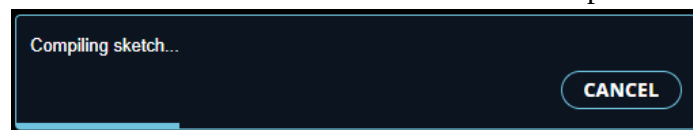


3. Die USB-Verbindung trennen und wieder anschließen. Anhand der Statusmeldung unten rechts nach wenigen Sekunden überprüfen, ob die Verbindung funktioniert:

Arduino Zero (Programming Port) on COM10

4. Falls dort der Zusatz **[not connected]** verbleibt, nochmals die USB-Verbindung neu aufbauen.
5. Zum Schreiben des Programms kann die USB-Verbindung getrennt werden.
6. Programm in der Arduino IDE schreiben. Gegebenenfalls testen (compilieren) mit 
7. Zur Übertragung an den Controller zuerst die USB-Verbindung aufbauen! Beachte die Hinweise unter 3 und 4. Der Akku muss dazu eingeschaltet sein!

8. Den Upload mit  einleiten. Der Vorgang kann einige Sekunden dauern. Zuerst erscheint unten rechts ein Statusfenster mit einem Fortschrittsbalken des Compilervorgangs:



9. Später erscheint dort der Text „Uploading ...“ und danach „Done Uploading“.
10. Währenddessen werden im Output unten links verschiedene Meldungen generiert:

11. Falls dort Fehlermeldungen angezeigt werden, das Programm entsprechend korrigieren.

```
** Programming Started **  
auto erase enabled  
wrote 25600 bytes from file C:\Users\Home\AppData\  
** Programming Finished **  
** Verify Started **  
verified 25396 bytes in 2.097362s (11.825 KiB/s)  
** Verified OK **  
** Resetting Target **  
shutdown command invoked
```

12. Zum Testen des Programms auf dem Controller kann die USB-Verbindung getrennt werden. Schließe aber die externe Spannungsversorgung (z.B. Akku) an.

C-Befehlsreferenz

Die folgende Übersicht zeigt eine Auswahl der wichtigsten Befehle aus den Programmiersprachen C und C++, sowie Erweiterungen der Arduino IDE.

Zu Zufallszahlen, Zeichenketten = Strings, Bitoperationen, Interrupts, I2C-Kommunikation usw. siehe entsprechende Literatur.

Grundstruktur des Arduino-Programms:

| | |
|--|---|
| <code>void setup();</code> | wird einmalig nach dem Programmstart durchlaufen |
| <code>void loop();</code> | wird kontinuierlich wiederholt |
| <code>//</code> | Kommentar bis zum Ende der Zeile |
| <code>/* kommentar */</code> | Kommentar bis <code>*/</code> kommt |
| <code>#include <bibliothek.h></code> | Einbinden einer Bibliothek (aus dem Ordner namens library) |
| <code>#include „datei.h“</code> | Einbinden einer Bibliothek oder C-Datei aus gleichem Ordner |
| <code>#define xxx yyy</code> | jedes xxx wird beim Compilieren durch yyy ersetzt |
| <code>{ ... }</code> | Anweisungs-Block |

Kontrollstrukturen:

| | |
|---|--|
| <code>for (int i = 0; i < n; i++) { }</code> | Schleife mit n Durchläufen (i = 0 ... n-1) |
| <code>while (bedingung) { }</code> | Schleife wird solange durchlaufen, wie Bedingung wahr ist |
| <code>do { } while (bedingung);</code> | Schleife wird mind. einmal durchlaufen, solange Bedingung wahr |
| <code>if (bedingung) { }</code> | falls Bedingung wahr ist, führe Befehle in { } aus |
| <code>else { }</code> | anderenfalls führe diese Befehle aus |
| <code>switch (variable) {</code> | Überprüfe Integer-Variable |
| <code>case 1: ...</code> | falls sie den Wert 1 besitzt, führe folgende Befehle aus |
| <code>break;</code> | bis break kommt. |
| <code>case 2: ...</code> | falls sie den Wert 2 besitzt, führe folgende Befehle aus |
| <code>break;</code> | bis break kommt. |
| <code>default: ...</code> | falls kein case zutrifft, führe dieses aus |
| <code>}</code> | |
| <code>return;</code> | Verlasse Unterprogramm void |
| <code>return x;</code> | Gib den Wert x aus dem Unterprogramm zurück |

Mathematische Operatoren:

| | |
|-------------------------------------|---|
| <code>+ - * / ()</code> | wie in der Mathematik |
| <code>%</code> | Modulo-Funktion (Rest der ganzzahligen Division) |
| <code>i++;</code> | erhöhe i um 1 |
| <code>i--;</code> | verringere i um 1 |
| <code>i+=n</code> | erhöhe i um n |
| <code>=</code> | Zuweisung eines Werts zu einer Variablen (z.B. x = 5) |
| <code>< <= > = ></code> | Vergleiche |
| <code>==</code> | Prüfen auf Gleichheit |
| <code>!=</code> | Prüfen auf Ungleichheit |
| <code>!</code> | logisches NICHT (Negation) |
| <code>&&</code> | logisches UND |
| <code> </code> | logisches ODER |

Mathematische Funktionen

| | |
|---|--|
| <code>sqrt(x)</code> | Quadratwurzel |
| <code>pow(basis, exponent)</code> | Potenzfunktion |
| <code>sin(x), cos(x), tan(x)</code> | Winkelfunktionen (Argument in Bogenmaß !) |
| <code>asin(x), acos(x), atan(x)</code> | Umkehr-Winkelfunktionen (Ergebnis in Bogenmaß !) |
| <code>atan2(y, x)</code> | 4-Quadranten-Arcustangens |
| <code>abs(x)</code> | Absolutwert |
| <code>min(x, y)</code> | Minimum der beiden Zahlen |
| <code>max(x, y)</code> | Maximum der beiden Zahlen |
| <code>constrain(x, minimum, maximum)</code> | Ergebnis begrenzt auf Intervall von minimum bis maximum |
| <code>map(x, x1, x2, y1, y2)</code> | $x=x1$ wird zu $y1$, $x=x2$ wird zu $y2$, ansonsten linear erweitert |

Hardware-Funktionen des Arduino

| | |
|--|--|
| <code>pinMode(pin, INPUT);</code> | Benutze pin als digitalen Eingang |
| <code>pinMode(pin, INPUT_PULLUP);</code> | Benutze pin als digitalen Eingang, HIGH wenn offen |
| <code>digitalRead(pin);</code> | Lese Zustand des Eingangs (Ergebnis: LOW = 0 oder HIGH = 1) |
| <code>pulseIn(pin, status, dauer);</code> | Warte auf Status (HIGH, LOW) des pin-Eingangs, max. Dauer μ s Rückgabe in Mikrosekunden |
| <code>pinMode(pin, OUTPUT);</code> | Benutze pin als digitalen Ausgang |
| <code>analogReference(AR_INTERNAL);</code> | Maximale analoge Eingangsspannung = 2.23 Volt |
| <code>analogRead(pin);</code> | Lese analogen Wert vom Eingang (per ADC): 0 ... 1023 |
| <code>millis();</code> | Lese Systemzeit im Millisekunden (max. 50 Tage) |
| <code>micros();</code> | Lese Systemzeit im Mikrosekunden (max. 70 Minuten) |
| <code>delay(ms);</code> | Programm hält für ms Millisekunden an |
| <code>delayMicroseconds(us);</code> | Programm hält für us Mikrosekunden an |

Monitor (angezeigt in der Arduino IDE)

| | |
|-------------------------------------|--|
| <code>Serial.begin(9600);</code> | Übertragung mit 9600 Bit/s |
| <code>Serial.begin(115200);</code> | Schnellste Übertragung mit 115200 Bit/s |
| <code>Serial.print(„text“);</code> | Schreibe Text auf den Monitor |
| <code>Serial.print(zahl);</code> | Schreibe Zahl auf den Monitor |
| <code>Serial.print(zahl, n);</code> | Schreibe Zahl auf den Monitor mit n Nachkommastellen |
| <code>Serial.println(...);</code> | Schreibe auf den Monitor mit nachfolgendem Zeilenumbruch |

Variablendeklaration

| | |
|---|--|
| <code>int i;</code> | Ganzzahl-Variable i |
| <code>int i = 1;</code> | mit Initialisierung |
| <code>float x;</code> | Rationale Zahl x |
| <code>float x = 2.0;</code> | mit Initialisierung (Punkt als Dezimaltrenner !) |
| <code>const pi = 3.1415;</code> | Konstante: kann nicht mehr geändert werden |
| <code>int x_array[n]</code> | Array mit n Elementen (0 ... n-1) |
| <code>int prim[] = {2,3,5,7,11,13,17};</code> | Array mit den ersten Primzahlen |
| <code>x = x_array[i]</code> | liefert i-ten Wert des Arrays (i = 0,1,2, ...) |
| <code>enum name {ZERO, ONE, TWO}</code> | Aufzählung (entspricht den Werten 0,1,2) |

Variablentypen in der Arduino IDE

| <i>Typbezeichnung</i> | <i>Bytes</i> | <i>Wertebereich</i> | <i>Konstanten</i> |
|--------------------------------|--------------|-----------------------------------|-----------------------|
| bool | 1 | false = 0 , true = 1 | false, true |
| uint8_t = byte = char | 1 | 0 ... 255 | zahlU im Wertebereich |
| uint16_t = unsigned int = word | 2 | 0 ... 65535 | zahlU im Wertebereich |
| int16_t = int | 2 | -32768 ... 32767 | Zahl im Wertebereich |
| uint32_t = unsigned long | 4 | 0 ... 4 294 967 295 | zahlUL |
| int32_t = long | 4 | - 2 147 483 648 ... 2 147 483 647 | zahlL |
| uint64_t = unsigned long long | 8 | 0 ... 1.8 E19 | |
| int64_t = long long | 8 | ± 9.2 E18 | |
| float | 4 | ± 3.4 E38 | Zahl mit . |
| Binärzahl | | | 0b..... |
| Hexadezimalzahl | | | 0x.... |

Bibliotheken-Befehlsreferenz

i2cMaster / anadigMaster

Es muss jeweils ein Objekt der entsprechenden Klasse aus der Bibliothek *i2cMaster* bzw. *anadigMaster* erzeugt werden:

klassenname objektname;

Display (Library i2cMaster)

| <i>Funktion</i> | <i>Parameter</i> | <i>Anmerkung</i> |
|-----------------|--------------------|---|
| start() | keine | sollte in setup() stehen |
| clear() | keine | Löschen des Display-Inhalts |
| print(inhalt) | Zahl oder „Text“ | Display-Ausgabe des Inhalts |
| print(zahl, n) | float zahl, byte n | Ausgabe der Zahl mit n Nachkommastellen |
| println(inhalt) | Zahl oder „Text“ | Wie print, mit abschließendem Zeilenumbruch |
| setRow(reihe) | byte reihe | Festlegung der nächsten Zeile (1 ... 4) |

In zeitkritischen Programmteilen für Sensoren oder Motoren sollten möglichst keine Display-Funktionen verwendet werden, da der I2C-Bus zusätzlich belastet wird.

ColorSensorA (Library i2cMaster)

ColorSensorB (Library i2cMaster)

| <i>Funktion</i> | <i>Parameter</i> | <i>Rückgabewert (int16_t)</i> | <i>Anmerkung</i> |
|-------------------|--------------------------------------|-------------------------------|--|
| start() | keine | kein | sollte in setup() stehen |
| start(d) | Digitalport (byte) | kein | nur ColorSensorA Version 2 Standard: d = 13 |
| calibrate() | keine | Attribute r0, g0, b0 | Dunkel-Kalibrierung |
| reset() | keine | kein | r0, g0, b0 auf 0 |
| getRGB() | keine | Attribute r, g, b | Durchführung der Farbmessung |
| getRGB(r, g, b) | Variablennamen r, g, b (uint16_t) | Variablen r, g, b | Farbmessung und Speicherung in den Variablen |
| flashRGB() | keine | Attribute r, g, b | Differenzmessung der Farbanteile, Messdauer 5 ms, nur ColorSensorA Version 2 |
| flashRGB(r, g, b) | Variablennamen r, g, b (uint16_t) | Variablen r, g, b | Differenzmessung und Speicherung in den Variablen, Messdauer 5 ms, nur ColorSensorA Version 2 |
| hue() | keine | Hue-Wert des HSL- Modells | Berechnet aus letzten Messwerten (-179 ... 180) |
| hue(r, g, b) | Farbanteile (uint16_t) | Hue-Wert des HSL- Modells | Berechnet aus den Parametern |

| | | | |
|---------------------|------------------------|-----------------------------------|--|
| saturation() | keine | Sättigung des HSL-Modells | Berechnet aus letzten Messwerten (0 ... 100) |
| saturation(r, g, b) | Farbanteile (uint16_t) | Sättigung des HSL-Modells | Berechnet aus den Parametern |
| color() | keine | Farbcode* | Berechnet aus letzten Messwerten (0 ... 5) |
| color(r, g, b) | Farbanteile (uint16_t) | Farbcode* | Berechnet aus den Parametern |
| intens() | keine | Intensität | Berechnet aus letzten Messwerten (0 ... ca.2500) |
| intens(r, g, b) | Farbanteile (uint16_t) | Intensität | Berechnet aus den Parametern |
| ledOn() | keine | kein | LEDs einschalten, nur ColorSensorA Version 2 |
| ledOff() | keine | kein | LEDs ausschalten, nur ColorSensorA Version 2 |
| blackLimit | (Variable uint16_t) | Intensitäts-Grenzwert für Schwarz | Einfluss auf Ergebnis von color() (default = 40) |
| r | (Variable uint16_t) | Rotanteil | letzter Messwert |
| g | (Variable uint16_t) | Grünanteil | letzter Messwert |
| b | (Variable uint16_t) | Blauanteil | letzter Messwert |

* Farbcodes:

| | | | | | | |
|------------|---------|-----|--------|-------|------|-------|
| Code # | 0 | 1 | 2 | 3 | 4 | 5 |
| Farbe | BLACK | RED | YELLOW | GREEN | BLUE | WHITE |
| alternativ | SCHWARZ | ROT | GELB | GRUEN | BLAU | WEISS |

Battery (Library anadigMaster)

| <i>Funktion</i> | <i>Parameter</i> | <i>Rückgabewert</i> | <i>Anmerkung</i> |
|------------------|------------------|--|-----------------------------|
| getVoltage() | keine | float Spannung | in Volt |
| percent(voltage) | float Spannung | uint16_t Prozent Ladezustand (LiPo 3 Zellen) | 0% = 10.5 V ; 100% = 12.4 V |

Button (Library anadigMaster)

| <i>Funktion</i> | <i>Parameter</i> | <i>Rückgabewert</i> | <i>Anmerkung</i> |
|---------------------------|--|-------------------------------|-------------------------------------|
| pressed() | keine | bool 0 = offen ; 1 = gedrückt | |
| wait(dly) wait() | uint32_t Verzögerung ms default = 0 | kein | Warten bis Taster gedrückt + dly ms |
| count(timeout) count() | uint8_t Timeout sec default = 2 | uint16_t Tasten-Klicks | Beenden nach timeout Inaktivität |

Led (Library anadigMaster)

| <i>Funktion</i> | <i>Parameter</i> | <i>Anmerkung</i> |
|----------------------|--|------------------|
| on() | keine | Schaltet LED ein |
| off() | keine | Schaltet LED aus |
| blink(count, period) | uint8_t Anzahl Blinker, uint16_t Periode in ms | LED blinkt |

LineSensor (Library anadigMaster)

| <i>Funktion</i> | <i>Parameter</i> | <i>Rückgabewert</i> | <i>Werte</i> |
|---------------------------|--|---|-----------------|
| getAmbient(a1, a2) | Variablennamen a1, a2 (int16_t) | Variablen a1, a2 | 1 ... 1023 |
| getReflections(a1, a2) | Variablennamen a1, a2 (int16_t) | Variablen a1, a2 | 1 ... 1023 |
| getDiff() | keine | int16_t Differenz der Reflexionen | a1 - a2 |
| getSum() | keine | int16_t Summe der Reflexionen | a1 + a2 |
| getAmbientDiff() | keine | int16_t Differenz der Lichter | a1 - a2 |
| getAmbientSum() | keine | int16_t Summe der Lichter | a1 + a2 |
| ledOn() | keine | kein | |
| ledOff() | keine | kein | |
| calibrate(wL, wR, sL, sR) | Sensorwerte L/R für weiß/schwarz (nur für getOffset) | kein | 1 ... 1023 |
| getOffset() | keine | int16_t Relative Abweichung von der Linienmitte (links <0, rechts >0) | -1000 ... +1000 |

UltrasonicSensor (Library anadigMaster)

| <i>Funktion</i> | <i>Parameter</i> | <i>Rückgabewert</i> | <i>Werte</i> |
|---------------------------------|---------------------|--|---------------------------------|
| getDistance() getDistance1() | keine | Gemessener Abstand in mm des ersten Sensors | 1 ... 1000 0 = kein Messwert |
| getDistance2() | keine | Gemessener Abstand in mm des zweiten Sensors | 1 ... 1000 0 = kein Messwert |
| soundSpeed | (Variable uint16_t) | Schallgeschwindigkeit m/s | default = 343 |

ServoMotor (Library anadigMaster)

Bei der Objektdefinition muss der Typ angegeben werden (port = 8 oder 9 je nach Controllerbuchse):
ServoMotor servo(GEEK, port); oder ServoMotor servo(MINI, port);

| <i>Funktion</i> | <i>Parameter</i> | <i>Anmerkung</i> |
|-----------------|--------------------|--|
| turnTo(winkel) | int16_t Zielwinkel | Absoluter Winkel gegen UZS GEEK: 0 ... 360 MINI: 0 ... 180 |

| | | |
|-----------------------|-----------------------------------|--------------------------------|
| slowTo(winkel, speed) | int16_t Zielwinkel, int16_t Speed | Drehung mit speed Grad/Sek. |
| coast() | keine | Servomotor stromlos geschaltet |

Drivetrain = Fahrmotoren-Steuerung (Library i2cMaster)

Steuerung der Schrittmotoren durch I2C-Kommandos:

Die Bibliotheks-Funktionen werden über den I2C-Bus übertragen.

Dazu muss ein Objekt der Klasse *Drivetrain* aus der Bibliothek *i2cMaster* erzeugt werden (am Beginn der ino-Datei):

```
#include "i2cMaster.h"
```

```
Drivetrain robotDrive(4); // I2C address
```

Die Motor-Kommandos werden per I2C-Bus übertragen mit Hilfe des sendCommand-Befehls.

| command # | Command (<i>uint8_t</i>) | Bedeutung | Einheit | Wert/Bereich (<i>int16_t</i>) |
|-----------|-------------------------------|------------------------|----------------------|------------------------------------|
| 0 | NONE_S | keine | ----- | 0 |
| 1 | GO | Starte Antrieb | ----- | 0 |
| 2 | STOP | Stoppe Bewegung | ----- | 0 |
| 3 | SPEED | Setze Geschwindigkeit | Steps/s | -2000 ... +2000 * |
| 4 | STEERING | Setze Lenkwert | ----- | -100 ... +100 |
| 5 | ACCEL | Setze Beschleunigung | Steps/s ² | 100 ... 5000 * |
| 6 | DECEL | Setze Bremsverzögerung | Steps/s ² | 100 ... 5000 * |
| 7 | TARGET | Setze Ziel | Steps | 1 ... 32767 |
| 8 | COAST | Setze Freilaufmodus | ----- | 0 |
| 9 | BRAKE | Setze Blockiermodus | ----- | 0 |

* praktische Grenze

Bibliotheks-Funktionen:

| Funktion | Parameter (<i>int16_t</i>) | Bedeutung | Einheit | Bereich (<i>int16_t</i>) | Defaultwert |
|------------------|---------------------------------|--------------------------------------|--|-------------------------------|-------------|
| sendCommand | command, wert | Motor-Kommando über I2C (s. oben) | s. oben | s. Tabelle oben | ----- |
| setTargetSteps | steps | Schritte zum Ziel | steps | -32768 – +32767 | ----- |
| setSpeed | speed | Geschwindigkeit | 20 steps/s cm/s # | -100 – +100 * | ----- |
| setSteering | steering | Lenkwert | ----- | -100 – +100 | ----- |
| setAccelerations | accel, decel | Beschleunigungen | 20 steps/s ² cm/s ² # | 10 – 500 * | 250 |
| go() | ----- | Start der Roboterfahrt | ----- | ----- | ----- |
| stop() | ----- | Sofortiger Stopp | ----- | ----- | ----- |
| brake() | ----- | Motoren im Bremsmodus | ----- | ----- | ----- |
| coast() | ----- | Motoren im | ----- | ----- | ----- |

| | | | | | |
|--------------|--|---|-------|--|-------|
| | | Freilaufmodus | | | |
| getStatus() | ----- | Rückgabe (int16_t) des Fortschritts | steps | 0 – 32767 -1 = Motoren aus -9 = Fehler | ----- |
| isRunning() | ----- | Rückgabe (bool) true, wenn Motoren laufen | ----- | true/false | ----- |
| wait() | ----- | Wartet, solange Motoren laufen | ----- | ----- | ----- |
| estimateTime | distance, speed, accel, decel | Rückgabe (int16_t) der Fahrzeit | ms | wie Einzelparameter | ----- |

mit 200 mm Radumfang und 400 steps/Umdrehung

* praktische Grenze

Nach Start der Fahrbewegung mittels go() lassen sich die Zielschritte (setTargetSteps), Geschwindigkeit (setSpeed) und Beschleunigungen (setAccelerations) nicht mehr verändern. Lediglich der Lenkwert (setSteering) kann während des Laufs variieren (z.B. für eine Linienverfolgung).

getStatus() ist immer abrufbar und die Motoren lassen sich mittels stop() jederzeit stoppen.

MotorsX = Hilfsmotoren-Steuerung (Library i2cMaster)

Die folgenden Funktionen basieren auf Schrittmotoren mit 400 Halbschritten/Umdrehung (0.9 °/step).

Steuerung der Schrittmotoren durch I2C-Kommandos

Die Bibliotheks-Funktionen werden über den I2C-Bus übertragen.

Dazu muss ein Objekt der Klasse *MotorsX* aus der Bibliothek *i2cMaster* erzeugt werden (am Beginn der ino-Datei):

```
#include "i2cMaster.h"
```

```
MotorsX motorAB(5); // I2C address
```

Die Motor-Kommandos werden übertragen mit Hilfe des Befehls:

```
motorAB.sendCommand(command, wert);
```

| command # | command (unit8_t) | Bedeutung | Einheit | Wert/Bereich (int16_t) |
|-----------|-------------------|------------------------|----------------------|------------------------|
| 0 | NONE_X | keine | ----- | 0 |
| 1 | GO_A | Starte Antrieb | ----- | 0 |
| 2 | STOP_A | Stoppe Bewegung | ----- | 0 |
| 3 | SPEED_A | Setze Geschwindigkeit | steps/s | -1200 ... +1200 * |
| 4 | ACCEL_A | Setze Beschleunigung | steps/s ² | 100 ... 10000 * |
| 5 | DECEL_A | Setze Bremsverzögerung | steps/s ² | 100 ... 10000 * |
| 6 | TARGET_A | Setze Ziel | steps | 1 ... 32767 |
| 7 | COAST_A | Setze Freilaufmodus | ----- | 0 |
| 8 | BRAKE_A | Setze Blockiermodus | ----- | 0 |

| | | | | |
|----|----------|------------------------|----------------------|-------------------|
| 9 | GO_B | Starte Antrieb | ----- | 0 |
| 10 | STOP_B | Stoppe Bewegung | ----- | 0 |
| 11 | SPEED_B | Setze Geschwindigkeit | steps/s | -1200 ... +1200 * |
| 12 | ACCEL_B | Setze Beschleunigung | steps/s ² | 100 ... 10000 * |
| 13 | DECEL_B | Setze Bremsverzögerung | steps/s ² | 100 ... 10000 * |
| 14 | TARGET_B | Setze Ziel | steps | 1 ... 32767 |
| 15 | COAST_B | Setze Freilaufmodus | ----- | 0 |
| 16 | BRAKE_B | Setze Blockiermodus | ----- | 0 |

_A, _B entsprechend Motor

* praktische Grenze

Bibliotheks-Funktionen:

| <i>Funktion</i> | <i>Parameter (int16_t)</i> | <i>Bedeutung</i> | <i>Einheit</i> | <i>Bereich (int16_t)</i> | <i>Defaultwert</i> |
|--|--------------------------------|--|----------------------|--|--------------------|
| setTargetSteps_A setTargetSteps_B | steps | Schritte zum Ziel | Steps (0.9°/step) | -32768 – +32767 | ----- |
| setSpeed_A setSpeed_B | speed | Geschwindigkeit | °/s | -1080 – +1080 * | ----- |
| setAccelerations() | ----- | Beschleunigungen | °/s ² | ----- | 5000 |
| setAccelerations_A setAccelerations_B | accel, decel | Beschleunigungen | °/s ² | 100 – 10000 * | ----- |
| go_A() go_B() | ----- | Start der Drehbewegung | ----- | ----- | ----- |
| stop_A() stop_B() | ----- | Sofortiger Stopp | ----- | ----- | ----- |
| brake_A() brake_B() | ----- | Beide Motoren im Bremsmodus | ----- | ----- | ----- |
| coast_A() coast_B() | ----- | Beide Motoren im Freilaufmodus | ----- | ----- | ----- |
| getStatus() | ----- | Rückgabe (int16_t) des Fortschritts | ----- | 0 = beide ruhig 1 = MotorA bew. 2 = MotorB bew. 3 = beide bewegt -9 = Fehler | ----- |

* praktische Grenze